Developing Liferay DXP

A Complete Guide

THE LIFERAY DOCUMENTATION TEAM
Richard Sezov, Jr.
Jim Hinkey
Stephen Kostas
Jesse Rao
Cody Hoag
Nicholas Gaskill
Michael Williams

Liferay Press

Contents

Preface

Welcome to the world of the Liferay DXP development platform! This book was written for anyone who wants to create applications built on Liferay DXP. It contains everything you need to know about Liferay's development tools and projects. You'll learn all you need to know about plugins, OSGi, the Liferay Workspace, Service Builder, and more. Use this book as a handbook for everything you need to do to get your application running on Liferay DXP, and then keep it by your side as you update and add features to help your users work more effectively.

Conventions

The information contained herein has been organized in a way that makes it easy to locate information. The book has two parts. The first part, *Developer Tutorials*, shows you how to work step-by-step with Liferay's technology. The second part, *Developer Reference*, shows exhaustively the options and APIs you need.

Sections are broken up into multiple levels of headings, and these are designed to make it easy to find information.

Source code and configuration file directives are presented monospaced, as below.

```
Source code appears in a non-proportional font.
```

Italics represent links or buttons to be clicked on in a user interface.

`Monospaced type` denotes Java classes, code, or properties within the text.

Bold describes field labels and portlets.

Page headers denote the chapters and the section within the chapter.

Publisher Notes

It is our hope that this book is valuable to you, and that it becomes an indispensable resource as you work with Liferay DXP. If you need assistance beyond what is covered in this book, Liferay offers training[1], consulting[2], and support[3] services to fill any need that you might have.

[1]https://learn.liferay.com
[2]https://www.liferay.com/consulting
[3]https://help.liferay.com

For up-to-date documentation on the latest versions of Liferay, please see the documentation pages on Liferay Learn.[4]

As always, we welcome feedback. If there is any way you think we could make this book better, please feel free to mention it on our forums or in the feedback on Liferay Learn. You can also use any of the email addresses on our Contact Us page.[5] We are here to serve you, our users and customers, and to help make your experience using Liferay DXP the best it can be.

[4]https://learn.liferay.com

[5]https://www.liferay.com/contact-us

Part I

Developer Tutorials

Chapter 1

Introduction to Liferay Development

How many times have you had to start over from scratch? Probably almost as many times as you've started a new project, because each time you have to write not only the code to build the project, but also the underlying code that supports the project. It's never a good feeling to have to write the same kind of code over and over again. But each new project that you do after a while can feel like that: you're writing a new set of database tables, a new API, a new set of CSS classes and HTML, a new set of JavaScript functions.

Wouldn't it be great if there were a platform that provided a baseline set of features that gave you a head start on all that repetitive code? Something that lets you get right to the features of your app or site, rather than making you start over every time with the basic building blocks? There is such a thing, and it's called Liferay DXP.

Figure 1.1: With Liferay DXP, you never have to start from scratch.

1.1 Leveraging a Suite of Products, Frameworks and Libraries

Liferay DXP offers developers a complete platform for building web apps, mobile apps, and web services quickly, using features and frameworks designed for rapid development, good performance, and ease of use. The base platform is already there, and it's built as a robust container for applications that you can put together in far less time than you would from scratch.

It also ships with a default set of common applications you can make use of right away: web experience management, collaboration applications such as forums and wikis, documents and media, blogs, and more. All of these applications are designed to be customized, as is the system itself. You can also extend them to include your own functionality, and this is no hack: because of Liferay's extensible design, customization is by design.

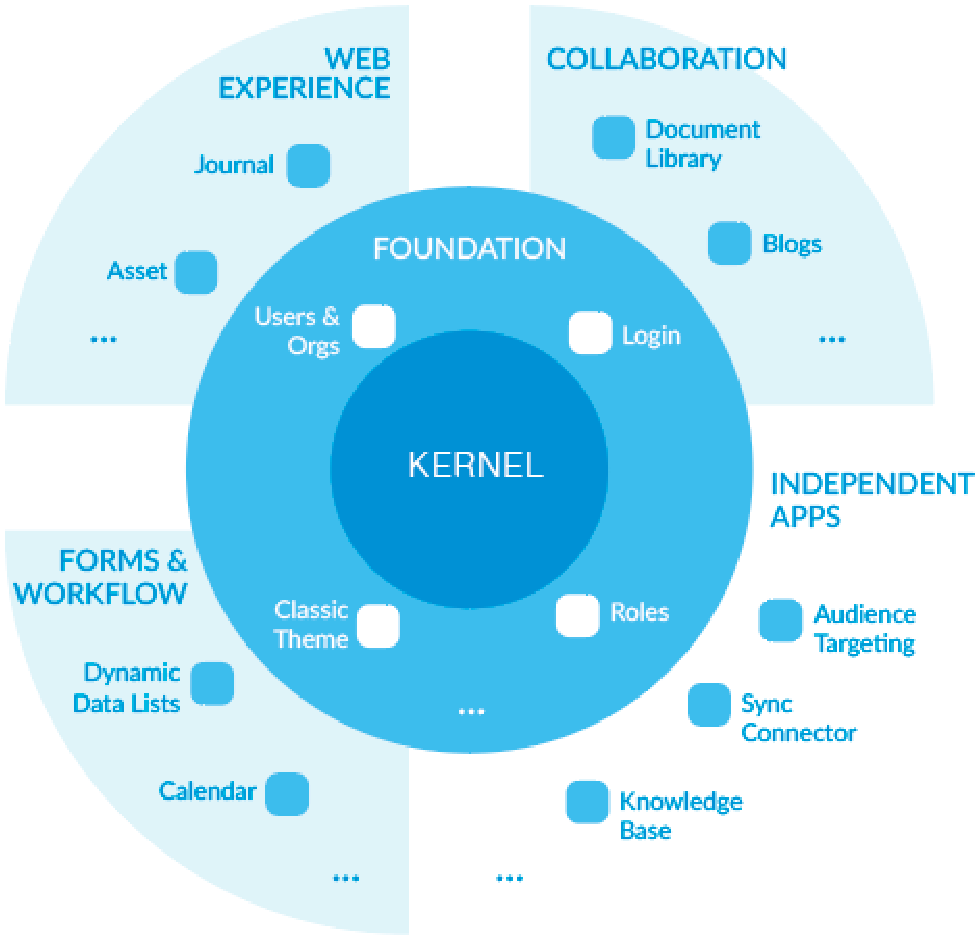


Figure 1.2: Liferay DXP ships with suites of applications to get you started building your site quickly.

In short, Liferay was written by developers for developers, to help you get your work done faster and more easily, to take the drudgery out of web and mobile app development, so that writing code becomes enjoyable again.

1.2 Build Websites, Intranets, Collaborative Environments, Mobile Apps, and More

One of the most often cited best characteristics of Liferay is how versatile it is. It can be used to build websites of all sorts, from very large websites with hundreds of thousands of articles, to smaller, highly dynamic and interactive sites. This includes public sites, internal sites like intranets, or mixed environments like collaboration platforms.

Developers often choose Liferay for one of these cases and quickly find that Liferay is a great fit for completely different projects.

1.3 Creating Your Own Applications and Extending the Existing Ones

Liferay DXP is based on the Java platform and can be extended by adding new applications, customizing existing applications, modifying its behavior, or creating new themes. You can do this with any programming language supported by the JVM, such as Java itself, Scala, jRuby, Jython, Groovy, and others. Liferay DXP is lightweight, can be deployed to a variety of Java EE containers and app servers, and it supports a variety of databases. Because of its ability to be customized, you can add support for more app servers or databases without modifying its source code: just develop and deploy a module with the features you need.

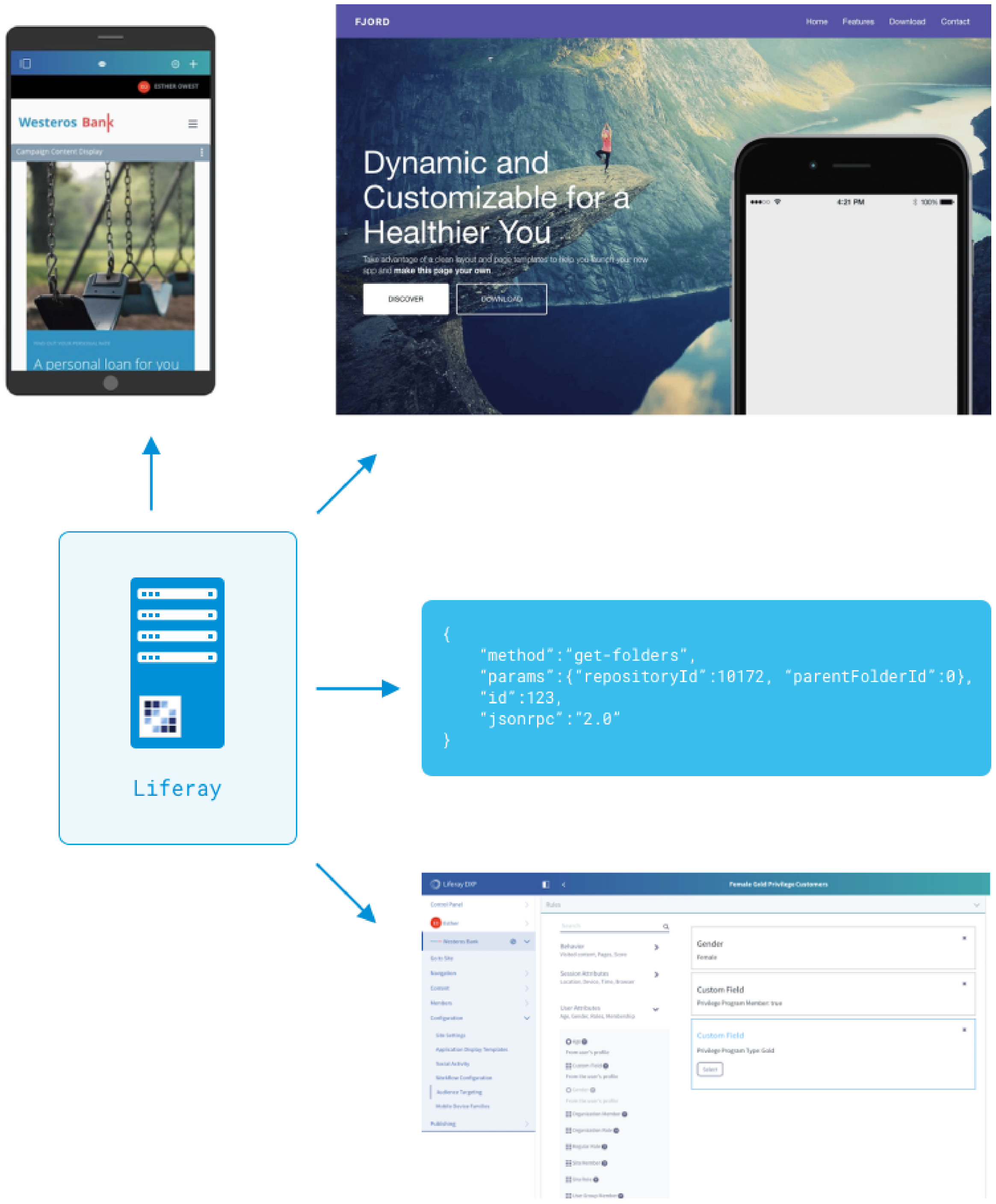
Speaking of code and deploying, here are some of the most common ways of expanding or customizing Liferay DXP's features:

1. Developing a new full-blown web application. The most common way to develop web applications for Liferay DXP is with portlets, because they integrate well with other existing applications. You are not, however, limited to portlets if you don't need to integrate your apps with others.
2. Customizing an existing web application or feature. Liferay DXP is designed to be extended. Many extension points can be leveraged to modify existing behavior, and most of these can be developed through a single Java class with some annotations (more details later).
3. Creating a new web service for an external system, a mobile app, an IoT device, or anything else.
4. Developing a mobile app that leverages Liferay as its back-end, which you can write in a fraction of the normal time thanks to Liferay Screens and Liferay Mobile SDK.
5. Developing a custom theme that adapts the look and feel of the platform to the visual needs of your project.

The Liferay platform can be used as a headless platform to develop web or mobile apps with any technology of your choice (Angular, React, Backbone, Cocoa, Android's Material Design components, Apache Cordova, etc). It can also be used as a web integration layer, leveraging technologies such as portlets to allow several applications to coexist on the same web page.

1.4 Fundamentals

What are the fundamentals that every Liferay developer should know?


Westeros Bank
Campaign Content Display
A personal loan for you
FJORD
Home
Features
Download
Contact
Dynamic and Customizable for a Healthier You
Take advantage of a clean layout and page templates to help you launch your new app and make this page your own
DISCOVER
DOWNLOAD
Liferay
Liferay DXP
Female Gold Privilege Customers
Gender
Female
Custom Field
Privilege Program Member: true
Custom Field
Privilege Program Type Gold

```
{
    "method":"get-folders",
    "params":{"repositoryId":10172, "parentFolderId":0},
    "id":123,
    "jsonrpc":"2.0"
}
```

Figure 1.3: Liferay DXP can be used by developers in many ways.

1. It's Open Source and puts a strong emphasis on following standards, instead of reinventing the wheel.
2. It's based on JavaEE and heavily leverages OSGi and several other popular technologies for the Java Platform.
3. It is based on a modular architecture and facilitates following a modular development paradigm for your own projects.
4. You can build your own web applications, portlets, or mobile apps on top of it.
5. It provides mature development tools, while staying agnostic so each developer can use his or her preferred tools.
6. It's all about reusing, providing reusable frameworks and libraries and allowing you to create your own.

Interested? More details below.

Open Source and based on Standards

Liferay DXP is both open source and built in the open, following a collaborative development model. That means that you can follow new development as it's happening, make comments on it, and contribute! Here are some tools that you can use to do all this:

1. Our ticketing system. All the changes made to the product, including all bug fixes, improvements, and new features start with a ticket created in JIRA. We have several projects there, but the main one for tracking the work of Liferay DXP or for reporting bugs you have found (with as many details as you can and steps to reproduce, of course) is LPS

2. GitHub: The home of our source code. You can use it to see the code changes as they happen and also to send pull requests for improvements. There are also many repos, but the main one is liferay-portal

3. Forums: It's where our community gets together to share ideas, discuss, and collaborate. Go ahead and ask your questions and help others ask theirs.

4. Blogs: Read the latest news, advice, and best practices from key core developers and our most active community members.

5. Participate: Learn how to get started participating. You will find options for all levels of expertise and time available.

In addition to being Open Source, Liferay is also heavily based on standards. This is great news for your project, since it significantly reduces the lock-in on Liferay. That also encourages us to improve constantly.

Here are some key standards supported by Liferay DXP:

- Portlets 1.0 (JSR-168) and Portlets 2.0 (JSR-286): Liferay DXP can run any portlets that follow these two versions of the specification. Liferay is also heavily involved in the upcoming Portlets 3.0 specification.

- JSF (JSR-127, JSR-314, JSR-344): The Java standard for building component based web applications. Liferay is an active contributor to the standard and lead of the JSF-Portlet Bridge specification.

- EcmaScript 2015: The latest incarnation of the JavaScript standard. Liferay's tooling provides the ability to use it in all modern browsers thanks to the integration of Babel JS.

- Content Management Interoperability Services (CMIS): Liferay's Documents and Media can behave as an interface for any external Documents Repository that supports this widely adopted standard.

- Java Content Repository (JSR-170): Files stored in the internal repository of Liferay's Documents and Media can be configured to be stored in a JSR-170 compatible repository if desired.

- WebDAV: Any Documents & Media folder can be mounted anywhere WebDAV is supported, such as Windows explorer or WebDAV-specific clients.

- SAML and OAuth 1.1: These are the most widely adopted security protocols for SSO and application sign in, supported through specific Apps that can be installed from Liferay's Marketplace.

- JAX-RS and JAX-WS: Incorporated since Liferay 7 as the preferred tooling to create web services.

- WSRP 1 and 2: Allows execution of portlets running in a remote container.

- OSGi r6: Liferay supports a wide range of the OSGi family of standards through its own implementations and also integrates the high quality implementations of the Apache Felix and Eclipse Equinox projects (which we also collaborate). Here are some of the most relevant supported standards:

  - OSGi runtime: Allowing any OSGi module to run in Liferay DXP
  - Declarative Services: Supports a dynamic component model for Liferay development.
  - Configuration Admin: Lets you create highly configurable applications that can be reconfigured on the fly. Liferay provides an auto-generated UI to change the configuration of any component that leverages this standard.

Technologies

Like any open source application, Liferay is built on the shoulders of giants. When we choose the technology on which to build our platform, it must have the following characteristics:

- It must balance being modern and being mature enough for demanding and critical enterprise environments.
- It should be widely adopted and have a mature community.
- It should be as easy as possible to contribute back, since we love to contribute to the open source projects we use.
- It should be possible to use only the piece of the project we need if we don't need the whole thing. That way, it's easier to replace that piece in the future if we find something that works better.

The goal, of course, is to give our developers and users the most up to date, easy-to-use, and stable platform to build your services on.

At its base, Liferay is a JavaEE application that also includes an OSGi container. This offers the best of both worlds: access to the world's most robust and fully featured enterprise platform, along with the benefits of the world's most fully featured and stable modular container. Now developers can develop and deploy enterprise-ready, scalable web and mobile-based applications in a dynamic, component-based environment.

With JavaEE and OSGi at the bottom of the stack, we build the rest of our core on well known or widely used products:

- Spring for transactions (and Dependency Injection in the core)
- Hibernate for database access (along with direct JDBC access for optimized queries)
- Elasticsearch for indexing and searching
- Ehcache for caching.

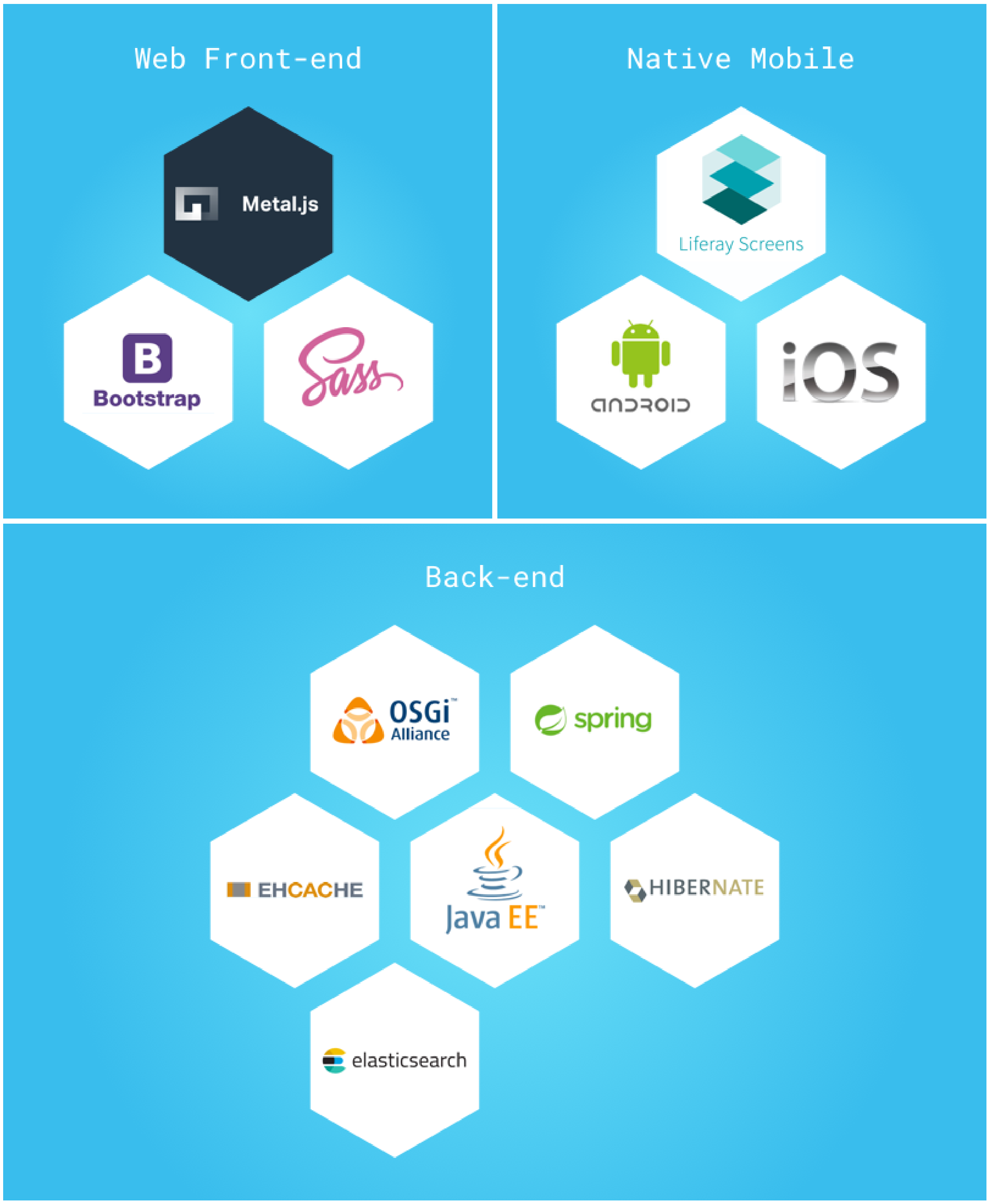


Figure 1.4: Liferay is based on popular, well known, and well supported technologies.

In the application layer, developers have access to many of the libraries they're familiar with and have been using for years:

- Xalan
- Xerces
- Apache Commons
- Tika
- dom4j

If you're approaching Liferay DXP with the intention of customizing it, you can know that most if not all of the tools you're familiar with are there. If you're writing applications on Liferay's platform, the sky's the limit: you can use any web framework you like, and you can write both servlet and portlet-based applications. If you're looking for a recommendation, though, we're happy to point you to either our MVCPortlet or our JSF-based LiferayFaces frameworks.

On the front-end, Liferay has kept pace with the most recent progressions in that space. If you've used Liferay in the past, you can of course continue to use Liferay's venerable Alloy UI, but you are also free to use the front-end technologies you love the most:

- Bootstrap
- SaSS
- EcmaScript 2015 (using Babel.js)

You can also use any JavaScript library, including

- Metal.js (developed by Liferay)
- jQuery (included)
- Lodash (included)
- Angular 1 or 2
- React
- Your library of choice

Liferay DXP follows a design language created by our designers at Liferay called Lexicon Experience Language, which has been implemented for use of the web as Lexicon.

Lexicon is automatically made available to application developers through a set of CSS classes and markup, although it's even easier to use our tag library.

For templating, JavaEE's JSP is there as expected as well as FreeMarker, but the modularity of the platform allows you to use Google's Soy (aka Closure Templates) or whatever else you like.

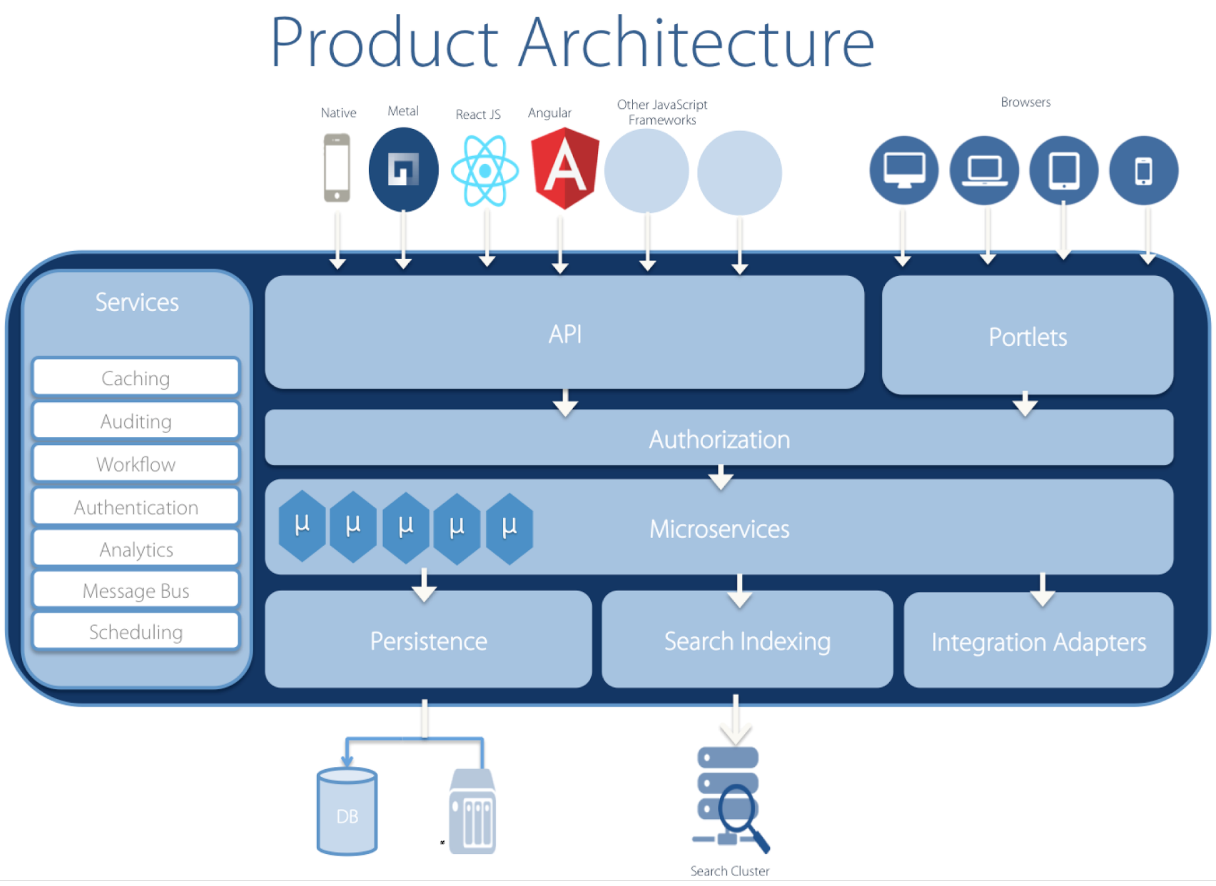
Liferay has also chosen build tools that give you freedom to use any development environment. Gradle along with bnd powers the product's build, but project layouts are dynamic, which means you can use anything from Maven to Ant/Ivy to build applications for Liferay.

In short, Liferay has done a lot to make sure its users and developers have access to the most widely used, robust tools possible–as well as the freedom to use the tools they like the most. Know that Liferay has your back and will do everything we can to provide you with the most flexible technology platform possible, so that you have the freedom to go and build great things on it–things we never could have expected or imagined.

Architecture

Liferay's design goals have from the beginning been to give you all the tools to create exactly the web presence you have in mind. To achieve this, the product must do these things:

- Provide a usable default configuration and interface
- Ship with best-of-breed apps that can be used to build sites quickly
- Make the UI customizable at any level of detail from small tweaks to a complete replacement
- Make the apps customizable at any level of detail
- Provide a robust development platform upon which new best-of-breed apps can be built and shared



These goals are now achieved to the furthest extent ever in Liferay's history, and it's all because of our new modular architecture.

Imagine an environment where every piece of functionality is an independent module. The modules declare three important things:

- The functionality they implement or define
- Their dependency on other modules
- Their priority relative to their functionality

Using this information, the container can start all the modules that fulfill their definitions, implementations, dependencies, and priorities.

Anything a developer wants to do is implemented as one or more modules. If it's a new application, that application can depend on existing modules and define a dependency on them. This enables you to use functionality that's already there without rewriting it yourself for your app. If it's a customization, in many cases it's just a simple matter of defining your customization with a higher priority than the existing functionality.

This is the power of a modular architecture.

Modules

All new applications, extensions, and customizations built on Liferay are built in a modular way. A module is the single unit of distribution and deployment in a modular architecture.

In the spirit of following existing standards, Liferay has leveraged a set of very powerful standards known as OSGi. OSGi defines, among other things, how modules can depend on each other and communicate. It also defines the packaging format for modules: OSGi bundles. An OSGi module is just a typical JAR file, familiar to Java developers as a ZIP file containing compiled code, templates, resources, and some meta information.

Services

One aspect of modern software architecture is the notion of services. These are independently running pieces of code that provide specific functionality when called. They operate just like services in the real world do. For example, you might call a service to come mow your lawn. You know how to call the service and to give it what it needs (money) in order to receive the service (a mown lawn). Software-based services work the same way.

Liferay's services are standard services as defined by the OSGi Alliance. Writing anything, whether it be an application, an interface to a database, or even a "service" as you define it, is easy to implement as an OSGi service, because they're both incredibly powerful and easy to develop. If you understand Java interfaces and how they are implemented–which is introductory Java material–you already understand more than 90% of what you need to know. First, you define the interface, or contract for the service: what it returns, and what it needs to return what it returns. Next, you define an implementation class that implements the contract.

In the services model, a class requests the service that provides the functionality it needs. This functionality is provided (often injected) with the right implementation automatically. It's similar to Spring or EJBs with one important addition: implementations can be changed at runtime, without restarting the system. This is achieved because when a service is deployed, it becomes part of a service registry maintained by Liferay's OSGi container. The container dynamically manages the lifecycle of the service and can start and stop services when appropriate.

The real power of services shines when they are extended. You can replace existing implementations or in advanced use cases have several implementations of a service. The developer can then choose to invoke all implementations or just the one with the highest priority (specified with what is called the service ranking). This means that if Liferay has a service that does something, you can customize or override that service by implementing its interface yourself and then deploying it with a higher ranking than the original service. The container then instantiates your implementation when the service is called by existing code. This simple, clean method is how most customizations are made to Liferay 7.

Components

In OSGi, possibly the best and certainly the easiest way to create services is through Declarative Services. In Declarative Services (aka DS), you create Components. A Component is a Java class (marked with an `@Component` annotation) that provides an implementation of a Service (as described above) and whose instantiation is handled automatically by DS. This is similar to what you might be used to if you have used Spring Beans or EJBs. DS also provides dependency injection using annotations (`@Reference`). This is convenient because the "wiring" of components is done by the container but can be changed while the server is running (unlike Spring).

Modules may contain as many service declarations and as many components as desired (or zero, of course).

In software engineering terms, a component is the smallest building block of a larger application, and that application is itself made up of many small components. This makes it easier to develop an application because you only have to deal with small, well-defined, bite-sized chunks of code at a time.

Real Life Benefits of Modular Development

The next question then becomes, so what? Why is this a big deal? Why should I have components, and what do I need them for?

It helps to examine two common development scenarios: a customization task and a full-blown application. Picture this: you have a system that generates a report in PDF format from data in a database. The data is captured from a web application running in Liferay. You come in to work in the morning and something's happened (it doesn't matter what it is; it could be corrupt data, the company has been bought, or a national emergency). You need to change that report as fast as possible, either to insert a new title page, add a warning to the existing title page, or whatever.

In the monolithic model you'd have to modify the application to change the report and then you'd have to redeploy the complete application. If this was a temporary change, to restore the application to its original state you'd again have to modify the application and redeploy it.

With a modular and component-based application, you'd fix a simple, small component–probably one Java class–that provides the functionality you need. You'd then deploy its module to the server. If you need to roll back that change in the future, you'd just do the same thing in reverse. In each case, you're only changing and redeploying the small piece of functionality that needs to change, not the whole application. At no time would you ever have to redeploy the whole application or take the server down.

For a full-blown application, the benefits are even greater. Modular development helps developers be more efficient in three important ways:

- An application made up of components can be written in parallel by multiple developers working on different components.
- An existing application can be extended by writing new components to implement features in different ways.
- Components can be enabled and disabled, allowing administrators to choose which features to enable in production.

For example, Liferay's Documents and Media library is a file repository that supports many back-ends. Each back-end is a component that can be maintained by different developers. They can be added and removed on the fly while the server is running.

Similarly, the services provided by the application are independent of the front-end technology. In fact, there can be multiple front-ends, from the web-based front-end Liferay provides out of the box, to a new front-end you might develop for either the web or mobile.

As you can see, many components running inside Liferay's OSGi container form something of an ecosystem of complementary services. Much of Liferay's functionality is in components, and when you deploy your code, it sits in the same ecosystem as Liferay's, with the same extension points. You can write components to provide new services or to override existing services with your own implementation, and the container manages it all. Liferay is an exciting platform that empowers developers to be more productive.

1.5 Liferay as a Development Platform

If you've been reading everything up to this point, you've heard all about Liferay DXP's architecture, modularity, and technologies. What's left is to tell you what it's like to use Liferay's platform as a basis for your site by customizing it or by developing applications on it. The platform is designed to make this easy and pleasant, and to integrate with the tools developers use every day.

But you're likely not interested in a bunch of prolegomena about it. Read on to learn the details.

Web Applications and Portlets

Liferay as a development platform has always provided flexibility for both administrators and developers by making it easy to have more than one application on a single page. Applications written this way are called *portlets*, and are a mainstay of Liferay's platform. You can use Liferay's MVC Portlet framework or common frameworks such as Spring MVC or JSF to write portlets. If you plan to have a web-based interface to your application, and want its administrator to have a lot of flexibility configuring it, portlets provide a very powerful model. In this model you can create several portlets instead of a larger application and let the administrator choose how to combine them with other pre-existing portlets into a larger interface.

That's not to say you don't have other choices. Since Liferay decouples its business logic from its UI (which is provided in separate modules), you have freedom to implement the UI in any other technology.

Because of this, you can use Liferay as a headless platform, because it's easy to create web services based on Service Builder, JAX-RS, and JAX-WS. Then you can build standalone web applications using any front-end technology or mobile technology you like.

Extensibility

As you might imagine, the system described above contains all the tools necessary to make a well designed system that allows developers not only to create applications based on modules, but also to extend the existing functionality of the system. Liferay can benefit from this now because the platform on which it rests is designed for both application development and customization.

Components make developing extensions and customizations convenient. If you compare this model to other products that aren't designed for customization, you'll see just how convenient it can be.

To customize an existing service, the only thing you need to do is deploy a component that extends the existing implementation. If you want to remove your implementation and revert back to the default behavior, you simply un-deploy your component.

Compare that with the traditional way of customizing software by downloading its source and maintaining a set of patches against it. Each time the software is updated, you have to re-download the source, re-apply your patches, and recompile the software.

With Liferay, your custom code is kept in your own modules, which the container takes care of applying based on metadata you supply.

Developer Tools

As you learned above, Liferay's OSGi container gives you these benefits:

- The container can start and stop components.
- A component implements an OSGi service.
- A component may use or consume OSGi services.
- The framework manages the binding of the services a component consumes (just like Spring or EJBs, but dynamically).

If all of this sounds great to you (as it does to us), there's only one thing left: how do you get started developing components? We believe in providing an easy path for new developers while at the same time preserving flexibility for experienced developers with strong tooling preferences. To achieve that, Liferay provides some great tools, and if you're an experienced developer, these also integrate into what you likely already use. If you use any of the standard build tools like Gradle or Maven, any text editor or common Java IDEs like Eclipse, intelliJ, or NetBeans, or any testing framework like Spock or JUnit, you can use them with Liferay to develop components.

Liferay's tools add some important enhancements:

- Blade CLI speeds you up by creating Gradle-based Liferay projects from templates.
- Liferay Workspace is an opinionated SDK based on Gradle that uses Blade CLI to integrate your projects and your runtime into one convenient, distributable and sharable place.
- Liferay IDE is an Eclipse-based development environment that integrates all the convenience of Blade CLI and Liferay Workspace into a best-of-breed graphical environment with all the bells and whistles you'd expect.
- Liferay Developer Studio provides all that Liferay IDE provides, plus additional tools that enterprise developers need.
- Liferay Service Builder helps you create your back-end faster by generating all your database tables, local services, and web services from a single XML file.

You can choose to use or ignore Liferay's tools. The point is you have the freedom to do that, because Liferay provides an open development framework that's designed to meet you where you are. We hate proprietary lock-in as much as you do, so our tools are designed to complement the tools you're using already instead of replacing them.

Beyond build tools and IDEs are the frameworks you'll use to build applications. Liferay's development frameworks include a lot of functionality–comments, social relationships, user management, and lots more–to speed up development of your applications. They help you build applications out of well-tested, modern, scalable, skinnable building blocks. You wind up not only with a great, functional application, but also with one that took less time to develop, looks the way you want it to, and performs well. This doesn't mean you're limited only to what Liferay provides; again, you can use third-party frameworks if that's what you like to use.

To develop portlets, Liferay provides a convenient and easy-to-use framework called MVCPortlet to make writing portlets easy, but developers are free to use any other framework, such as Spring MVC, to create portlets. MVCPortlet uses components to handle requests, benefiting from all the characteristics described above (lifecycle, extensibility, ease of composition, etc.). If you don't have a strong opinion on which framework to use, we recommend that you try it out.

Liferay also includes a utility called Service Builder that makes it easy to create back-end database tables, an object-relational map in Java for accessing them, and a place to put your business logic. It can also generate JSON or SOAP web services, giving developers a full stack for storing and retrieving data using web or mobile clients. But that doesn't prevent you from using Java Persistence (JPA) and generating JAX-WS web services.

In addition to the tooling, Liferay also provides many reusable frameworks.

Frameworks and APIs

Liferay's development platform provides a great framework for application development and also offers APIs. Lots of them. Applications can be created by leveraging Liferay's many frameworks that encapsulate features that are commonly needed by today's applications. For example, a commenting system allows developers to attach comments to any asset that they define, whether they be assets they develop or assets that ship with the system. Assets are shared by the system and are used to represent many common elements, such as Users, Organizations, Sites, User Groups, blog entries, and even folders and files.

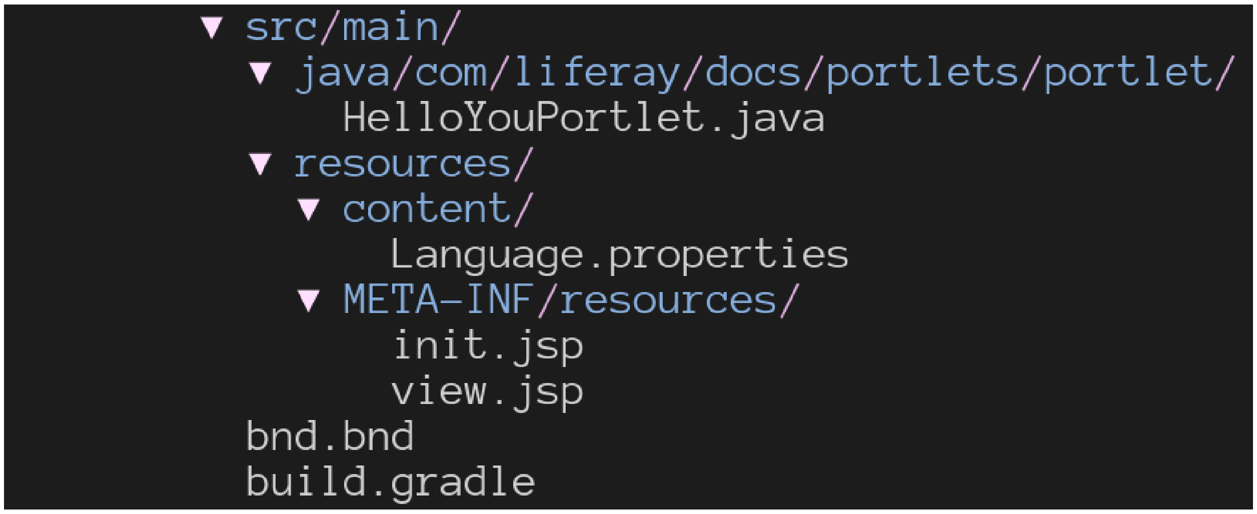
Liferay also includes many frameworks for operating on assets. A workflow system makes it easy to create applications that require an approval process for users to follow. The recycle bin stores deleted assets for a specified period of time, making it easy for users to restore data without the intervention of an administrator. A file storage API with multiple available back-ends makes storing and sharing files trivial. Search is built into the system as well, and it is designed for developers to integrate it with their applications. Many of the frameworks you might need when developing complex applications are already there; you just need to take advantage of them: a Social Networking API, user-generated forms with data lists, a message bus, an audit system, and much more.

Example Liferay Projects

Enough theory. It's time for practice. A good way to get the flavor of developing on Liferay's platform across is to show you some projects. First, you'll see a portlet developed with MVCPortlet, showcasing the use of components as well. Once you've seen that, the next best thing is to see an extension. Both of these examples serve to show you how easy it is to build functionality following a modular paradigm.

It would be nice to show you the standard Hello World project, Liferay style, but that would be too easy: the default template that Blade or Liferay IDE creates already does that by default. Instead, you'll see the Hello *You* portlet. This does the same thing as Hello World, except it adds the first name of the user to the message. If your name therefore is John, it'll return Hello, John.

Here's what the project layout looks like:



```
▼ src/main/
  ▼ java/com/liferay/docs/portlets/portlet/
      HelloYouPortlet.java
  ▼ resources/
    ▼ content/
        Language.properties
    ▼ META-INF/resources/
        init.jsp
        view.jsp
bnd.bnd
build.gradle
```

Figure 1.5: The Hello You portlet has a simple project structure.

No new files were created after this project was generated by Liferay's Blade CLI tool, so this is as simple as it gets. You have your portlet class, which is in the .java file. You also have two different kinds of resources: language properties and JSP files. Finally, the `bnd.bnd` file describes the application's metadata for the OSGi container, and the `build.gradle` file builds the project.

Any web developer that's familiar with Java can understand the JSPs, but some explanation is in order because of the style. Liferay's coding style defines a single `init.jsp` that contains all the imports and tag library initializations necessary for the front-end. This way, any JSP can simply include `init.jsp`, and all of its imports are satisfied. The `init.jsp` for this project was not modified from the generated project, and it looks like this:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib uri="http://java.sun.com/portlet\_2\_0" prefix="portlet" %>

<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>

<%@ taglib uri="http://liferay.com/tld/portlet" prefix="liferay-portlet" %>

<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>

<%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>

<liferay-theme:defineObjects />

<portlet:defineObjects />
```

As you can see, all it does is declare the tag libraries you probably want to use, and then it calls a couple of tags that makes objects from the portlet framework available. Since there's nothing really interesting here, you'll want to look at `view.jsp` next:

```
<%@ include file="/init.jsp" %>

<jsp:useBean id="userName" type="java.lang.String" scope="request" />

<p>

    <b>Hello, <%=userName %>!</b>

</p>
```

Now we've got something. The portlet class (the Controller, in MVC terms) has made a `userName` string available in the request, and this JSP retrieves it and uses it to say hello to the user. The real functionality, therefore is in the portlet class:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=hello-you Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class HelloYouPortlet extends MVCPortlet {

    @Override
    public void render(RenderRequest renderRequest,
                RenderResponse renderResponse)
            throws IOException, PortletException
    {
        ThemeDisplay themeDisplay = (ThemeDisplay)
                renderRequest.getAttribute(WebKeys.THEME\_DISPLAY);

        User user = themeDisplay.getUser();

        renderRequest.setAttribute("userName",
                user.getFirstName());



        super.render(renderRequest, renderResponse);
    }

}
```

Now we're talking; here's the real stuff. At the top is the `@Component` annotation, which tells the OSGi container how it should treat this module. By specifying `immediate=true`, you're saying that when this module is deployed and all of its dependencies are satisfied, it should be started immediately instead of being lazy-loaded. Next are several properties specific to portlets: the category in which it should appear in Liferay's UI, its display name, its default view, and more. Finally, the service–which is just a Java Interface–that it implements is defined, which is the portlet class.

Next, you have the class itself, which extends Liferay's `MVCPortlet` class (that extends `GenericPortlet`, that implements `Portlet`). The only method overridden is the `render()` method, and Liferay's API is used to get the user's first name and put it in a request attribute called `userName`.

So you can see how this works: the portlet runs and retrieves the user's first name, puts that in the request, and then by the use of the template path and view template properties specified in the `@Component` annotation, forwards processing to `view.jsp`, where the user's first name is retrieved and displayed.

The only other item of interest is the `bnd.bnd` file:

```
Bundle-SymbolicName: com.liferay.docs.hello.you
Bundle-Version: 1.0.0
```

This declares the name of the module (sometimes also called a bundle). It's a good practice to namespace it properly to avoid name conflicts in the container. The version is also declared, which allows the container to manage dependencies down to the version level of a module. This is called Semantic Versioning, and is a discussion by itself.

That's all there is to this portlet. Next, you'll see an extension, which in many cases is even simpler than a portlet.

Liferay's UI is divided up into several areas. There's the control menu and the product menu, which contains the add menu and the simulation menu. If you want to extend the UI, you can do that by deploying a module that adds what you want. In this example, you'll add a link to the product menu, which is the menu that by default sits in the top right of the browser:

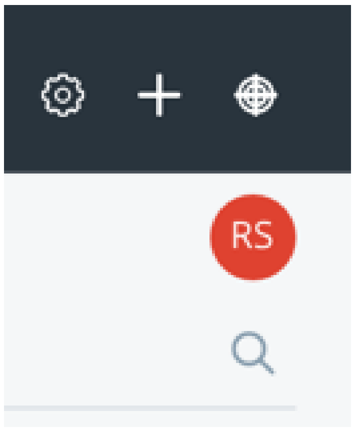


Figure 1.6: The product menu appears beneath the user's profile link.

To this, you'll add a link to this website:

As with the portlet project, this project's layout contains only a few items that are easy to understand.

As before, you have a build script, a `bnd.bnd` file that declares the module's name and version, and this time, only a Java class and a language properties file.

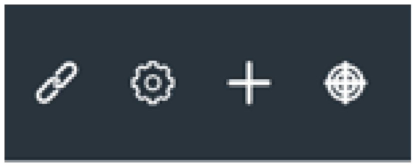
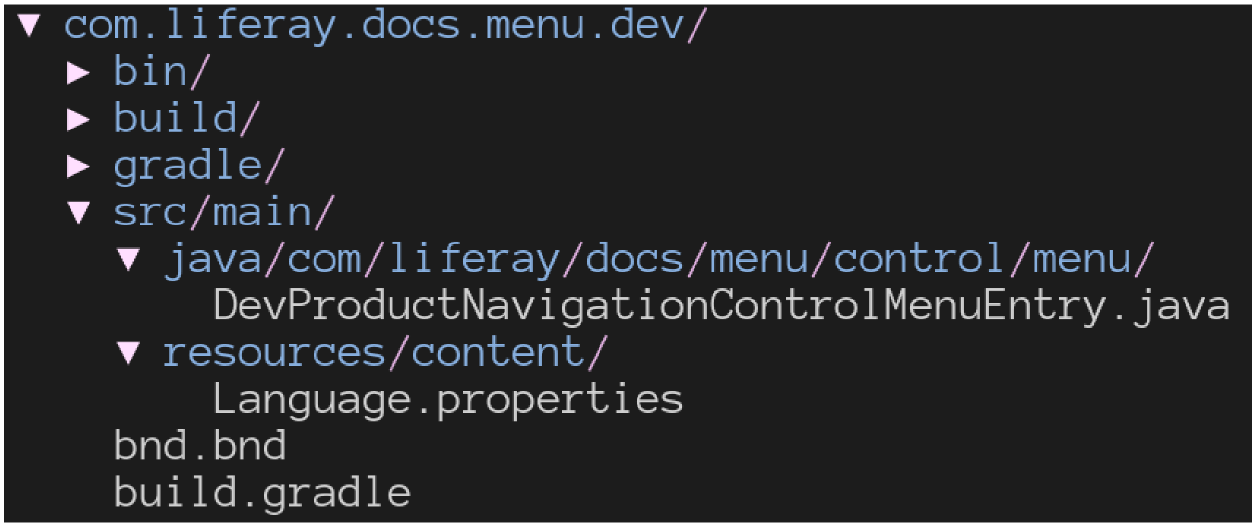


Figure 1.7: You can add links to the product menu by deploying a component.



```
▼ com.liferay.docs.menu.dev/
  ▶ bin/
  ▶ build/
  ▶ gradle/
  ▼ src/main/
    ▼ java/com/liferay/docs/menu/control/menu/
        DevProductNavigationControlMenuEntry.java
    ▼ resources/content/
        Language.properties
    bnd.bnd
    build.gradle
```

Figure 1.8: The product menu project is simpler than the portlet was.

The Java class defines only four methods:

```
@Component(
    immediate = true,
    property = {
        "product.navigation.control.menu.category.key=" +
                ProductNavigationControlMenuCategoryKeys.USER,
        "product.navigation.control.menu.entry.order:Integer=1"
    },
    service = ProductNavigationControlMenuEntry.class
)
public class DevProductNavigationControlMenuEntry
    extends BaseProductNavigationControlMenuEntry
    implements ProductNavigationControlMenuEntry {

    @Override
    public String getIcon(HttpServletRequest request) {

        return "link";
    }

    @Override
    public String getLabel(Locale locale) {

        ResourceBundle resourceBundle = ResourceBundleUtil.getBundle(
            "content.Language", locale, getClass());

        return LanguageUtil.get(resourceBundle, "custom-message");
    }

    @Override
    public String getURL(HttpServletRequest request) {

        return "https://portal.liferay.dev";

    }

    @Override
    public boolean isShow(HttpServletRequest request) throws
        PortalException {

        return true;
    }

}
```

As before, this project was generated using a template from Blade CLI. The source code is part of the template; the only thing you'll need to do is provide the link.

The first method gets the Font Awesome icon you want to use in the menu. The next gets the "label," the text that appears when a user hovers the mouse over the link. This text is the value of the only property in the `Language.properties` file:

```
custom-message=Liferay Developer Network
```

The next method returns the URL that's the destination for this link, and the final method returns a boolean for showing or hiding the link.

When you deploy this module, the link is added to the menu when the module starts. You don't have to mess around looking in Liferay's JSP or JavaScript files to customize the menu: it's an extension point, and it is designed to be customized.

This is the modular paradigm for development. It helps you keep a clean separation of your code, whether it be applications or extensions, from the code that ships by default, and it gives you the power to customize the system dynamically, while it's running, to avoid downtime. It is a different way of doing things, but we believe it's a better way. When you start working with modules and see the benefits you can gain, we think you'll agree.

Now you're ready to explore some more about Liferay. We're not planning to leave you here, as though this were a dead site. Please feel free to use the suggestions link at the bottom of every article you'll encounter if you think something could be improved about that article. If you have feedback about the site itself, the feedback button is always floating at the bottom right. There are living, breathing people behind all the content on this site, and we stand ready to assist you on your Liferay journey.

Chapter 2

Introduction to Front-End Development

When approaching the development of your application's front-end, Liferay DXP offers a wide range of approaches, frameworks, utilities, and mechanisms to make your life easier.

2.1 JavaScript

If you've used Liferay in the past, you can of course continue to use Liferay's venerable Alloy UI, but you are also free to use the front-end technologies you love the most:

- ECMAScript 2015
- Metal.js (developed by Liferay)
- AlloyUI (developed by Liferay)
- jQuery (included)
- Lodash (included)

2.2 Lexicon

Liferay DXP follows a design language created by our designers at Liferay called Lexicon Experience Language, which has been implemented for use of the web as Lexicon.

Lexicon is automatically made available to application developers through a set of CSS classes and markup, although it's even easier to use our tag library.

2.3 Templates

For templating, Java EE's JSP is there as expected as well as FreeMarker, but the platform's modularity enables using Google's Soy (aka Closure Templates) or whatever else you like.

2.4 Themes

A Liferay Theme is the overall look and feel for a site. Themes are a combination of CSS, JavaScript, HTML, and FreeMarker templates. Although the default themes are nice, you may wish to create your own look and feel for your site.

In Liferay DXP, Liferay provides an easy-to-use tool called the Liferay Theme Generator that helps automate the theme development process.

Themes created with the Liferay Theme Generator give you access to theme gulp tasks that offer basic functions, such as `build` and `deploy`, along with more complex interactions, such as auto deploying when a change is made and setting the base theme.

2.5 Front-End Extensions

Liferay DXP's modularity has many benefits for the front-end developer, in the form of development customizations and extension points. These extensions assure the stability, conformity, and future evolution of your applications.

Below are some of the available front-end extensions:

- Theme Contributors
- Context Contributors
- Portlet Decorators
- Editor Config Contributors

CHAPTER 3

JAVASCRIPT IN LIFERAY DXP

Liferay DXP's front-end is extendable, flexible, and future ready.

Like previous versions, many components are written using AlloyUI. AlloyUI is based on YUI, and is no longer under active development. Because of this, we have included jQuery and also have developed a new framework called MetalJS.

3.1 MetalJS

Metal.js is a JavaScript library for building UI components in a solid and flexible way. Metal is built from the ground up with performance in mind and is flexible enough to be built as global objects, AMD modules, or jQuery plugins. Metal is cutting edge JavaScript, using ECMAScript 6 (ES6)/ ECMAScript 2015 (ES2015), which provides you with clean code that's easy-to-read.

For more information see the Metal.js docs.

3.2 ES2015

ECMAScript 6 (ES6)/ ECMAScript 2015 (ES2015) is enabled by default in your plugins, so you can write your own modules using the latest improvements to the language.

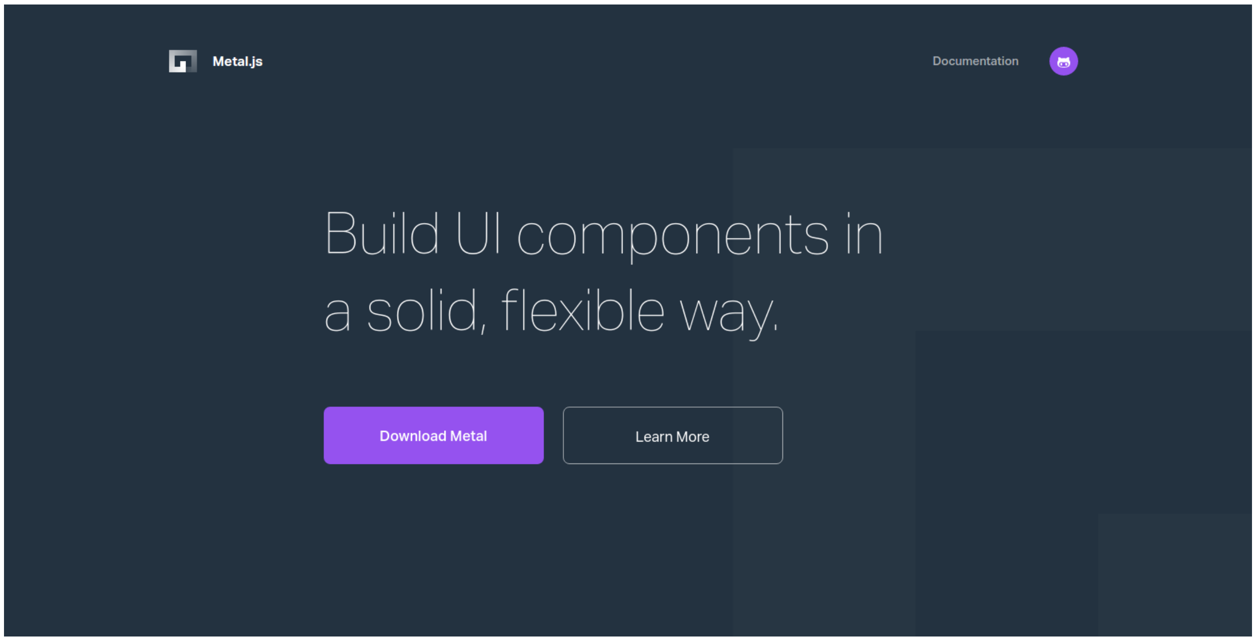
You can learn more about how to leverage ES6 in your modules in the Preparing Your JavaScript Files for ES2015 and Using ES2015 Modules in your Portlet tutorials.

3.3 AlloyUI

AlloyUI is an open source front-end framework built on top of Yahoo! User Interface Library (YUI). It leverages all of YUI's modules and adds even more cutting edge components and features to help you build terrific UIs. AlloyUI provides the following key benefits:

- Create modern UI components that provide a consistent look & feel across Liferay DXP.
- Server-agnostic, so you can use it with any technology.

As of 7.0, AlloyUI has been officially sunsetted. This means that we are no longer developing new features for it, but it is still included in the product and actively maintained.



Build UI components in a solid, flexible way.

Figure 3.1: Metal.js is a new framework for building UI components.

Simply Powerful

Robust UI tools at your fingertips.

AlloyUI is a framework built on top of YUI3 (JavaScript) that uses Bootstrap (HTML/CSS) to provide a simple API for building high scalable applications.

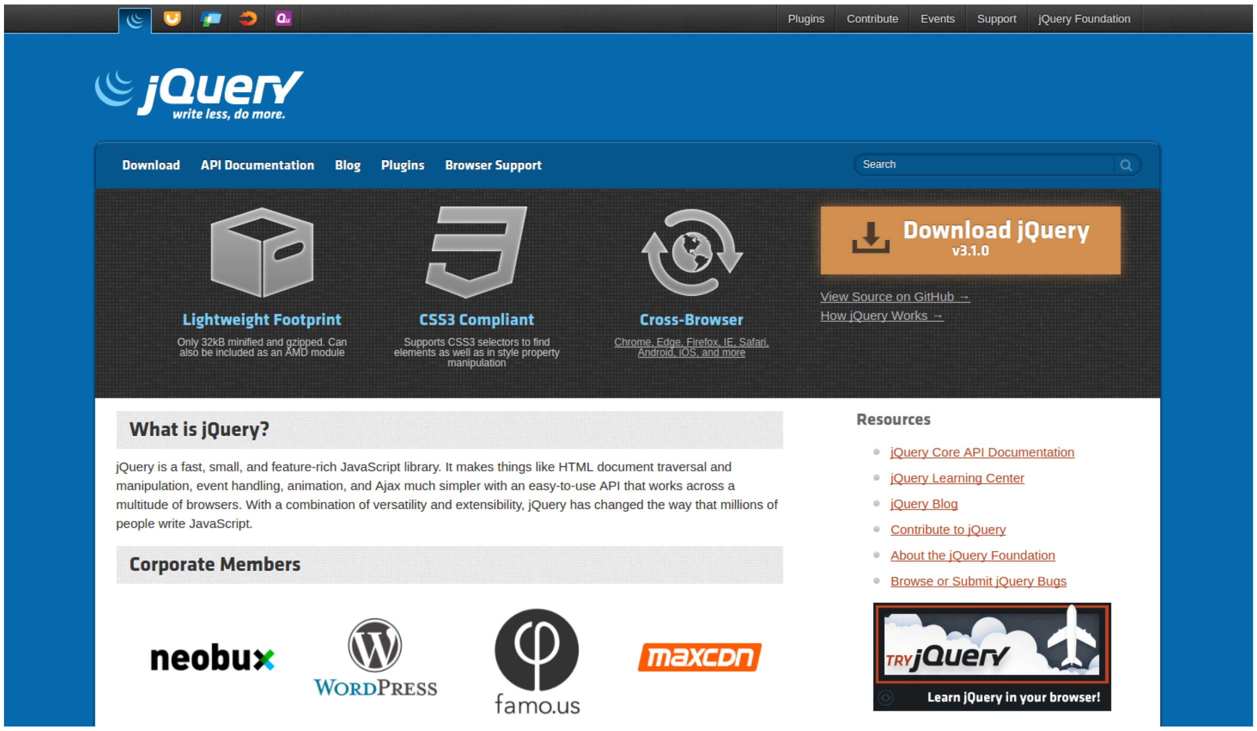
"When AlloyUI arrived on the scene, it added a new dimension to YUI contributorship. Professional, robust, and beautifully designed, the Alloy project and its engineers created a world of new options for YUI implementers, and it helped the YUI project turn the corner to become a contributor-driven, community-owned open source project."
— Eric Miraglia, Product Manager, Google Inc. & Former YUI Manager

Figure 3.2: AlloyUI is sunsetted as of 7.0.

3.4 jQuery

jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML. It is the most popular JavaScript library in use today. The syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.

While jQuery is great for small websites, once you start creating highly scalable applications like Java portals, you'll need a more robust solution. That being the case, we strongly recommend you to use one of our other provided solutions mentioned above.



Lightweight Footprint
Only 32kB minified and gzipped. Can also be included as an AMD module

CSS3 Compliant
Supports CSS3 selectors to find elements as well as in style property manipulation

Cross-Browser
Chrome, Edge, Firefox, IE, Safari, Android, iOS, and more

What is jQuery?

jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

Corporate Members

Resources

- jQuery Core API Documentation
- jQuery Learning Center
- jQuery Blog
- Contribute to jQuery
- About the jQuery Foundation
- Browse or Submit jQuery Bugs

Figure 3.3: jQuery is a fast, small, and feature-rich JavaScript library.

Chapter 4

Metal.js

Metal.js is a lightweight, easy-to-use JavaScript framework that lets you create UI Components with ease, thanks to its integration with templating languages.

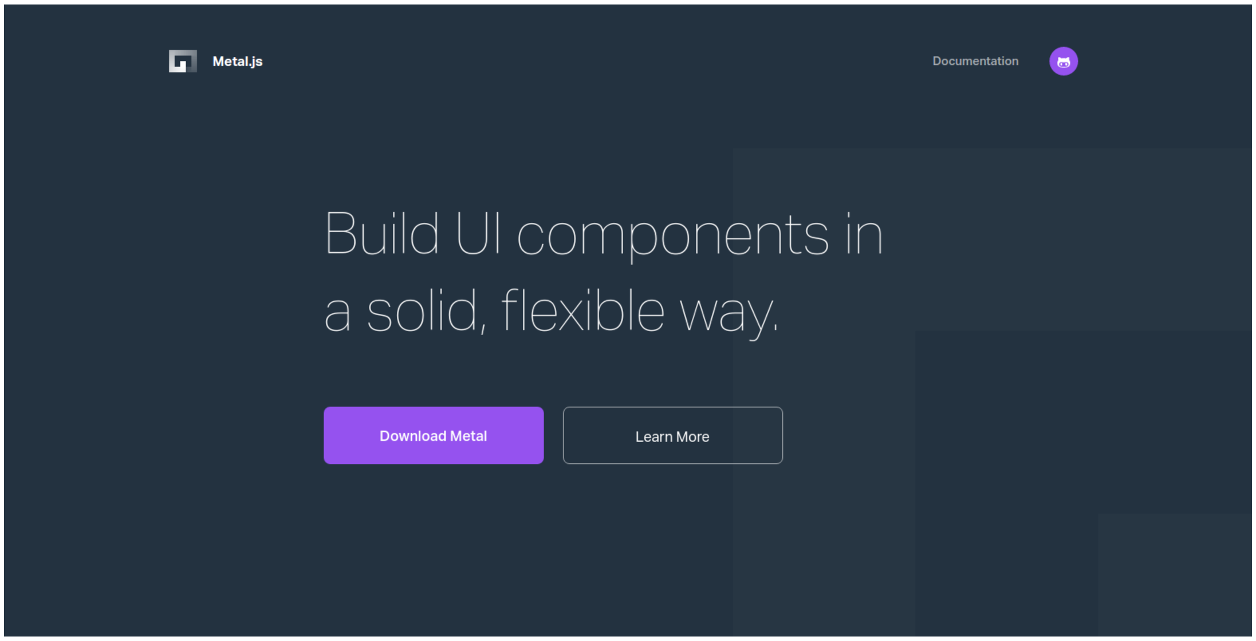


Figure 4.1: You can create UI's easily, thanks to Metal.js.

Metal.js is built with you in mind, offering flexibility with how your rendering logic is handled. You can use template languages to write your rendering logic or keep your rendering logic and business logic within the same file if you prefer.

By default, Metal.js offers integration points with Google closure templates and Facebook JSX templates. The rendering layer is completely customizable though, so you can add more rendering options if needed.

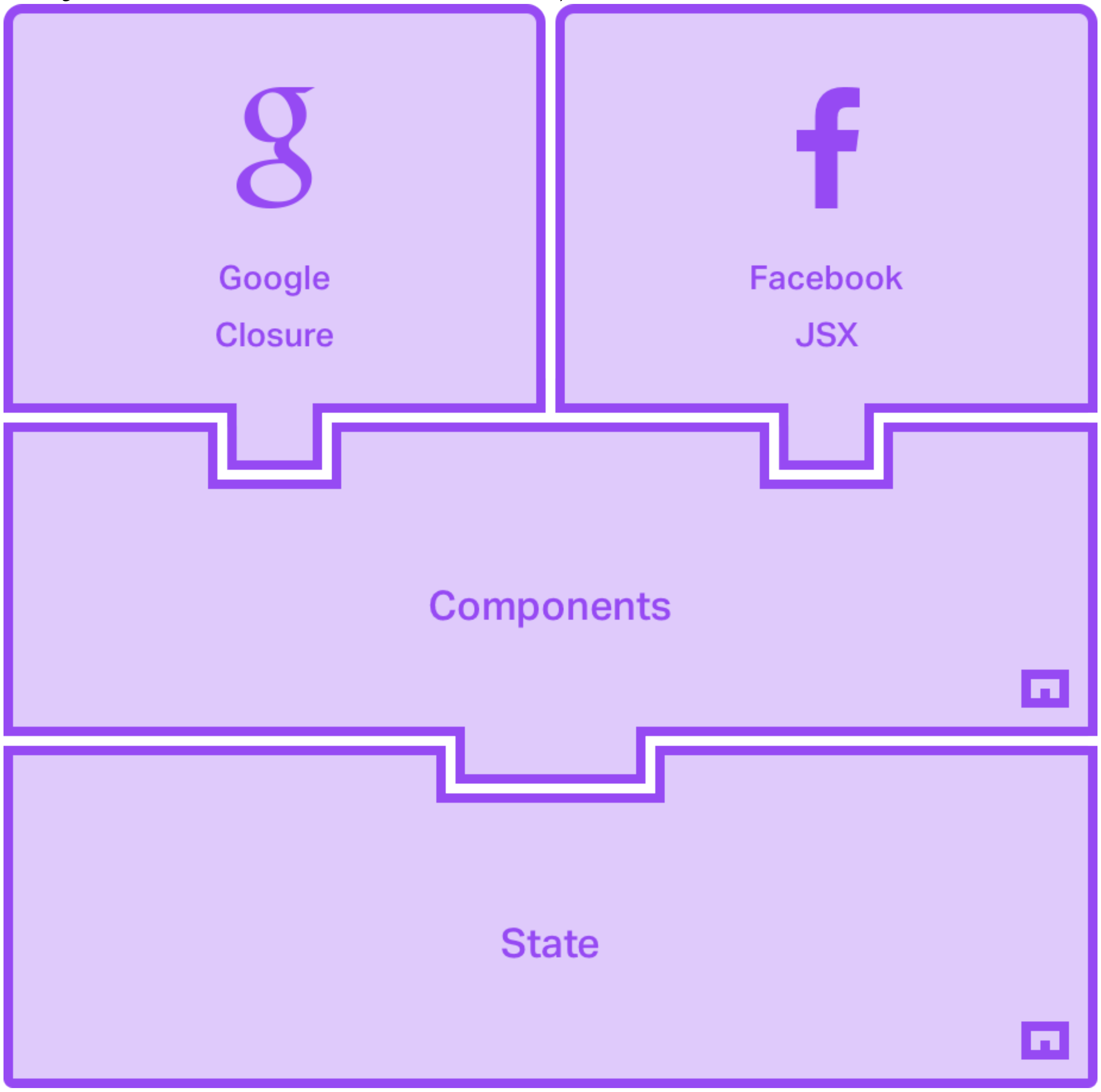
Below is an example of a closure(Soy) template written for Metal.js:

```
{template .render}
// ...
```

```
<button onClick="{$close}" type="button" class="close">
// ...
{/template}
```

Metal.js has two main classes: `State`, and `Component` which extends from `State`. The `Component` class adds additional rendering features for your Component. If your Component doesn't require rendering, you can just use `State`.

The figure below illustrates the architecture for Metal.js:



Metal.js takes full advantage of the ECMAScript 6 (AKA ECMAScript 2015) language, so you can use the latest features that the language has to offer. Below is a list of some of the great features that you get with ES6:

- Class syntax like other OO languages. Classes support prototype-based inheritance, super calls, instance and static methods and constructors.

- Arrow method syntax. `var odds = numbers.map(v => v + 1);`
- Language-level support for modules for Component definition. Codifies patterns from popular JavaScript module loaders like AMD (as shown below):

```
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

These are just a few of the ES6 features available that you can use in your Metal.js components.

4.1 Related Topics

Creating a Soy Portlet
JavaScript Module Loaders

Chapter 5

Starting Module Development

Developing modules for Liferay DXP requires:

- **Creating a folder structure**: A good folder structure facilitates evolving and maintaining code, especially in collaboration. Popular tools use pre-defined folder structures familiar to developers.
- **Writing code and configuration files**: A manifest, Java classes, and resources. Modules stubbed out with them let developers focus on implementing logic.
- **Compilation**: Acquiring dependencies and building the module. Common build tools that manage dependencies include Gradle, Maven, and Ant/Ivy.
- **Deployment**: Interacting with the runtime environment to install, monitor, and modify modules.

There are several good build tools for developing modules on Liferay DXP. This tutorial demonstrates starting a new module using Liferay Workspace. It's Liferay's opinionated build environment based on Gradle and BndTools that simplifies module development and automates much of it.

Note: Liferay supports developers using their build tool of choice. In addition to providing Liferay Workspace for those who don't already have a preferred build environment, Liferay provides good support for Maven and Gradle. The following tutorials and samples demonstrate developing in these environments.

- Maven tutorials and samples
- Gradle in Liferay @ide@ and samples

Note: Themes and Layout Templates are not built as modules. To learn how to create them, see the Themes and Layout Templates tutorials.

Here are the steps for starting module development:

1. Set up a Liferay Workspace
2. Create a module

3. Build and deploy the module

On completing this tutorial you'll have created a module and deployed it to a local Liferay DXP bundle.

5.1 Setting up a Liferay Workspace

Creating and configuring a Liferay Workspace (Workspace) is straightforward using a tool called Blade CLI (Blade). Blade is a command line tool that creates Workspaces and performs common tasks.

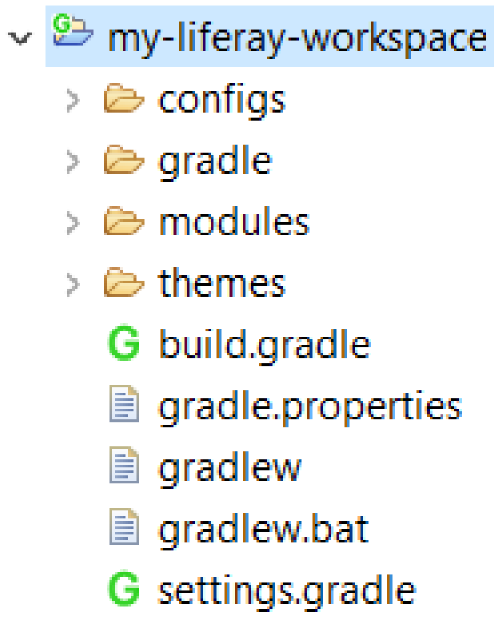
Follow the steps in this tutorial to install Blade if you don't already have it.

The `blade` executable is now in the system path.

You can create a Workspace in the current directory by executing this command:

```
blade init <workspaceName>
```

You've created a Workspace! Its directory structure looks like the one shown in the figure below.



```
my-liferay-workspace
    configs
    gradle
    modules
    themes
    build.gradle
    gradle.properties
    gradlew
    gradlew.bat
    settings.gradle
```

Figure 5.1: Liferay Workspace aggregates projects so they can leverage the Gradle build environment.

Workspace can be configured to use a Liferay DXP installation bundle anywhere on the local file system. The `liferay.workspace.home.dir` property in `gradle.properties` sets the default bundle location to a folder `<workspace>/bundles` (not yet created). For convenience it's suggested to install a Liferay DXP bundle there. If you install it to a different location, uncomment the `liferay.workspace.home.dir` property and set it to that location.

Note: User interfaces in Liferay @ide@ lets developers create and import Liferay Workspace projects. To create a project, follow the tutorial Creating a Liferay Workspace Project with Liferay @ide@. To import a project, use the wizard from *File* → *Import* → *Liferay* → *Liferay Workspace Project*.

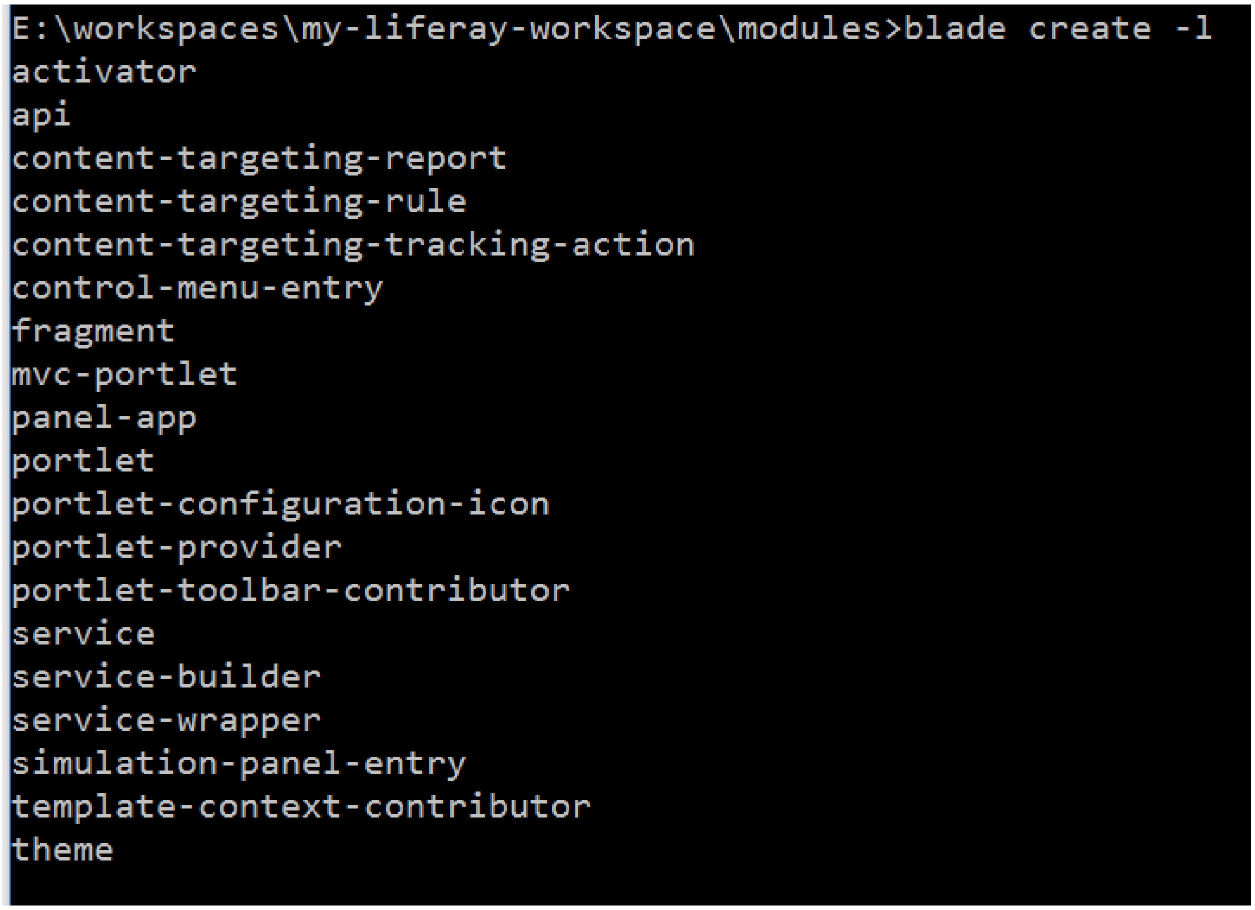
The Workspace is ready for creating modules.

5.2 Creating a Module

Blade provides module *templates* and module *samples*. The templates stub out files for different types of modules. The samples can be generated in a Workspace and demonstrate many module types. Developers can use templates and samples to develop modules.

Using Module Templates

The Blade command `blade create -l` lists the module templates.



```
E:\workspaces\my-liferay-workspace\modules>blade create -l
activator
api
content-targeting-report
content-targeting-rule
content-targeting-tracking-action
control-menu-entry
fragment
mvc-portlet
panel-app
portlet
portlet-configuration-icon
portlet-provider
portlet-toolbar-contributor
service
service-builder
service-wrapper
simulation-panel-entry
template-context-contributor
theme
```

Figure 5.2: Blade's `create` command generates a module based on a template. Executing `create -l` lists the template names.

Note: Liferay @ide@'s module wizard lets developers select a template for their module project. For details, see the tutorial Creating a Module with Liferay @ide@.

Here's the command syntax for creating a module:

```
blade create [options] moduleName
```

Module templates and their options are described here.

Here's an example of creating a Liferay MVC Portlet module:

```
blade create -t mvc-portlet -p com.liferay.docs.mymodule -c MyMvcPortlet my-module
```

Module projects are created in the `modules` folder by default.
Here's the module project anatomy:

- `src/main/java/` → Java package root
- `src/main/resources/content/` (optional) → Language resource bundle root
- `src/main/resources/META-INF/resources/` (optional) → Root for UI templates, such as JSPs
- `bnd.bnd` → Specifies essential OSGi module manifest headers
- `build.gradle` → Configures dependencies and more using Gradle

The figure below shows an MVC portlet module project.

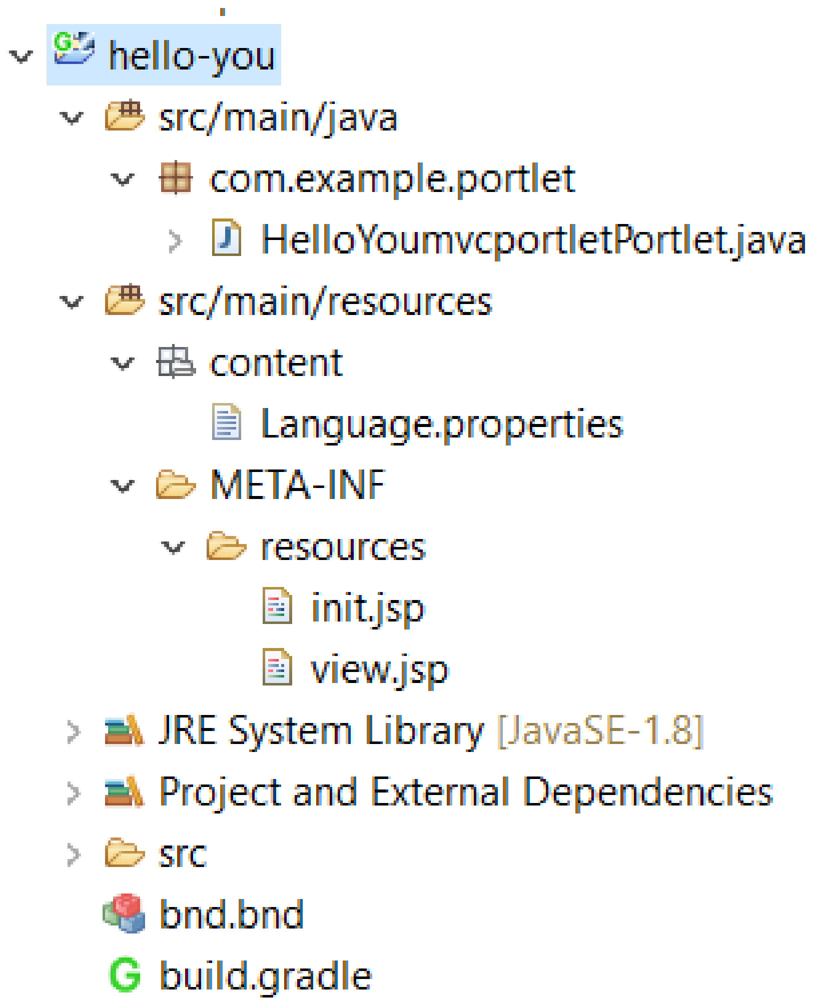


Figure 5.3: Liferay modules use the standard Maven directory structure.

Sample modules are another helpful development resource.

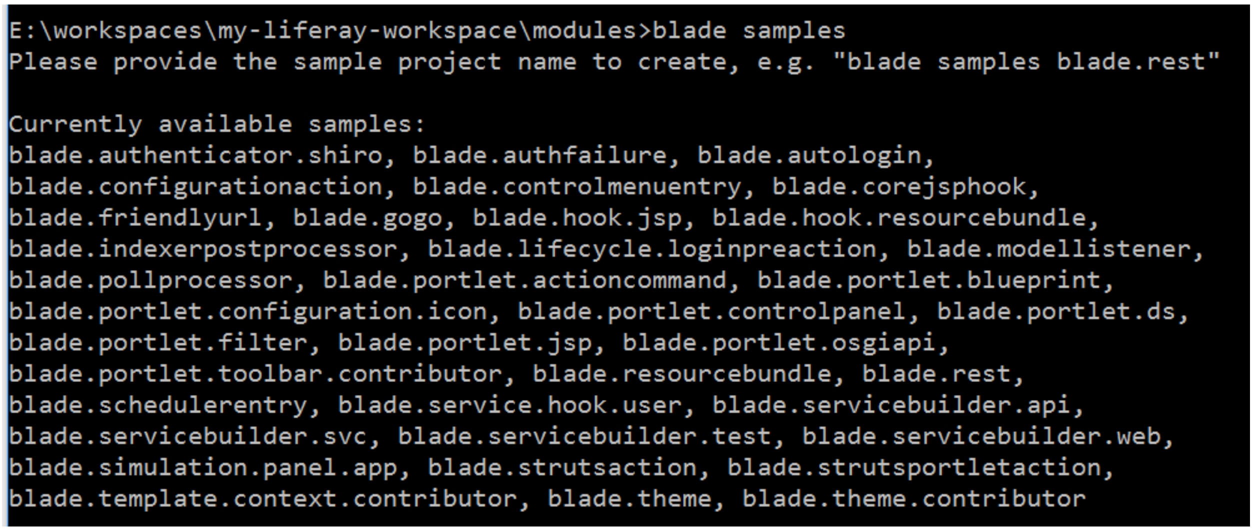
Using Module Samples

An alternative to creating a module from a template is to generate a *sample* module. Developers can examine or modify sample modules as desired.

This command lists the sample names:

```
blade samples
```

The figure below shows the listing.



```
E:\workspaces\my-liferay-workspace\modules>blade samples
Please provide the sample project name to create, e.g. "blade samples blade.rest"

Currently available samples:
blade.authenticator.shiro, blade.authfailure, blade.autologin,
blade.configurationaction, blade.controlmenuentry, blade.corejsphook,
blade.friendlyurl, blade.gogo, blade.hook.jsp, blade.hook.resourcebundle,
blade.indexerpostprocessor, blade.lifecycle.loginpreaction, blade.modellistener,
blade.pollprocessor, blade.portlet.actioncommand, blade.portlet.blueprint,
blade.portlet.configuration.icon, blade.portlet.controlpanel, blade.portlet.ds,
blade.portlet.filter, blade.portlet.jsp, blade.portlet.osgiapi,
blade.portlet.toolbar.contributor, blade.resourcebundle, blade.rest,
blade.schedulerentry, blade.service.hook.user, blade.servicebuilder.api,
blade.servicebuilder.svc, blade.servicebuilder.test, blade.servicebuilder.web,
blade.simulation.panel.app, blade.strutsaction, blade.strutsportletaction,
blade.template.context.contributor, blade.theme, blade.theme.contributor
```

Figure 5.4: The `blade samples` command lists the names of sample modules developers can create, examine, and modify to meet their needs.

Here's the Blade samples command syntax:

```
blade samples <sampleName>
```

It creates the sample project in a subfolder of the current folder.

Building a module and deploying it to Liferay DXP is easy.

5.3 Building and Deploying a Module

Liferay Workspace provides Gradle tasks for building and deploying modules. Blade's `blade gw` command solves a common need in Gradle projects: invoking the Gradle wrapper from any project directory. You can use `blade gw` just as you would invoke `gradlew`, without having to specify the wrapper path.

Note: For an even simpler Gradle wrapper command, install *gw*.
`(sudo) jpm install gw@1.0.1`
Usage: `gw <task>`

In a module folder, execute this command to list the Gradle tasks available:

```
blade gw tasks
```

Workspace uses BndTools to generate the module's OSGi `MANIFEST.MF` file and package it in the module JAR. To compile the module and generate the module JAR, execute the `jar` Gradle task:

```
blade gw jar
```

The generated JAR is in the module project's `build/libs` folder and ready for deployment to Liferay DXP.

Start your Liferay DXP server, if you haven't already started it.

Tip: To open a new terminal window and the Workspace's Liferay DXP server (bundled with Tomcat or JBoss/Wildfly), execute this command:

`blade server start -b`

Blade can deploy modules to any local Liferay DXP server. It communicates with Liferay DXP's OSGi framework using Felix Gogo shell and deploys modules directly to the OSGi container using Felix File Install commands. The command uses the default port 11311.

To deploy the module, execute this command:

```
blade deploy
```

Also Blade lets developers deploy all modules in the current folder tree. To deploy all modules in a Workspace's modules folder, for example, execute `blade deploy` in the `<workspace>/modules` folder.

If you're using Liferay @ide@, you can deploy modules by dragging them from the Package Explorer onto the Liferay DXP server. @ide@ provides access to Liferay Workspace Gradle tasks too.

Note: When deploying a module to Liferay DXP using Blade CLI, the module is directly installed into Liferay DXP's OSGi container. This means that the module is stored differently in Liferay DXP than if it were copied into the `LIFERAY_HOME/deploy` folder. See the Deploying Modules with Blade CLI tutorial for more information.

Once you've deployed a portlet module, it's available in the Liferay DXP UI under the application category and name you specified via the portlet component's `com.liferay.portlet.display-category` and `javax.portlet.display-name` properties in the `@Component` annotation.

5.4 Redeploying Module Changes Automatically

Blade lets developers set a *watch* on changes to a module project's output files. If they're modified, Blade redeploys the module automatically. To set a watch on a module at deployment, execute this command in the module project:

```
blade deploy -w
```

Here's output from deploying (and watching) a module named *com.liferay.docs.mymodule*:

```
E:\workspaces\my-liferay-workspace\modules\my-module-project>blade deploy -w

:modules:my-module-project:compileJava UP-TO-DATE
:modules:my-module-project:buildCSS UP-TO-DATE
:modules:my-module-project:processResources UP-TO-DATE
:modules:my-module-project:transpileJS SKIPPED
:modules:my-module-project:configJSModules SKIPPED
:modules:my-module-project:classes UP-TO-DATE
:modules:my-module-project:jar UP-TO-DATE
```

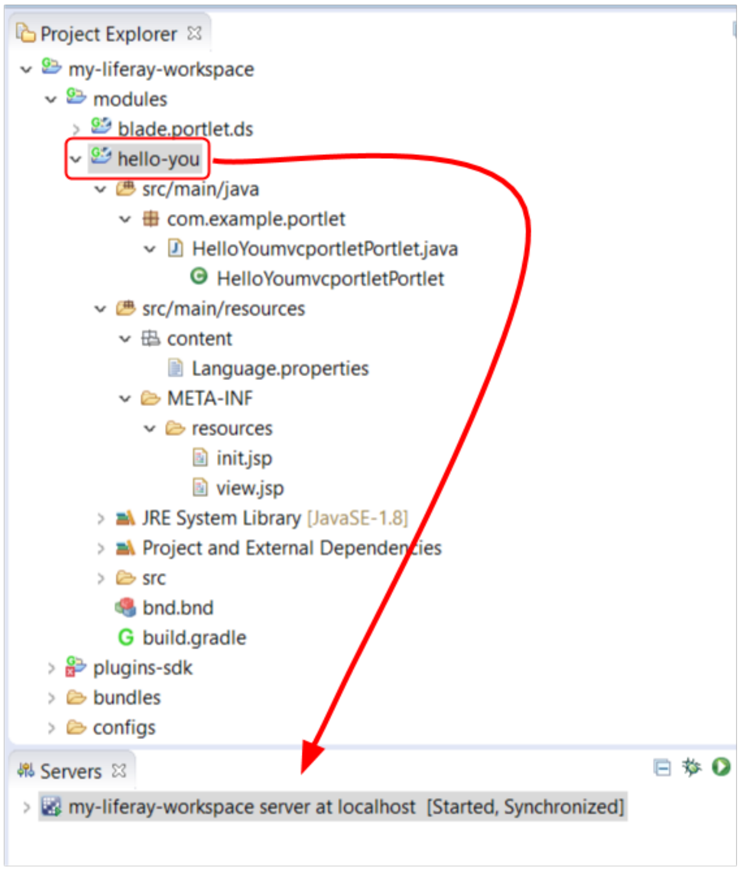
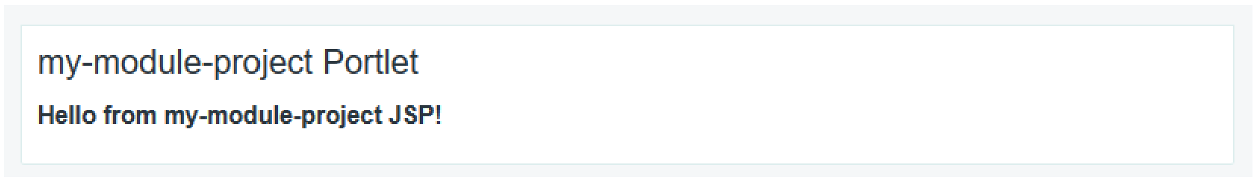


Figure 5.5: Liferay @ide@ lets developers deploy modules using drag-and-drop.


my-module-project Portlet

Hello from my-module-project JSP!

Figure 5.6: Here's a bare-bones portlet based on one of Liferay's module templates.

```
:modules:my-module-project:assemble UP-TO-DATE
:modules:my-module-project:build

BUILD SUCCESSFUL

Total time: 2.962 secs
install file:/E:/workspaces/my-liferay-workspace/modules/my-module-project/build/libs/com.liferay.docs.mymodule-1.0.0.jar
Bundle ID: 505
start 505

Scanning E:\workspaces\my-liferay-workspace\modules\my-module-project

...

Waiting for changes to input files of tasks... (ctrl-d then enter to exit)
```

Output from the `blade deploy -w` command indicates that the module is installed and started, reports the module's OSGi bundle ID, and stands ready to redeploy the module if its output files change.

Congratulations on a great start to developing your module!

5.5 Related Articles

Configuring Dependencies

Liferay Workspace

Tooling

OSGi Basics for Liferay Development

Portlets

Chapter 6

Configuring Dependencies

Using external artifacts in your project requires configuring their dependencies. To do this, look up the artifact's attributes and plug them into dependency entries for your build system (either Gradle, Maven, or Ant/Ivy). Your build system downloads the dependency artifacts your project needs to compile successfully.

Before specifying an artifact as a dependency, you must first find its attributes. Artifacts have these attributes:

- *Group ID*: Authoring organization
- *Artifact ID*: Name/identifier
- *Version*: Release number

This tutorial shows you how to make sure your projects have the right dependencies:

- Find Core artifacts
- Find Liferay app and independent artifacts
- Configure dependencies

6.1 Finding Core Artifacts

Each Liferay artifact is a JAR file whose `META-INF/MANIFEST.MF` file contains the artifact's OSGi metadata. The manifest also specifies the artifact's attributes. For example, these two OSGi headers specify the artifact ID and version:

```
Bundle-SymbolicName:  [artifact ID]
Bundle-Version: [version]
```

Important: Artifacts in Liferay DXP fix packs override Liferay DXP installation artifacts. Subfolders of a fix pack ZIP file's `binaries` folder hold the artifacts. If an installed fix pack provides an artifact you depend on, specify the version of that fix pack artifact in your dependency.

This table lists the group ID, artifact ID, version, and origin for each core Liferay DXP artifact:

Core Liferay DXP Artifacts:

File	Group ID	Artifact ID	Version	Origin
`portal-kernel.jar`	`com.liferay.portal`	`com.liferay.portal.kernel`	(see JAR's `MANIFEST.MF`)	fix pack ZIP, Liferay DXP installation, or Liferay DXP dependencies ZIP
`portal-impl.jar`	`com.liferay.portal`	`com.liferay.portal.impl`	(see JAR's `MANIFEST.MF`)	fix pack ZIP or Liferay DXP `.war`
`portal-test.jar`	`com.liferay.portal`	`com.liferay.portal.test`	(see JAR's `MANIFEST.MF`)	fix pack ZIP or Liferay DXP `.war`
`portal-test-integration.jar`	`com.liferay.portal`	`com.liferay.portal.test.integration`	(see JAR's `MANIFEST.MF`)	fix pack ZIP or Liferay DXP `.war`
`util-bridges.jar`	`com.liferay.portal`	`com.liferay.util.bridges`	(see JAR's `MANIFEST.MF`)	fix pack ZIP or Liferay DXP `.war`
`util-java.jar`	`com.liferay.portal`	`com.liferay.util.java`	(see JAR's `MANIFEST.MF`)	fix pack ZIP or Liferay DXP `.war`
`util-slf4j.jar`	`com.liferay.portal`	`com.liferay.util.slf4j`	(see JAR's `MANIFEST.MF`)	fix pack ZIP or Liferay DXP `.war`
`util-taglibs.jar`	`com.liferay.portal`	`com.liferay.util.taglib`	(see JAR's `MANIFEST.MF`)	fix pack ZIP or Liferay DXP `.war`
`com.liferay.*` JAR files	`com.liferay`	(see JAR's `MANIFEST.MF`)	(see JAR's `MANIFEST.MF`)	fix pack ZIP, Liferay DXP installation, Liferay DXP dependencies ZIP, or the OSGi ZIP

Next, you'll learn how to find artifacts for Liferay DXP apps and independent modules.

6.2 Finding Liferay App and Independent Artifacts

Independent modules and modules that make up Liferay DXP's apps aren't part of the Liferay DXP core. You must still, however, find their artifact attributes if you want to declare dependencies on them. The resources below provide the artifact details for Liferay DXP's apps and independent modules:

Resource	Artifact Type
App Manager	Deployed modules
Reference Docs	Liferay DXP modules (per release)
Maven Central	All artifact types: Liferay DXP and third party, module and non-module

Important: `com.liferay` is the group ID for all Liferay DXP's apps and independent modules.

The App Manager is the best source for information on deployed modules. You'll learn about it next.

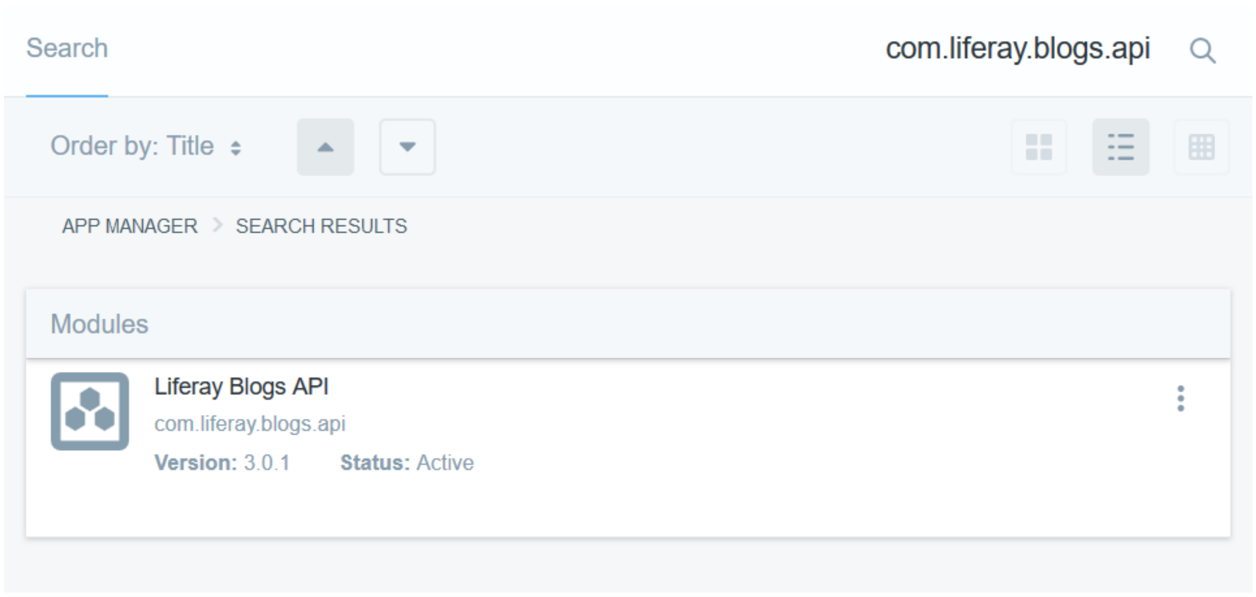
App Manager

The App Manager knows what's deployed on your Liferay DXP server. You can use it to find whatever modules you're looking for.

Follow these steps to get a deployed module's information:

1. In Liferay DXP, navigate to *Control Panel* → *Apps* → *App Manager*.

2. Search for the module by its display name, symbolic name, or related keywords. You can also browse for the module in its app. Whether browsing or searching, the App Manager shows the module's artifact ID and version number.



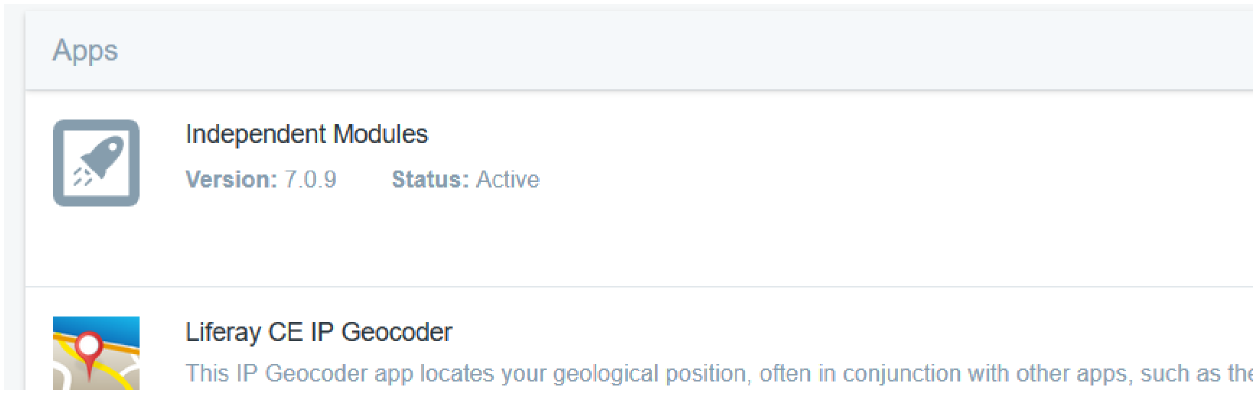
Liferay Blogs API
com.liferay.blogs.api
Version: 3.0.1 **Status:** Active

Figure 6.1: You can inspect deployed module artifact IDs and version numbers.

If you don't know a deployed module's group, use the Felix Gogo Shell to find it:

1. Open a Gogo Shell session by entering the following into a command prompt:

```
telnet localhost 11311
```


Apps

Independent Modules

Version: 7.0.9 Status: Active

Liferay CE IP Geocoder

This IP Geocoder app locates your geological position, often in conjunction with other apps, such as the

Figure 6.2: The App Manager aggregates Liferay and independent modules.

This results in a `g!`: the Felix Gogo Shell command prompt.

2. Search for the module by its display name (e.g., `Liferay Bookmarks API`) or a keyword. In the results, note the module's number. You can use it in the next step. For example, these results show the Liferay Bookmarks API module's number is 52:

```
g! lb | grep "Liferay Bookmarks API"

   52|Active     |   10|Liferay Bookmarks API (2.0.1)
```

3. To list the module's manifest headers, pass the module number to the `headers` command. In the results, note the `Bundle-Vendor` value: you'll match it with an artifact group in a later step:

```
g! headers 52

Liferay Bookmarks API (52)
--------------------------
Manifest-Version = 1.0
Bnd-LastModified = 1464725366614
Bundle-ManifestVersion = 2
Bundle-Name = Liferay Bookmarks API
Bundle-SymbolicName = com.liferay.bookmarks.api
Bundle-Vendor = Liferay, Inc.
Bundle-Version = 2.0.1
...
```

4. Disconnect from the Gogo Shell session:

```
g! disconnect
```

5. On Maven Central or MVNRepository, search for the module by its artifact ID.

6. Determine the group ID by matching the `Bundle-Vendor` value from step 3 with a group listed that provides the artifact.

Next, you'll learn how to use Liferay DXP's reference documentation to find a Liferay DXP app module's attributes.

Reference Docs

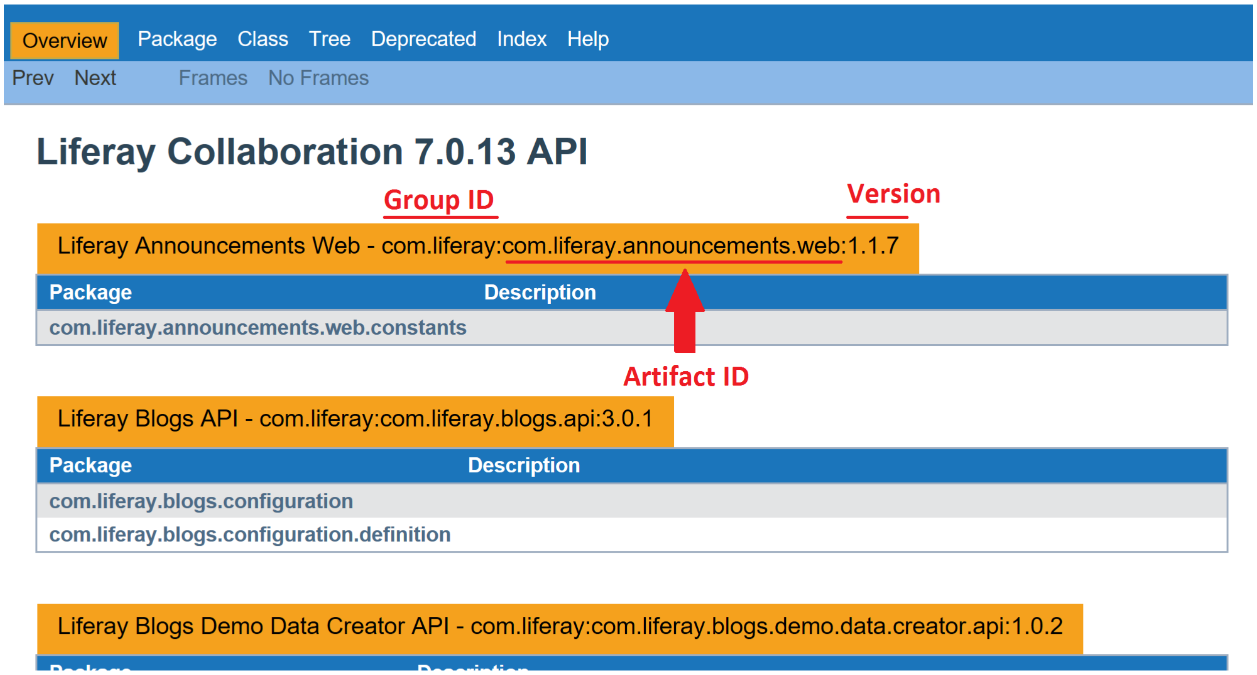
Liferay DXP's app Javadoc lists each app module's artifact ID, version number, and display name. This is the best place to look up Liferay DXP app modules that aren't yet deployed to your Liferay DXP instance.

Note: To find artifact information on a Core Liferay DXP artifact, refer to the previous section Finding Core artifacts.

Follow these steps to find a Liferay DXP app module's attributes in the Javadoc:

1. Navigate to Javadoc for an app module class. If you don't have a link to the class's Javadoc, find it by browsing @app-ref@.
2. Copy the class's package name.
3. Navigate to the *Overview* page.
4. On the *Overview* page, search for the package name you copied in step 2.

The heading above the package name shows the module's artifact ID, version number, and display name. Remember, the group ID for all app modules is `com.liferay`.



Liferay Collaboration 7.0.13 API

Liferay Announcements Web - com.liferay:com.liferay.announcements.web:1.1.7

Package	Description
com.liferay.announcements.web.constants	

Liferay Blogs API - com.liferay:com.liferay.blogs.api:3.0.1

Package	Description
com.liferay.blogs.configuration	
com.liferay.blogs.configuration.definition	

Liferay Blogs Demo Data Creator API - com.liferay:com.liferay.blogs.demo.data.creator.api:1.0.2

Figure 6.3: Liferay DXP app Javadoc overviews list each app module's display name, followed by its group ID, artifact ID, and version number in a colon-separated string. It's a Gradle artifact syntax.

Note: Module version numbers aren't currently included in any tag library reference docs.

Next, you'll learn how to look up artifacts on MVNRepository and Maven Central.

Maven Central

Most artifacts, regardless of type or origin, are on MVNRepository and Maven Central. These sites can help you find artifacts based on class packages. It's common to include an artifact's ID in the start of an artifact's package names. For example, if you depend on the class `org.osgi.service.component.annotations.Component`, search for the package name `org.osgi.service.component.annotations` on one of the Maven sites.

Note: Make sure to follow the instructions listed earlier to determine the version of Liferay artifacts you need.

Now that you have your artifact's attribute values, you're ready to configure a dependency on it.

6.3 Configuring Dependencies

Specifying dependencies to build systems is straightforward. Edit your project's build file, specifying a dependency entry that includes the group ID, artifact ID, and version number.

Note: To configure third-party libraries in a module, see the tutorial Adding Third Party Libraries to a Module.

Note that different build systems use different artifact attribute names, as shown below:
Artifact Terminology

Framework	Group ID	Artifact ID	Version
Gradle	`group`	`name`	`version`
Maven	`groupId`	`artifactId`	`version`
Ivy	`org`	`name`	`rev`

The following examples demonstrate configuring a dependency on Liferay's Journal API module for Gradle, Maven, and Ivy.

Gradle

Here's the dependency configured in a `build.gradle` file:

```
dependencies {
    compileOnly group: "com.liferay", name: "com.liferay.journal.api", version: "1.0.1"
    ...
}
```

Maven

Here's the dependency configured in a `pom.xml` file:

```
<dependency>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.journal.api</artifactId>
    <version>1.0.1</version>
</dependency>
```

Ivy

Here's the dependency configured in an `ivy.xml` file:

```
<dependency name="com.liferay.journal.api" org="com.liferay" rev="1.0.1" />
```

Important: Liferay DXP exports many third-party packages. If you're developing a WAR, deploy it to check if the packages you're using are in the OSGi runtime container already. If they are already in there, specify their corresponding artifacts as being "provided". Here's how to specify a provided dependency:

Maven: `<scope>provided</scope>`

Gradle: `providedCompile`

Don't deploy a provided package's JAR again or embed the JAR in your project. Exporting the same package from different JARs leads to "split package" issues, whose side affects differ from case to case. If the package is in a third-party library (not an OSGi module), refer to [Resolving Third

---

Party Library Dependencies](/docs/7-0/tutorials/-/knowledge_base/t/adding-third-party-libraries-to-a-module).

If you're developing a WAR that requires a different version of a third-party package that Liferay DXP or another module exports, specify that package in your `Import-Package:` list. If the package provider is an OSGi module, publish its exported packages by deploying that module. Otherwise, follow the instructions for adding a third-party library (not an OSGi module).

Nice! Now you know how to find artifacts and configure them as dependencies. Now that's a skill you can depend on!

6.4 Related Topics

Importing Packages

Reference

Liferay API Modules

Adding Third Party Libraries to a Module

Third Party Packages Portal Exports

Classes Moved from portal-service.jar

Tooling

Portlets

Chapter 7

Finding Extension Points

Liferay DXP provides many features that help users accomplish their tasks. Sometimes, however, you may find it necessary to customize a built-in feature. It's easy to **find** an area you want to customize, but it may seem like a daunting task to figure out **how** to customize it. Liferay DXP was developed for easy customization, meaning there are many extension points you can use to add your own flavor.

There's a process you can follow that makes finding an extension point a breeze.

1. Locate the bundle (module) that provides the functionality you want to change.
2. Find the components available in the module.
3. Discover the extension points for the chosen component.

In this tutorial, you'll learn how to find an extension point. You'll step through a simple example that locates an extension point for importing LDAP users. This will require the use of Liferay DXP's Application Manager and Felix Gogo Shell.

7.1 Locate the Related Module and Component

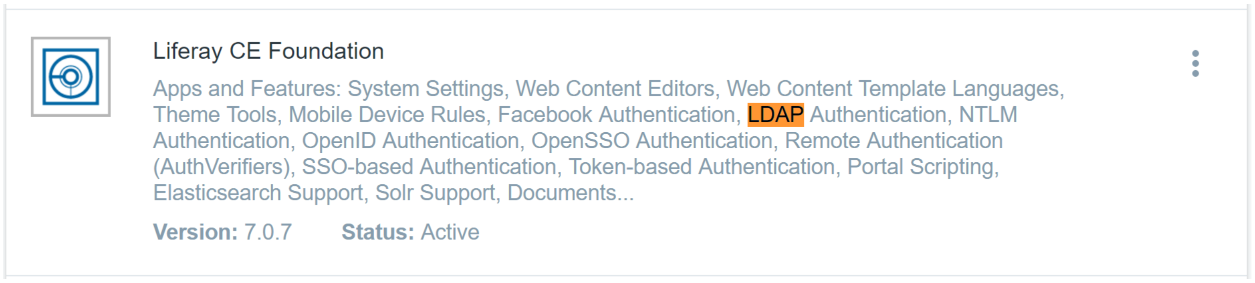
You must first think of words that describe the application behavior you want to change. With the right keywords, you can easily track down the desired module and its component. Consider the example for importing LDAP users. Some candidate keywords for finding the component are *import*, *user*, and *LDAP*.

The easiest way to discover the module responsible for a particular feature in Liferay DXP is to use the Application Manager. The Application Manager lists app suites and their included modules/components in an easy-to-use interface. It even lists third party apps! You'll use your keywords to target the applicable component.

1. Open the App Manager by navigating to *Control Panel → Apps → App Manager*. The top level lists app suites, independent apps, and independent modules.

2. Navigate the app suites, apps, and modules, or use Search to find components that might provide your desired extension point. Remember to check for your keywords in element names and descriptions. The keyword *LDAP* resides under the Liferay Foundation app suite; select it.

3. Select the *LDAP* application from the app listing.

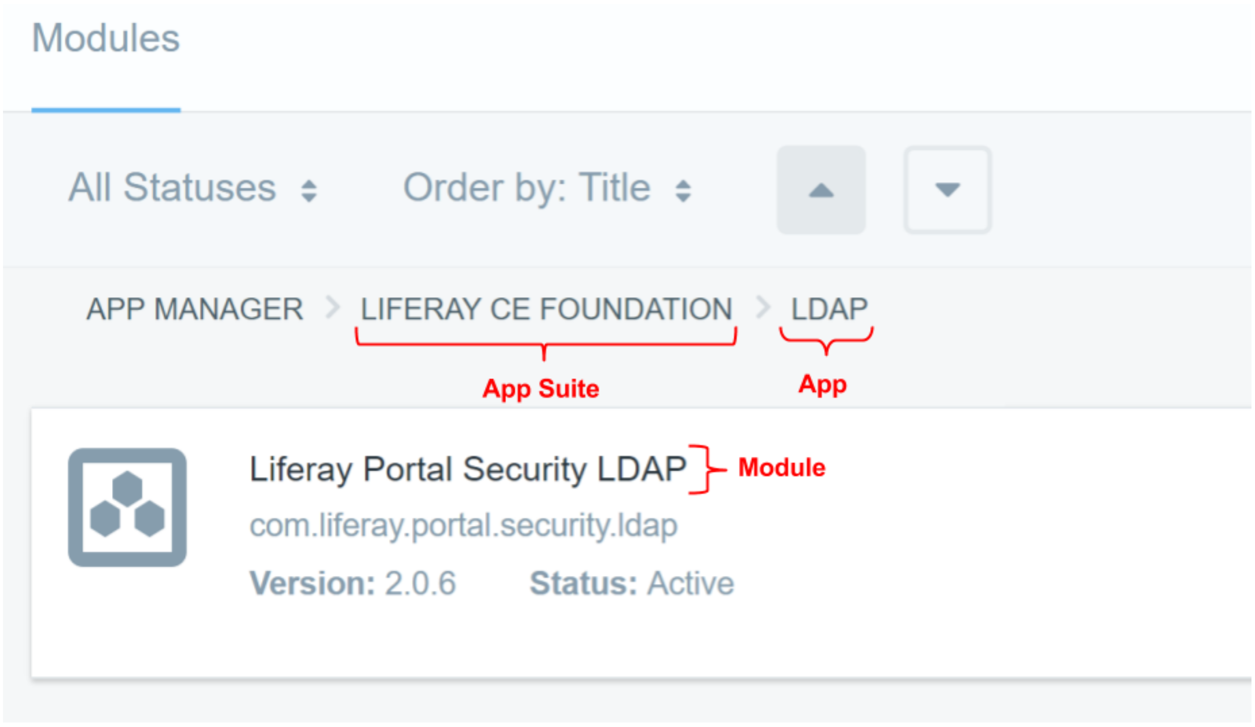

Liferay CE Foundation

Apps and Features: System Settings, Web Content Editors, Web Content Template Languages, Theme Tools, Mobile Device Rules, Facebook Authentication, LDAP Authentication, NTLM Authentication, OpenID Authentication, OpenSSO Authentication, Remote Authentication (AuthVerifiers), SSO-based Authentication, Token-based Authentication, Portal Scripting, Elasticsearch Support, Solr Support, Documents...

Version: 7.0.7 **Status:** Active

Figure 7.1: The Liferay Foundation app suite contains the LDAP Authentication application.

4. The LDAP application only has one module, but typically, applications have more than one module to inspect. Select the *Liferay Portal Security LDAP* module.



Liferay Portal Security LDAP

com.liferay.portal.security.ldap

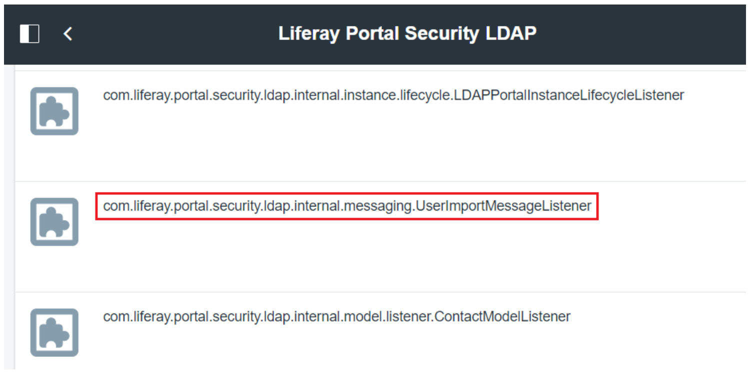
Version: 2.0.6 **Status:** Active

Figure 7.2: The App Manager lists the module, package name, version, and status.

5. Search through the components, applying your keywords as a guide. Copy the component name you think best fits the functionality you want to customize; you'll inspect it later using the Gogo shell.

```
**Note:** When using the Gogo shell later, understand that it can take
several tries to find the component for which you're looking; naming
conventions should allow you to find your desired extension point in a
manageable time frame.
```

Next, you'll begin using the Gogo shell to inspect the component for extension points.



Liferay Portal Security LDAP

com.liferay.portal.security.ldap.internal.instance.lifecycle.LDAPPortalInstanceLifecycleListener

com.liferay.portal.security.ldap.internal.messaging.UserImportMessageListener

com.liferay.portal.security.ldap.internal.model.listener.ContactModelListener

Figure 7.3: The component name can be found using the App Manager.

7.2 Finding Extension Points in a Component

Once you have the component that relates to the functionality you want to extend, you can use the Gogo shell's Service Component Runtime (SCR) commands to inspect it. You can execute SCR commands using Liferay Blade CLI or in Gogo shell via telnet. This tutorial assumes you're using the Gogo shell via telnet.

Execute the following command:

```
scr:info [COMPONENT_NAME]
```

For the LDAP example component you copied previously, the command would look like this:

```
scr:info com.liferay.portal.security.ldap.internal.messaging.UserImportMessageListener
```

The output includes a lot of information. For this exercise, you're interested in the services the component references. These are extension points. For example, here's the reference for the service that imports LDAP users:

```
...
Reference: LdapUserImporter
Interface Name: com.liferay.portal.security.ldap.exportimport.LDAPUserImporter
Cardinality: 1..1
Policy: static
Policy option: reluctant
Reference Scope: bundle
...
```

The `LDAPUserImporter` is the extension point needed to customize the process of importing users with LDAP! If none of the references satisfy what you're looking for, search other components from the App Manager.

If you plan on overriding the referenced service, you'll need to understand the reference's policy and policy option. If the policy is `static` and the policy option is `reluctant`, binding a new higher ranking service in place of a bound service requires reactivating the component or changing the target. For information on the other policies and policy options, visit the OSGi specification, in particular, sections 112.3.5 and 112.3.6. If you want to learn how to override a component's service reference, visit the following tutorial.

Important Not all Extension points in Liferay DXP are available as referenced services. Referenced services are common extension points when using Declarative Services (DS), but there are extension points not exposed this way. If your project does not use the DS component framework, you'd need to look for the API that describes its service consumption from the OSGi registry. Here's a brief list of other potential extension points in Liferay DXP:

- Instances of `org.osgi.util.tracker.ServiceTracker<S, T>`
- Uses of Liferay's `Registry.getServiceTracker`
- Uses of Liferay's `ServiceTrackerMap` or `ServiceTrackerCollection`
- Any other component framework or whiteboard implementation (e.g., HTTP, JAX-RS) that supports tracking services; Blueprint, Apache Dependency Manager, etc. could also introduce extension points.

There you have it! You successfully formulated keywords that described the functionality you wanted to customize, used those keywords in the App Manager to target the applicable module component, and used the Gogo shell to inspect the component for extension points.

Chapter 8

From Liferay Portal 6 to 7

Becoming familiar with a platform as large and fully featured as Liferay is a big task. You learn the ins and outs of what it can do, the tips and best practices of the experts, and you work your way through the APIs. As you do this, you become more and more familiar with how things work, become more proficient with the platform as you multiply successes on it, and start to think in terms of how you'd solve problems most effectively using the tools the platform gives you. Eventually, if you use it long enough, it can seem like an old friend that's ready to stand by you and help you succeed in your projects.

7.0 was designed as an enhancement that builds off of what you already know. Its Upgrade Planner and this tutorial series–or Learning Path–help get your existing plugins running on 7.0 right away. The tool automates much of the process. After you upgrade your plugins, you can build and deploy them as you always have.

7.0 has exciting improvements for developers too. This Learning Path shows you how to leverage them. Since you already know previous versions of Liferay Portal, you're several steps ahead of everybody else.

This Learning Path describes the benefits of 7.0 for developers compared to previous versions, the architectural improvements, the benefits that modularity brings, and how to develop modules and how they differ from traditional plugins. You'll see all the options for leveraging new developer features, learn the pros and cons of each, and examine steps for optimizing your existing plugins for 7.0.

In the end, we believe you'll both want to adopt 7.0, and you'll see how you can thrive using it.

Note: If you want to learn about 7.0's architectural improvements, OSGi and modularity, and tooling improvements, read on. If you're more interested in upgrading your plugins first, skip to Planning Plugin Upgrades and Optimizations.

You'll start by seeing the familiar, good things that remain the same and then examine what's changed the most since Liferay Portal 6.

Chapter 9

What Hasn't Changed and What Has

7.0 is a new major version of the Liferay platform and as such it includes many improvements over previous versions. Having said that, most of the characteristics from Liferay Portal 6 that you have learned to love are preserved, having been changed only slightly or not at all. Any experienced Liferay developer will be able to reuse most of his/her existing knowledge to developing for 7.0.

What has not changed? Even though there are many improvements in 7.0, there are also many great familiar aspects from previous versions that have been preserved. Here are some of the most relevant ones:

1. The Portal Core and each Liferay app continue to use the three layer architecture: presentation, services, and persistence. The presentation layer is now always provided as an independent module, facilitating replacing it with a different presentation, if desired.

2. Support remains for previously supported standards such as Portlets (JSR-168, JSR-286), CMIS, WebDAV, LDAP, JCP (JSR-170), etc.

3. Most Liferay APIs have remained functionally similar to those of 6.2, even if many of their classes have moved to new packages, as part of the modularization effort.

4. Liferay @ide@ is still the preferred tool to develop for Liferay, even though you are still free to use tools that best fit your needs.

5. Service Builder and other developer tools and libraries continue to work as they have in 6.2.

6. Traditional plugins for portlets and hooks still work (once they're adapted to 7.0's API) through a compatibility layer.

7. The Plugins SDK can also still be used and transition to the new Liferay Workspace, if desired, is easy.

Here are some key changes of interest to existing Liferay developers:

1. Extraction of many features as modules: So far you have been used to working with Liferay as a large web application, of which all of it had to be deployed or none of it. In 7.0, many out of the box portlets, features, and associated APIs have been extracted as OSGi modules. You can choose which ones to deploy and use.

2. Adoption of modern OSGi standards: OSGi is a set of standards for building modular systems. It's very powerful. Although it was previously difficult to learn and use, its modernized standards, such as Declarative Services, have made learning and using it much easier.

3. Core Public APIs are provided through portal-kernel (previously known as portal-service); all other public APIs are provided by their own modules.

4. You can reuse modules and libraries, and manage the dependencies among them.

5. Registration of classes implementing extension points is now simpler and more consistent; it's based on the standard `@Component` annotation instead of declarations in `portal.properties` or `portlet.xml`. Note, previous registration mechanisms have been preserved where possible. See the Breaking Changes article to examine where extensions and configurations that have not kept backwards compatibility.

6. Third party extensions and applications are now first-class citizens. Traditional plugins had some limitations that developments done in the core (or done as Ext Plugins) did not have. Modules don't have these limitations and are much more powerful than plugins ever were.

7. Complete integration of Liferay specific tools (such as Service Builder) within Maven and Gradle. Additionally we've adopted some new tools such as Bnd.

Since the modularization of the Liferay web application is the change most relevant to you as a developer, let's dig deeper into that change and how it affects Liferay's architecture.

9.1 Embracing a Modular Architecture

The largest improvement in Liferay's architecture is the adoption of a modular development paradigm. Within each Liferay module (or group of modules that form an app), as well as within what remains as Liferay's core, the existing great characteristics of previous versions of Liferay prevail.

Tiered Architecture

Liferay Portal 6's architecture diagrams often focused on the tiers for the frontend, services layer (for the business logic), and persistence layer (mostly auto-generated by Service Builder). These layers still exist and have been embraced throughout the modularization effort.

The most significant change (and improvement) over this architecture is that the portal is no longer a single large Java EE Web Application. Liferay has been broken down into many modules to benefit from the Modular Development Paradigm. Those benefits are described in the next section. The modules are often grouped into apps (such as Wiki or Message Boards) and the main apps are grouped into suites (such as Web Experience, Collaboration, and Forms & Workflow).

Modular Architecture

The figure below represents 7.0's architecture from a structural perspective.

Liferay Core

As its name implies, it's 7.0's central and most important part. The Liferay Core is a Java EE application in charge of bootstrapping the system and receiving and delegating all requests. It also contains Liferay's OSGi Engine on top of which all applications run.

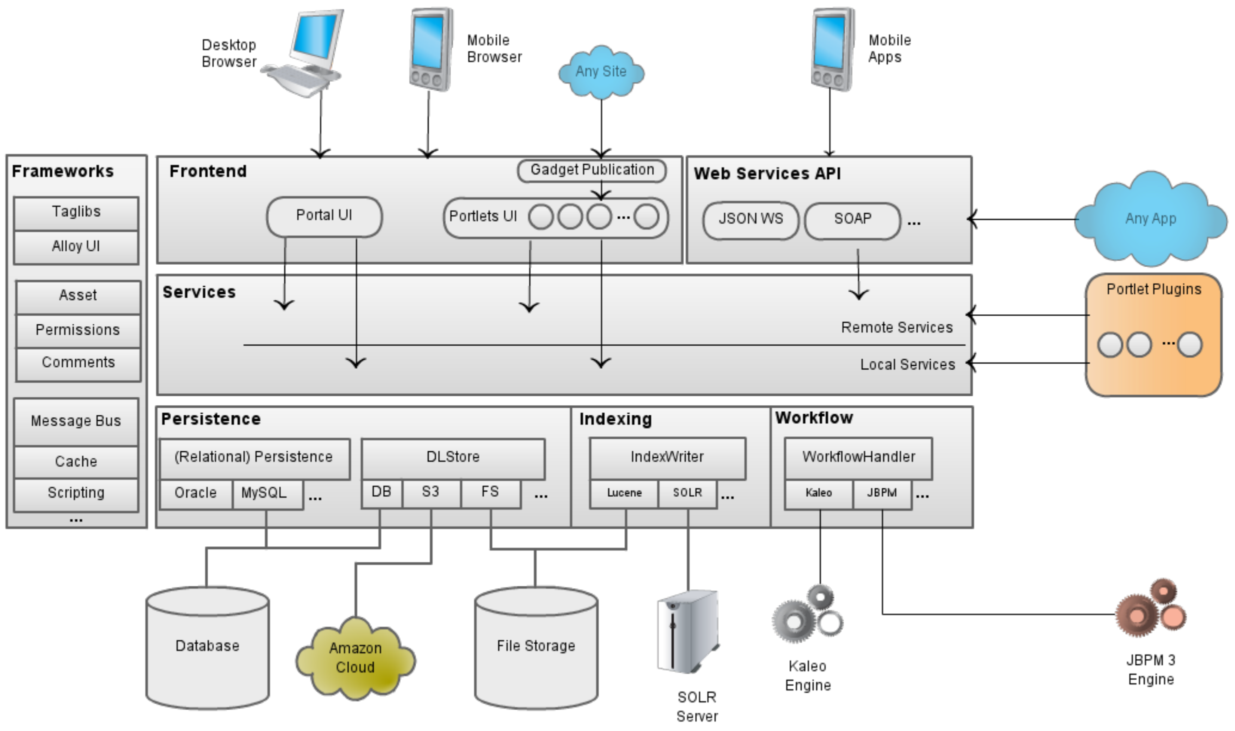


Figure 9.1: Liferay Portal 6's architecture, shown in this figure, is still generally valid in 7.0.

Foundation

The Foundation suite sits on top of the core, providing administrative interfaces and familiar development building blocks. It includes modules for user and role administration, LDAP integration, authentication, licensing, upgrades, clustering, DAO, and front-end mainstays for themes, CSS, taglibs, and JavaScript. The Foundation suite's modules depend on Liferay Core, as do all the App Suites and non-core modules.

Most of the apps, frameworks, and APIs you've come to know and love have been aggregated in App Suites. The suites are available in Liferay bundles and are also available on the Marketplace. Here are the different App Suites:

Liferay Web Experience

Contains apps such as Web Content and Site management, Web Content Display, Asset Publisher, and Breadcrumbs and features and frameworks such as Application Display Templates, Tags, and Recycle Bin.

Liferay Collaboration

Comprises Liferay's social apps and collaboration apps, such as Message Boards, Wiki, and Blogs. It also contains Liferay's Documents & Media Library.

Liferay Forms and Workflow

Provides apps such as Forms (New!), Dynamic Data Lists, Kaleo Workflow, and Calendar. It also contains the Dynamic Data Mapping framework used by Web Content and Documents & Media to provide custom form and templating capabilities.

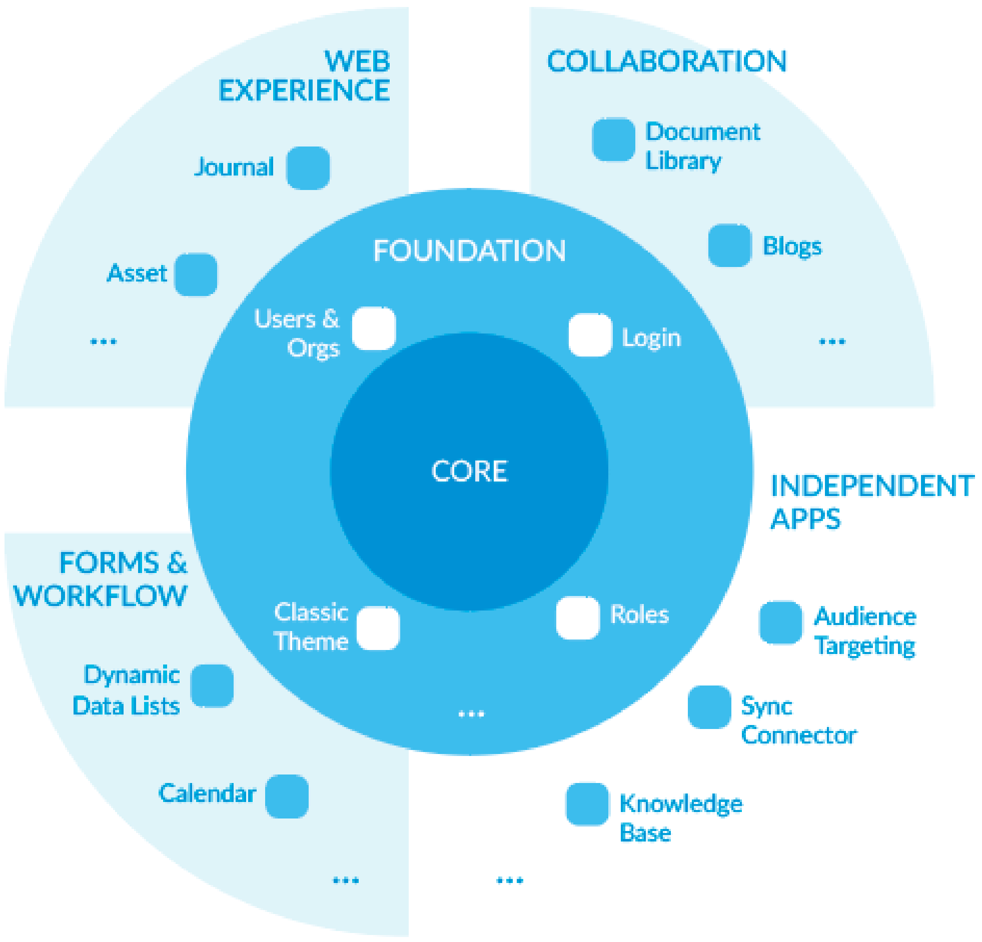


Figure 9.2: 7.0 is composed of the Liferay Core, independent application modules, and App Suites, each with their own set of application and framework modules.

Independent Apps

Last but not least, Liferay's independent apps and modules also play a part. They provide unique functionality and stand on their own; it would be unnatural to add any one of them to a particular suite. Apps such as Liferay Sync, the Marketplace Client, Knowledge Base, and many more apps available on the Marketplace are independent Liferay apps.

The beauty of the 7.0 ecosystem is that it is made up of simple easy-to-use modules that depend on and communicate with each other. And you as a third-party developer can create and deploy your own modules into the mix.

You can continue developing traditional WAR-style apps for 7.0 too. Liferay DXP's Portlet Compatibility Layer converts each plugin WAR to a Web Application Bundle (WAB), which is a module.

Let's consider the structure of a 7.0 modular app.

The Structure of a Modular App

As mentioned, each app can be formed by one or more modules. This section explains the most common way to structure an app.

The best practice for structuring an app is in several modules. In particular the following modules are the often the best way of structuring an app:

- **Service**: Contains the service (business logic) and persistence implementations.
- **API**: Contains the public API of the application. By being separate from the service it's simpler and faster to deploy new versions of the implementation without affecting any module using the API. It also allows changing the versioning of the implementation independent from the versioning of the API.
- **Web**: Contains the presentation tier, very often the portlets provided by this app.
- **Test**: Contains the tests. These are not included in the app for production.
- **Specific purpose modules**: Other modules are also often created for specific purposes or to provide alternative implementations of some of the app's features. For example the Wiki app has one module for each of the supported Wiki Engines.

All the modules in an app usually sit in directories next to each other in the source to facilitate referencing them.

For deployment to production Liferay provides the LPKG packaging format that allows bundling a set of modules into a single file and add additional metadata about it. This format can also be used to upload apps to Liferay's Marketplace.

Now you have a basic understanding of the architectural changes introduced in 7.0 and have become acquainted with the new structure used in Liferay's apps. You have learned some key concepts that are new for Liferay Portal 6 developers and have been assured about developer features you've used in previous Liferay releases that have been carried into 7.0.

Next, you'll explore how these new concepts and the new modular architecture benefit you as a developer.

Chapter 10

Benefits of 7.0 for Liferay Portal 6 Developers

More than in any other Liferay release, 7.0 centers on you, the developer. Liferay's platform has been rebuilt, making it easier to build on and maintain, and providing more new developer features than any previous Liferay release.

Here are some key benefits of this release for developers:

1. **Simpler and Leaner**
2. **Modular Development Paradigm**
3. **Enhanced Reusability**
4. **More extensible, easier to maintain**
5. **Optimized for your tooling of choice**
6. **Powerful Configurability**

Let's consider how they make development easier for you.

10.1 Simpler and Leaner

Liferay has always been simple and lean, compared to the proprietary alternatives; this version widens the gap even more.

7.0 is simpler than its predecessors, thanks to a streamlined and modular architecture. In addition, many Liferay specific ways of creating extensions and applications have evolved to follow official or de-facto standards. As a result, developers can now more easily reuse their existing knowledge and use what they learn developing for Liferay outside of it.

7.0 is also leaner. Its modularized core allows developers and system administrators to remove parts they don't need or don't want; this facilitates deployment, reduces startup times and memory footprints, and results in more efficiencies and performance improvements.

10.2 Modular Development Paradigm

If you have been using Liferay, you've already experienced some of the benefits of modular development, thanks to plugins. 7.0 takes these benefits to a whole new level.

In addition to building plugins as you have previously, you can take advantage of a complete module development and runtime system based on OSGi standards. 7.0 facilitates creating applications of all types by composing and reusing modules.

And don't worry, modules are easy to understand. A module is distributed as a JAR file and can be as small as one Java class or as large as any application you can think of. An application for Liferay can comprise one single module or as many modules as you want. The cool thing is that modules can cooperate, allowing you to build applications by combining smaller pieces that are easier to develop, deploy, maintain, and reuse.

10.3 Enhanced Reusability

If you have worked on large developments on top of Liferay you have probably experienced situations in which you wanted to share a subset of classes from from one plugin with another.

Java EE does not provide any standard way to achieve this, but Liferay provided certain capabilities to achieve it with a mechanism known as CLP that used class loader *magic* to allow plugins to invoke services in other plugins created with Service Builder. This mechanism, however, is still a bit limited (Java EE's class loader doesn't allow for much more) and doesn't give you the freedom to specify any or all classes from one module to use from within another module.

7.0 enables greater reusability, both in code and runtime memory, several folds. For any desired reusable functionality you just create a module (remember, it's just a JAR file with some metadata) with the classes you want and deploy it. Other modules need only declare that they use the classes in that module (by specifying their packages) and 7.0 automatically wires them together. All invocations are regular Java calls! Try it out; it's beautiful. :)

This mechanism eliminates the dreaded "JAR/classpath hell" issue. No longer do you have to jockey JAR files in classpaths; nor do you have to implement intricate class loaders. The runtime environment uses separate class spaces per module; it even accommodates using multiple versions of libraries in the same application (as long as they can coexist).

10.4 More Extensible, Easier to Maintain

Whenever we ask Liferay developers what is their favorite characteristic of Liferay, "Great extensibility" is one of the top three most popular responses. You can customize almost every detail and add your own functionality on top.

Is 7.0 more extensible? You bet! Many more extension points have been added. But not only that, all new extension points and many existing ones which have been upgraded, use a new extension mechanism based on OSGi's service model. Here are some of the mechanism's benefits:

1. **Simpler**: An implementation of an extension point is now always a Java class that implements an interface and has one annotation (@Component). That's it; it couldn't be any easier.

2. **Easier to maintain**: Extension points are now more strictly defined through a Java interface that uses Semantic Versioning rules. This means that your extensions can work without changes, even across several Liferay versions, as long as the specific extension API is backwards compatible.

3. **Dynamic**: Extensions can be loaded and removed at any time during development or in production.

But that is not all. Your own developments can now also leverage this model and become extensible. You can create simple extension points by just creating an interface and annotating a setter method with an annotation (@Reference). Implementing extensibility has never been easier.

10.5 Optimized for Your Tooling of Choice

7.0 empowers you to use the tools you like.

If you don't have strong preferences and are open to our suggestions, we offer Liferay Workspace. It provides an opinionated directory structure and build system based on Gradle and Bnd. Liferay Workspace can be used standalone through the command line or with Liferay @ide@, which runs on Eclipse.

And if you want to continue using the Plugins SDK, we've got you covered. The Plugins SDK is available to facilitate your transition to 7.0. In fact, a Plugins SDK structure can reside in a Liferay Workspace alongside new developments that use the new build environment; you can switch between traditional projects and new projects at your own pace.

Finally, we have also developed a lightweight tool called Blade CLI, which facilitates starting new projects from templates – it's especially useful for Gradle which doesn't have Maven's concept of archetypes. Blade CLI also offers commands to start/stop the server and deploy and administer modules.

10.6 Powerful Configurability

Creating configurable code is a breeze with 7.0. And applications that use Liferay's new Configuration API allow administrators to change the configuration on the fly, through an auto-generated user interface called System Settings.

Now you understand how 7.0 enriches your experience as a developer and makes developing apps and customizations fun.

Next, we'll take a look at OSGi and modularity to discuss key concepts and demonstrate how easy and gratifying it is to build modules.

Chapter 11

OSGi and Modularity for Liferay Portal 6 Developers

To create a powerful, reliable platform for developing modular applications, Liferay sought best-of-breed standards-based frameworks and technologies. It was imperative not only to meet demands for enterprise digital experiences but also to offer developers, both experienced with Liferay and new to Liferay, a clear and elegant way to create apps.

Here were some of the key goals:

- Allow breaking down a large system into smaller pieces of code, whose boundaries and relationships could be clearly defined.
- Explicitly differentiate public APIs from private APIs.
- Facilitate extensibility of existing code.
- Modernize the development environment, leveraging more state-of-the-art tools to provide a great developer experience.

It wasn't long before Liferay discovered that OSGi and its supporting tools/technologies fit the bill!

In this tutorial, you'll learn how 7.0 uses OSGi to meet these objectives. And equally important, you'll find out how easy and fun modular development can be.

Here are the topics you'll dig into:

1. Modules as an Improvement over Traditional Plugins: Development and customization of applications for Liferay has been done traditionally in WAR-style plugins (Portlet, Hook, Ext, and Web). In 7.0, traditional Liferay plugins can be replaced with (or can be automatically converted to) modules. You'll see the similarities and differences of plugins and modules, and you'll learn the benefits of using modules.
2. Leveraging Dependencies: In 7.0, you can both declare dependencies among modules and combine modules to create applications. Since leveraging dependencies provides huge benefits, it's important to devote a large section for it.

3. OSGi Services and Dependency Injection: OSGi provides a powerful concept called OSGi Services (also known as microservices). OSGi's Declarative Services standard provides a clean way to inject dependencies in a dynamic environment. This is similar to Spring DI, except the changes happen while the system is running. It also offers an elegant extensibility model that 7.0 leverages extensively.

4. Dynamic Deployment: Module deployment is managed by 7.0 (not the application server). This section demonstrates how to use dynamic deployment for better control and efficiency.

After investigating these topics, you'll get hands-on experience creating and deploying an OSGi module. Let's start with learning how modules are better than traditional plugins.

11.1 Modules as an Improvement over Traditional Plugins

In 7.0, you can develop applications using OSGi modules or using traditional Liferay plugins (WAR-style portlets, hooks, EXT, and web applications). Liferay's Plugin Compatibility Layer (explained later) makes it possible to deploy traditional plugins to the OSGi runtime framework. To benefit from all 7.0 and OSGi offer, however, you should use OSGi modules.

Modules offer these benefits:

- **Better Encapsulation** - The only classes a module exposes publicly are those it exports explicitly. This lets you define internal public classes transparent to external clients.

- **Dependencies by Package** - Dependencies are specified by Java package, not by JAR file. In traditional plugins, you had to add *all* of a JAR file's classes to the classpath to use *any* of its classes. With OSGi, you need only import packages containing the classes you need. Only the classes in those packages are added to the module's classpath.

- **Lightweight** - A module can be as small as you want it to be. In contrast to a traditional plugin, which may require several descriptor files, a module requires only a single descriptor file–a standard JAR manifest. Also, traditional plugins are typically larger than modules and deployed on app server startup, which can slow down that process considerably. Modules deploy more quickly and require minimal overhead cost.

- **Easy Reuse** - Modules lend themselves well to developing small, highly cohesive chunks of code. They can be combined to create applications that are easier to test and maintain. Modules can be distributed publicly (e.g., on Maven Central) or privately. And since modules are versioned, you can specify precisely the modules you want to use.

- **In-Context Descriptors** - Where plugins use descriptor files (e.g., `web.xml`, `portlet.xml`, etc.) to describe classes, module classes use OSGi annotations to describe themselves. For example, a module portlet class can use OSGi Service annotation properties to specify its name, display name, resource bundle, public render parameters, and much more. Instead of specifying that information in descriptor files separate from the code, you specify them in context in the code.

These are just a few ways modules outshine traditional plugins. Note, however, that developers experienced with Liferay plugins have the best of both worlds. 7.0 supports traditional plugins *and* modules. Existing Liferay developers can find comfort in the simplicity of modules and their similarities with plugins.

Here are some fundamental characteristics modules share with plugins:

- Developers use them to create applications (portlets for Liferay)

- They're zipped up packages of classes and resources

- They're packaged as a standard Java JARs

Now that you've compared and contrasted modules with plugins, it's time to tour the module anatomy.

Module Structure: A JAR File with a Manifest

A module's structure is extremely simple. It has one mandatory file: `META-INF/MANIFEST.MF`. You add code and resources to the module and organize them as desired.

Here's the essential structure of a module JAR file:

```
- [Module's files]
_ META-INF
    _ MANIFEST.MF
```

The `MANIFEST.MF` file describes the module to the system. The manifest's OSGi headers identify the module and its relationship to other modules.

Here are some of the most commonly used headers:

- `Bundle-Name`: User friendly name of the module.

- `Bundle-SymbolicName`: Globally unique identifier for the module. Java package conventions (e.g., `com.liferay.journal.api`) are commonly used.

- `Bundle-Version`: Version of the module.

- `Export-Package`: Packages from this module to make accessible to other modules.

- `Import-Package`: Packages this module requires that other modules provide.

Other headers can be used to specify more characteristics, such as how the module was built, development tools used, etc.

For example, here are some headers from the Liferay Journal Web module manifest:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Liferay Journal Web
Bundle-SymbolicName: com.liferay.journal.web
Bundle-Vendor: Liferay, Inc.
Bundle-Version: 1.1.2
Export-Package:\
    com.liferay.journal.web.asset,\
    com.liferay.dynamic.data.mapping.util,\
    com.liferay.journal.model,
    com.liferay.journal.service,com.liferay.journal.util, [..]
Import-Package:\
    aQute.bnd.annotation.metatype,\
    com.liferay.announcements.kernel.model,
    com.liferay.application.list,\
    com.liferay.asset.kernel,\
    com.liferay.asset.kernel.exception, [..]
```

Note: to remove unnecessary "noise" from this example, some headers have been abbreviated ([..]) and some have been removed.

You can organize and build a module's Java code and resources however you like. You're free to use any directory structure conventions, such as those used in Maven or by your development team. And you can use any build tool, such as Gradle or Maven, to manage dependencies.

Liferay Workspace is an environment for managing module projects (and theme projects). A default Workspace provides Gradle build scripts and a Workspace created from the Liferay Project Templates Workspace archetype provides Maven build scripts for developing on Liferay. Workspace can be used from the command line or from within Liferay @ide@. Note also that Liferay @ide@ provides plugins for Gradle, Maven, and BndTools. Tooling details are covered later in this series.

Now that you're familiar with the module structure and manifest, it's time to explore how to build modules.

Building Modules with Bnd

The most common way to build modules is with a little tool called Bnd. It's an engine that, among other things, simplifies generating manifest metadata. Instead of manually creating a `MANIFEST.MF` file, developers use Bnd to generate it. Bnd can be used on its own or along with other build tools, such as Gradle or Maven. Liferay Workspace uses Bnd together with Gradle or Maven.

One of Bnd's best features is that it automatically transverses a module's code to identify external classes the module uses and adds them to the manifest's list of packages to import. Bnd also provides several OSGi-specific operations that simplify module development.

Bnd generates the manifest based on a file called `bnd.bnd` in the project root. This file's header list is similar to (but shorter than) that of the `MANIFEST.MF`. Compare the Liferay Journal Web module's `bnd.bnd` file content (simplified a bit) below to its `MANIFEST.MF` file content that was listed earlier:

```
Bundle-Name: Liferay Journal Web
Bundle-SymbolicName: com.liferay.journal.web
Bundle-Version: 1.1.2
Export-Package:\
    com.liferay.journal.web.asset,\
    com.liferay.journal.web.dynamic.data.mapping.util,\
    com.liferay.journal.web.social,\
    com.liferay.journal.web.util
```

The main difference is that the `bnd.bnd` file doesn't specify an `Import-Package` header. It's unnecessary because Bnd generates it in the `MANIFEST.MF` file automatically. It's metadata made easy!

Bnd plugins are available to use with Gradle and Maven. And since Liferay Workspace includes Bnd, developers can use Bnd from the command line and from Liferay @ide@.

Now that you're familiar with Bnd and the `Export-Package` and `Import-Package` manifest headers, let's explore how to use them to leverage dependencies.

11.2 Leveraging Dependencies

Using an OSGi manifest, a module declares the Java packages it consumes and shares. The manifest's `Import-Package` and `Export-Package` settings expose this information. As you determine whether to use a particular module, you know up-front what it offers and what it depends on. As an improvement over Java EE, OSGi takes away dependency guesswork.

This part of the tutorial explains:

- **How dependencies work**

- **How to develop modular apps using dependencies**

Let's start by learning how dependencies operate in 7.0.

How Dependencies Work

Since all of 7.0 leverages dependencies, it also demonstrates how to use them. As mentioned previously, all of what was in Liferay Portal 6 and its apps has been refactored into OSGi modules. The `portal-service` API (the main API in Liferay Portal 6) has been replaced by the `portal-kernel` module (7.0's kernel API) and many small, highly-cohesive modules that provide frameworks, utilities, apps, and more.

Not only do Liferay DXP modules depend on third-party modules but they also depend on each other. You can likewise leverage dependencies in your projects. Whether you're developing new OSGi modules or continuing to develop traditional apps, you need only set dependencies on modules whose packages you need.

Each module's manifest lists the packages the module depends on. Using a build environment such as Gradle, Maven, or Ant/Ivy, you can set dependencies on each package's module. At build time, the dependency framework verifies the entire dependency chain, downloading all newly specified modules. The same thing happens at runtime: the OSGi runtime knows exactly which modules depend on which other modules (failing fast if any dependency is unmet). Dependency management is explicit and enforced automatically upfront.

Versioning is independent for each Liferay module and its exported packages. You can use a specific package version by depending on the version of the module that exports it. And you're free to use a mix of Liferay modules in the versions you want (but remember, "With great power comes great responsibility," so unless you really know what you're doing, use the same version of each module you depend on).

For all its modules, Liferay DXP uses Semantic Versioning. It's a standard that enables API authors to communicate programmatic compatibility of a package or module automatically as it relates to dependent consumers and API implementations. If a package is programmatically (i.e., semantically) incompatible with a project, Bnd (used in Liferay Workspace) fails that project's build immediately. Developers not using Bnd can check package versions manually in each dependency module's manifest.

Semantic Versioning also gives module developers flexibility to specify a version range of packages and modules to depend on. In other words, if several versions of a package work for an app, the developer can configure the app to use any of them. What's more, Bnd automatically determines the semantically compatible range of each package a module depends on and records the range to the module's manifest.

On testing your project, you might find a new version of a dependency package has bugs or behaves differently than you'd like. No problem. You can adjust the package version range to include versions up to, but not including, the one you don't want.

Next you want to consider when to modularize existing apps and when to combine modules to create apps.

Dependencies Facilitate Modular Development

7.0's support of dependencies and semantic versioning facilitates modular development. The dependency frameworks enable you to use modules and link them together. You can use these modules throughout your organization and distribute them to others. 7.0's integration with dependency management frees you to modularize existing apps and develop apps that combine modules. It's a powerful and fun way to develop apps on Liferay DXP.

Here are some general steps to consider when modularizing an existing app:

1. **Start by putting the entire app in a single module**: This is a minimal first step that acquaints you with 7.0's module framework. You'll gain confidence as you build, deploy, and test your app in an environment of your choice, such as a Liferay Workspace, Gradle, or Maven project.

2. **Split the front-end from the back-end**: Modularizing front-end portlets and servlets and back-end implementations (e.g., Service Builder or OSGi component) is a logical next step. This enables each code area to evolve separately and allows for varying implementations.

3. **Extract non-essential features to modules**: You may have functionality or API extensions that need not be tied to an app's core codebase. They can be refactored as independent modules that implement APIs you provide. Examples might be connectors to third-party systems or support for various data export/import formats.

The principles listed above also apply to developing new modular-based apps. As you design an app, consider possible implementation variations with respect to its features, front-end, and back-end. Encapsulate the variations using APIs. Then develop the APIs and implementations as separate modules. You can wire them together using dependencies.

Liferay's Blogs application exemplifies modularization in the manner we've described:

API:

- `blogs-api` - Encapsulates the core implementation

Back-end:

- `blogs-service` - Implements `blogs-api`

Front-end:

- `blogs-web` - Provides the app's UI

Non-essential features and extensions:

- `blogs-editor-configuration` - Extends the `portal-kernel` module for extending editors
- `blogs-recent-bloggers-web` - Provides the Recent Bloggers app
- `blogs-item-selector-api` - Encapsulates the item-selector implementation
- `blogs-item-selector-web` - Renders the Blogs app's item-selector
- `blogs-layout-prototype` - Creates a Page Template showcasing blog entries

The Blogs app, like many modular apps, separates concerns into modules. In this way, front-end developers concentrate on front-end code, back-end developers concentrate on that code, and so on. These logical boundaries free developers to design, implement, and test the modules independently.

As you develop app-centered modules, you can consider bundling them with your app (e.g., as part of a Liferay Marketplace app). Including them as part of the app is convenient for the consumer. By bundling a module with an app, however, you're committing to the app's release schedule. In other words, you can't directly deploy a new version of a module for the app–you must release it as part of the app's next release.

So far, you've learned how dependencies and Semantic Versioning work. You've considered guidelines for modularizing existing apps and creating new modular apps. Now, to add to the momentum around OSGi and modularity, you'll explore OSGi Services and dependency injection using OSGi Declarative Services.

11.3 OSGi Services and Dependency Injection with Declarative Services

In 7.0, the OSGi framework registers objects as *services*. Each service offers functionality and can leverage functionality other services provide. The OSGi Services model supports a collaborative environment for objects.

Declarative Services (DS) provides a service component model on top of OSGi Services. DS service components are marked with the `@Component` annotation and implement or extend a service class. Service component can refer to and use each other's services. The Service Component Runtime (SCR) registers component services and handles binding them to other components that reference them.

Here's how the "magic" happens:

1. **Service registration:** On installing a module that contains a service component, the SCR creates a component configuration that associates the component with its specified service type and stores it in a service registry.

2. **Service reference handling:** On installing a module whose service component references another service type, the SCR searches the registry for a component configuration that matches the service type and on finding a match binds an instance of that service to the referring component.

It's publish, find, and bind at its best!

How does a developer use DS to register and bind services? Does it involve creating XML files? No, it's much easier than that. The developer uses two annotations: `@Component` and `@Reference`.

- `@Component`: Add this annotation to a class definition to make the class a component–a service provider.

- `@Reference`: Add this annotation to a field to inject it with a service that matches the field's type.

The `@Component` annotation makes the class an OSGi component. Setting a `service` property to a particular service type in the annotation, allows other components to reference the service component by the specified service type.

For example, the following class is a service component of type `SomeApi.class`.

```
@Component(
    service = SomeApi.class
)
public class Service1 implements SomeApi {

   ...
}
```

On deploying this class's module, the SCR creates a component configuration that associates the class with the service type `SomeApi`.

Specifying a service reference is easy too. Applying the `@Reference` annotation to a field marks it to be injected with a service matching the field's type.

```
@Reference
SomeApi _someApi;
```

On deploying this class's module, the SCR finds a component configuration of the class type `SomeApi` and binds the service to this referencing component class.

At build time, Bnd creates a *component description* file for each module's components automatically. The file specifies the component's services, dependencies, and activation characteristics. On module deployment, the OSGi framework reads the component description to create the component and manage its dependency on other components.

The SCR stands ready to pair service components with each other. For each referencing component, the SCR binds an instance of the targeted service to it.

As an improvement over dependency injection with Spring, OSGi Declarative Services supports dynamic dependency injection. Developers can create and publish service components for other classes to use. Developers can update the components and even publish alternative component implementations for a service. This kind of dynamism is a powerful part of 7.0.

11.4 Dynamic Deployment

In OSGi, all components, Java classes, resources, and descriptors are deployed via modules. The `MANIFEST.MF` file describes the module's physical characteristics, such as the packages it exports and imports. The module's component description files specify its functional characteristics (i.e., the services its components offer and consume). Also modules and their components have their own lifecycles and administrative APIs. Declarative Services and shell tools give you fine-grained control over module and component deployment.

Since a module's contents depend on its activation, consider the activation steps:

1. *Installation*: Copying the module JAR into Liferay DXP's `deploy` folder installs the module to the OSGi framework, marking the module `INSTALLED`.

2. *Resolution*: Once all the module's requirements are met (e.g., all packages it imports are available), the framework publishes the module's exported packages and marks it `RESOLVED`.

3. *Activation*: Modules are activated *eagerly* by default. That is, they're started in the framework and marked `ACTIVE` on resolution. An active module's components are enabled. If a module specifies a `lazy` activation policy, as shown in the manifest header below, it's activated only after another module requests one of its classes.

   ```
   Bundle-ActivationPolicy: lazy
   ```

The figure below illustrates the module lifecycle.

The Apache Felix Gogo Shell lets developers manage the module lifecycle. They can install/uninstall modules and start/stop them. Developers can update a module and notify dependent modules to use the update. Liferay's tools, including Liferay @ide@, Liferay Workspace, and Blade CLI offer similar shell commands that use the OSGi Admin API.

On activating a module, its components are enabled. But only *activated* components can be used. Component activation requires all its referenced services be satisfied. That is, all services it references must be registered. The highest ranked service that matches a reference is bound to the component. When the container finds and binds all the services the component references, it registers the component. It's now ready for activation.

Components can use *delayed* (default) or *immediate* activation policies. To specify immediate activation, the developer adds the attribute `immediate=true` to the `@Component` annotation.

```
@Component(
    immediate = true,
    ...)
```

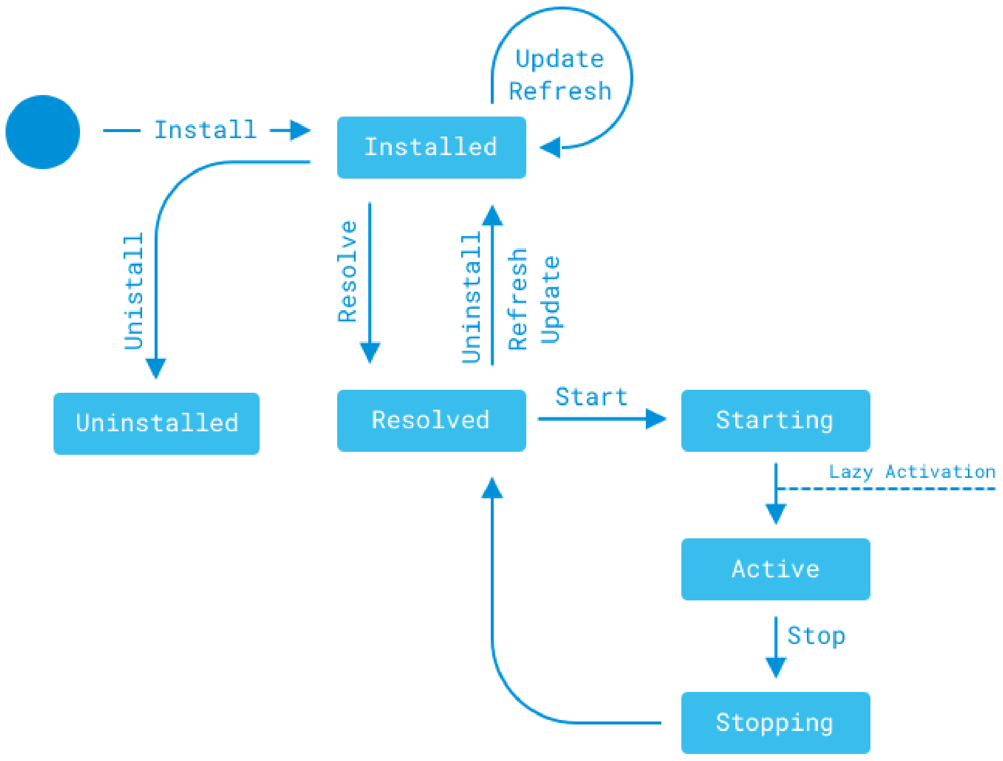


Figure 11.1: This state diagram illustrates the module lifecycle.

Unless immediate activation is specified, the component's activation is delayed. That is, the component's object is created and its classes are loaded once the component is requested. In this way delayed activation can improve startup times and conserve resources.

Gogo Shell's Service Component Runtime commands let you manage components:

- `scr:list [bundleID]`: Lists the module's (bundle's) components.
- `scr:info [componentID|fullClassName]`: Describes the component, including its status and the services it provides.
- `scr:enable [componentID|fullClassName]`: Enables the component.
- `scr:disable [componentID|fullClassName]`: Disables the component. It's disabled on the server (or current server node in a cluster) until the server is restarted.

Service references are static and reluctant by default. That is, an injected service remains bound to the referencing component until the service is disabled. Alternatively, developers can specify *greedy* service policies for references. Every time a higher ranked matching service is registered, the framework unbinds the lower ranked service from the component and binds the new service in its place automatically. Here's a `@Reference` annotation that uses a greedy policy:

```
@Reference(policyOption = ReferencePolicyOption.GREEDY)
```

Declarative Services annotations let you specify component activation and service policies. Gogo Shell commands let you control modules and components. Next, you'll create and deploy a module and component to Liferay DXP.

11.5 Example: Building an OSGi Module

The previous sections explained some of the most important concepts for Liferay Portal 6 developers to understand about OSGi and modularity. Now it's time to put this knowledge to practice by creating and deploying a module.

The module includes a Java class that implements an OSGi service using Declarative Services. The project uses Gradle and Bnd, and can be built and deployed from within a Liferay Workspace.

Here's the module project's anatomy:

- `bnd.bnd`
- `build.gradle`
- `src/main/java/com/liferay/docs/service/MyService.java`

On building the module JAR, Bnd generates the module manifest automatically.

Here's the Java class:

```
package com.liferay.docs.service;

import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;

@Component(
    immediate = true,
    service = MyService.class
)
public class MyService {

    @Activate
    void activate() throws Exception {

        System.out.println("Activating " + this.getDescription());
    }

    public String getDescription() {

        return this.getClass().getSimpleName();
    }

}
```

It contains these methods:

- `getDescription` - returns the class's name
- `activate` - prints the console message *Activating MyService*. The `@Activate` annotation signals the OSGi runtime environment to invoke this method on component activation.

The `@Component` annotation defines the class as an OSGi service component. The following properties specify its details:

- `service=MyService.class` - designates the component to be a service component for registering under the type `MyService`. In this example, the class implements a service of itself. Note, service components typically implement services for interface classes.

- `immediate=true` - signals the Service Component Runtime to activate the component immediately after the component's dependencies are resolved.

The `bnd.bnd` file is next:

```
Bundle-SymbolicName: my.service.project
Bundle-Version: 1.0.0
```

The `Bundle-SymbolicName` is the arbitrary name for the module. The module's version value `1.0.0` is appropriate.

Bnd generates the module's OSGi manifest to the file `META-INF/MANIFEST.MF` in the module's JAR. In this project, the JAR is created in the `build/libs` folder.

The last file to examine is the Gradle build file `build.gradle`:

```
dependencies {
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

Since the `MyService` class uses the `@Component` annotation, the project depends on the OSGi service component annotations module. The build script is so simple because Liferay Workspace module projects leverage the Workspace's Gradle build infrastructure.

Although this module project was created in a Liferay Workspace, it can easily be modified to use in other build environments. To keep the focus on what's most important, it was created in a Liferay Workspace.

Place the project files in a folder under the `modules` folder (e.g., `[Liferay_Workspace]/modules/my.service.project`).

To build the module JAR and deploy it to Liferay DXP, execute the `deploy` Gradle task:

```
../../gradlew deploy
```

Note: If Blade is installed (recommended), Gradle can be executed by entering `blade gw` followed by a task name (e.g., `blade gw deploy`). For details on Blade commands, see Blade CLI.

On deploying the module, the following message is printed to the server console:

`Activating MyService`

Congratulations! You've successfully built and deployed an OSGi module to Liferay DXP.

11.6 Learning More about OSGi

There is much more to learn about developing apps using OSGi. Several resources are listed below and many more abound. To make the best of your time, however, avoid OSGi service articles that explain techniques that are older and more complicated than Declarative Services.

Developers new to OSGi should check out these resources:

- Introduction to Liferay Development: For using OSGi to develop on Liferay DXP.

- OSGi enRoute is a site the OSGi Alliance provides to the OSGi community. Its Tutorials provide hands-on experience with OSGi modules and Declarative Services.

- OSGi Alliance's Developer section explains OSGi's architecture and modularity.

Developers ready to dive deep into OSGi should read the OSGi specifications. They're well-written and provide comprehensive details on all that OSGi offers. *The OSGi Alliance OSGi Compendium: Release 6* specifies the following services that 7.0 leverages extensively.

- *Declarative Services Specification*
- *Configuration Admin Service Specification*: For modifying deployed bundles. Since Configuration Admin services are already integrated with Declarative Services, however, Liferay developers need not use the low-level API.
- *Metatype Service Specification*: For describing attribute types as metadata.

Chapter 12

Improved Developer Tooling: Liferay Workspace, Maven Plugins and More

Creating applications is fun when you have the right tools. Here are some key ingredients:

- Rich templates for stubbing out projects
- Extensible build environments that offer state-of-the-art plugins
- Deployment and runtime management tools
- Application upgrade automation

Liferay Workspace (Workspace) boils over with all these things! It's a Gradle-based development environment that integrates with Liferay @ide@ and can be used in conjunction with other IDEs, such as a "vanilla" Eclipse, IntelliJ, and NetBeans. You can extend Workspace's Gradle environment with community-developed (or home-grown) plugins for testing, code coverage analysis, and more.

Workspace comes with Blade CLI: a command line tool for creating and deploying modules, managing the runtime environment, and more. It provides all kinds of module templates, to create modules for developing in any Gradle environment.

Liferay's tools also streamline the application upgrade process. Liferay @ide@'s Upgrade Planner adapts traditional plugins to 7.0 APIs. Liferay's Liferay Theme Generator migrates themes and layout templates to the new NodeJS-based environment and adapts them to 7.0.

Liferay DXP offers you more with Maven too. The archetype Liferay Project Templates Workspace lets you develop in Liferay Workspace using Maven. 7.0's' lean artifacts and new project archetypes and Maven plugins make Liferay DXP development with Maven easier than ever.

Here are the tooling improvement topics:

- Moving from the Plugins SDK to Liferay Workspace
- Developing modules with Liferay Workspace
- What's new in Liferay DXP for Maven Users
- Using other build systems and IDEs

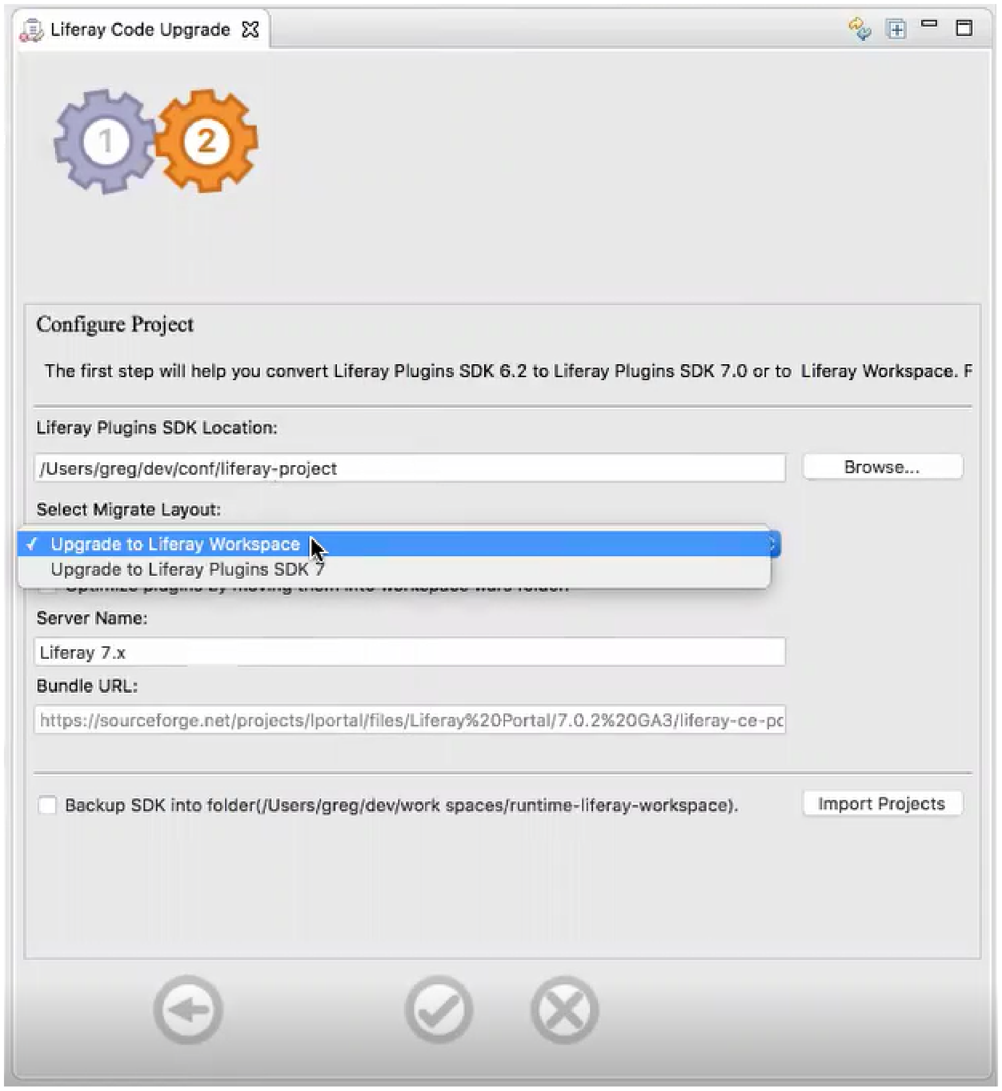
12.1 From the Plugins SDK to Liferay Workspace

The Liferay Plugins SDK is deprecated as of 7.0. You can continue developing on it, but should plan to eventually move to a new environment. Liferay Workspace succeeds the Plugins SDK as Liferay's opinionated development environment. You should use it if you're not using an alternative build system like Gradle or Maven.

Here are Workspace's key features:

- Module and component templates
- Sample projects
- Portal server configurations
- Folder structure flexibility
- Commands to migrate plugins, install Liferay DXP bundles, and start/stop Portal instances

The plugin upgrade tutorials later in this series show how Liferay 7.0 automatically adapts existing plugins to @product-ver@. There's also a tutorial that demonstrates how you can optionally migrate traditional plugins to Workspace.



Configure Project

The first step will help you convert Liferay Plugins SDK 6.2 to Liferay Plugins SDK 7.0 or to Liferay Workspace. F

Liferay Plugins SDK Location:
/Users/greg/dev/conf/liferay-project

Select Migrate Layout:
✓ Upgrade to Liferay Workspace
Upgrade to Liferay Plugins SDK 7

Server Name:
Liferay 7.x

Bundle URL:
https://sourceforge.net/projects/lportal/files/Liferay%20Portal/7.0.2%20GA3/liferay-ce-pc

Backup SDK into folder(/Users/greg/dev/work spaces/runtime-liferay-workspace).

Figure 12.1: Liferay @ide@'s Upgrade Planner automates many aspects of the plugin upgrade process.

Here's an example Workspace folder structure:

Here's the Workspace anatomy:

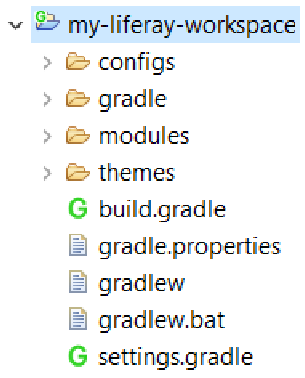


Figure 12.2: Liferay Workspace aggregates projects to use the same server configurations and Gradle build environment.

- `bundles/` (generated) → default folder for Liferay DXP bundles
- `configs/` → holds Portal server configurations
- `gradle/` → holds the Gradle wrapper files
- `modules/` → holds module projects
- `plugins-sdk/` (generated) → holds plugins from previous releases
- `themes/` → holds NodeJS-based theme projects
- `wars/` (generated) → holds traditional web application projects
- `build.gradle` → common Gradle build file
- `gradle.properties` → specifies the Portal server configuration and project locations
- `gradlew / gradlew.bat` → executes the Gradle command wrapper
- `pom.xml` (only in Workspaces generated by Maven) → common Maven build file
- `settings.gradle` → applies plugins to the Workspace and configures its dependencies

Workspace module, theme, and war projects use the same Portal server configurations. Developers can create configurations for module development, user acceptance testing, production, and more.

Each subfolder under `configs` holds a Portal server configuration defined by its `portal-ext.properties` file. Gradle property `liferay.workspace.environment` in Workspace's `gradle.properties` file specifies the configuration to use.

Other Gradle properties let you set root locations for the Liferay DXP bundle, modules, themes, and a Plugins SDK.

Workspace Folder Structure Properties

Property	Description
`liferay.workspace.environment`	Name of a `configs` subfolder holding the Portal server configuration to use
`liferay.workspace.home.dir`	Liferay DXP bundle root folder
`liferay.workspace.modules.dir`	Module projects root folder
`liferay.workspace.plugins.sdk.dir`	Plugins SDK root folder
`liferay.workspace.themes.dir`	Theme projects root folder

Workspace has Gradle tasks equivalent to the Plugins SDK Ant targets.

Plugins SDK to Workspace Task Map

Plugins SDK Ant Target	Workspace Gradle Task	Task Description
`build-css`	`buildCSS`	Builds CSS files
`build-lang`	`buildLang`	Translates language keys using Language Builder
`build-service`	`buildService`	Runs Service Builder
`clean`	`clean`	Deletes all build outputs
`compile`	`classes`	Compiles classes
`deploy`	`deploy` (or `blade deploy`)	Installs the current module to Liferay DXP's module framework
`jar`	`jar`	Compiles the project and packages it as a JAR file
`war`	`assemble`	Assembles project output

Other Workspace Gradle tasks provide additional functionality.

Workspace Gradle Task	Task Description
`buildSoy`	Compiles Closure Templates in JavaScript functions
`components`	Lists the project's components
`dependencies`	Lists the project's declared dependencies
`initBundle`	Downloads and installs a Liferay DXP bundle
`model`	Lists the project's configuration model
`transpileJS`	Transpiles the project's JavaScript files

Next, learn how Workspace facilitates module development.

12.2 Developing Modules with Liferay Workspace

Workspace is a great Liferay module development environment because of these features:

- Templates that bootstrap module creation
- Gradle and Maven build systems for managing dependencies and assembling modules
- Module deployment and runtime management capabilities

Blade CLI (Blade), which is a part of Workspace, has over twenty templates for Gradle-based module projects–and more are being added. The templates stub out classes and resource files for you to fill in with business logic and key information.

Here are some of the template's names:

- Activator
- API

- Content Targeting Report
- Content Targeting Rule
- Content Targeting Tracking Action
- Control Menu Entry
- MVC Portlet
- Panel App
- Portlet
- Portlet Configuration Icon
- Portlet Provider
- Portlet Toolbar Contributor
- Service
- Service Builder
- Service Wrapper
- Simulation Panel Entry
- Template Context Contributor

Blade creates modules based on these templates.

For example, the following Blade command creates a Liferay MVC Portlet module called `my-module`:

```
blade create -t mvc-portlet -p com.liferay.docs.mymodule -c MyMvcPortlet my-module
```

Liferay @ide@'s module project wizard creates Workspace modules from the templates too.

Liferay @ide@'s component wizard facilitates creating component classes for portlets, service wrappers, Struts actions, and more.

Building and deploying modules in a Workspace is a snap using Liferay @ide@ and Blade. Workspace uses BndTools to generate each module's OSGi headers in a `META-INF/MANIFEST.MF` file. Workspace deploy modules to the OSGi container using Felix File Install commands.

Liferay @ide@ lets you deploy modules by dragging them onto your Portal server.

In a terminal, you can deploy modules using Blade's `deploy` command. For example, the following command deploys the current module and "watches" for module changes to redeploy automatically.

```
blade deploy -w
```

To learn more about Workspace and using it in Liferay @ide@, see these tutorials:

- Workspace
- Blade CLI
- Liferay @ide@

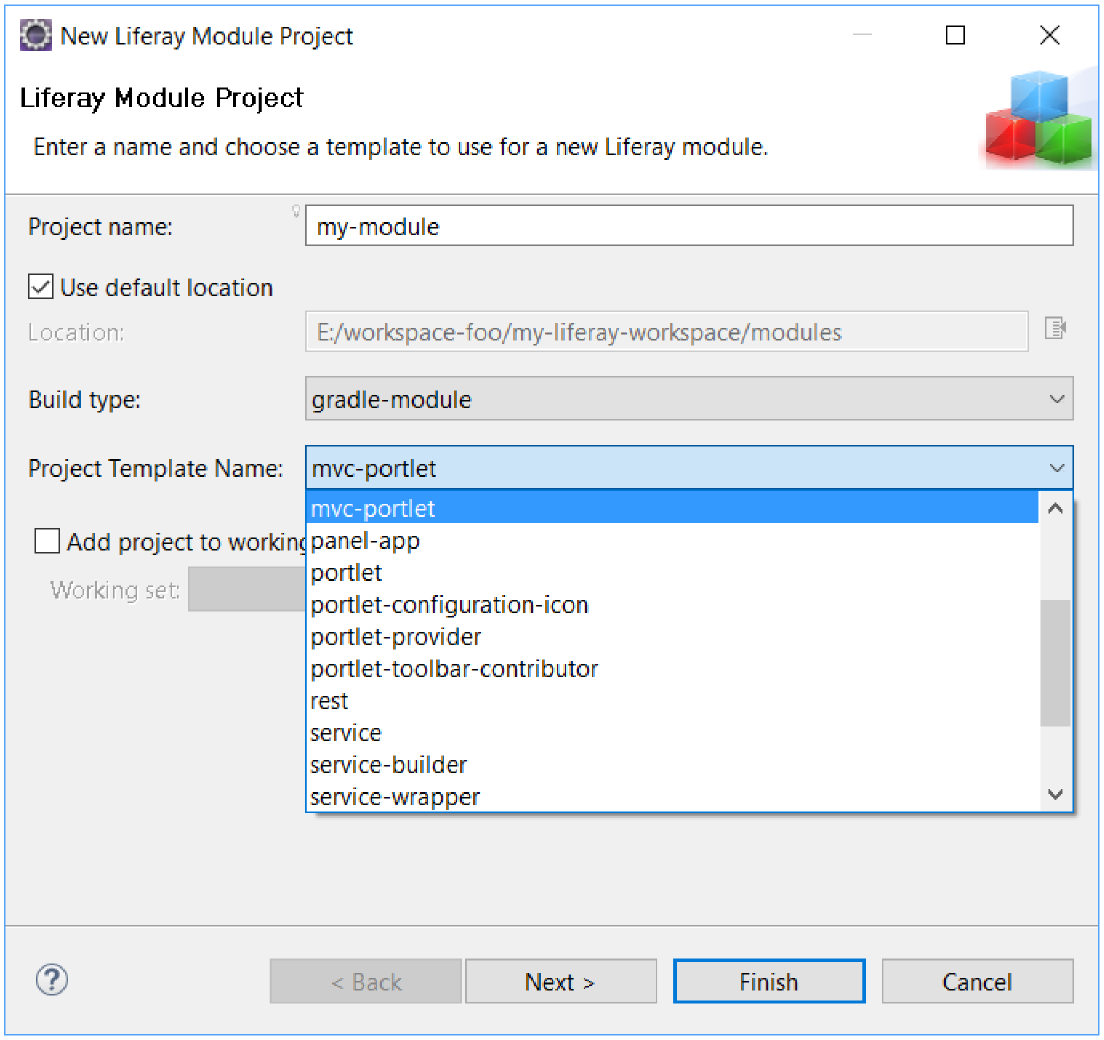
And make sure to check out the tutorial Starting Module Development.

Next, you'll learn new features for developing on Liferay DXP using Maven.

12.3 What's New in 7.0 for Maven Users

7.0 fully supports Maven development and offers several new and improved features:

- Liferay Workspace for Maven
- New archetypes
- New Maven plugins


New Liferay Module Project

Liferay Module Project

Enter a name and choose a template to use for a new Liferay module.

Project name: my-module

[✓] Use default location

Location: E:/workspace-foo/my-liferay-workspace/modules

Build type: gradle-module

Project Template Name: mvc-portlet

mvc-portlet
panel-app
portlet
portlet-configuration-icon
portlet-provider
portlet-toolbar-contributor
rest
service
service-builder
service-wrapper

[ ] Add project to working set

Working set:

Figure 12.3: Liferay @ide@ lets developers select templates to stub out modules.

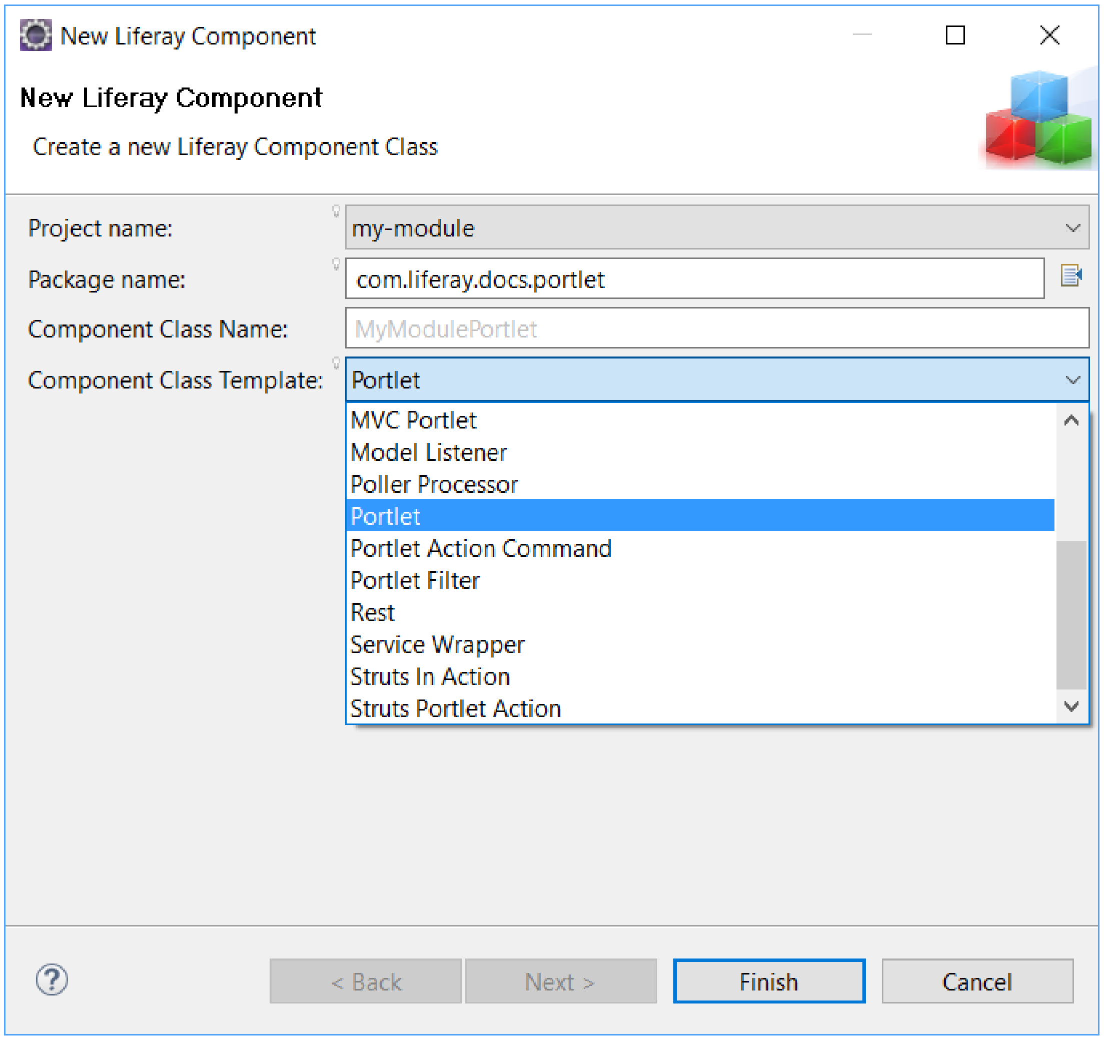
- More granular dependency management

The new archetype Liferay Project Templates Workspace generates a Liferay Workspace that includes a POM file for developing in Workspace using Maven. You can develop modules and themes in the Workspace subfolders.

7.0 provides many new Maven archetypes for various Liferay module projects. There are over twenty-five Maven archetypes for 7.0, and more are in development. Here are some popular ones:

- Portlets
- Themes
- Configuration Icons
- Menu Buttons
- Service Builder

Liferay's Maven archetypes cover many different Liferay frameworks and service types. These make Maven a first-class tool for creating Liferay modules and themes. Visit the Generating New Projects Using Archetypes tutorial to learn more about Liferay's Maven archetypes and how to use them.


New Liferay Component

New Liferay Component

Create a new Liferay Component Class

Project name: my-module

Package name: com.liferay.docs.portlet

Component Class Name: MyModulePortlet

Component Class Template: Portlet

- MVC Portlet
- Model Listener
- Poller Processor
- Portlet
- Portlet Action Command
- Portlet Filter
- Rest
- Service Wrapper
- Struts In Action
- Struts Portlet Action

Figure 12.4: Liferay @ide@'s component wizard facilitates creating component classes.

Liferay also provides several new and updated Maven plugins that simplify the build process. The following plugins build style sheets, services, and themes respectively:

- CSS Builder
- Service Builder
- Theme Builder

7.0's modularity provides a more granular dependency management experience. You no longer need to depend on `portal-impl` or `portal-service` (now `portal-kernel`) for everything. For example, to use Liferay's Wiki framework, you need only depend on the Wiki module. You set dependencies on concise modules that provide the functionality you want without inheriting extra baggage.

Liferay's new Workspace environment, Maven archetypes, Maven plugins, and streamlined modules make developing on Liferay DXP easier than ever. To learn more, see the Maven tutorials.

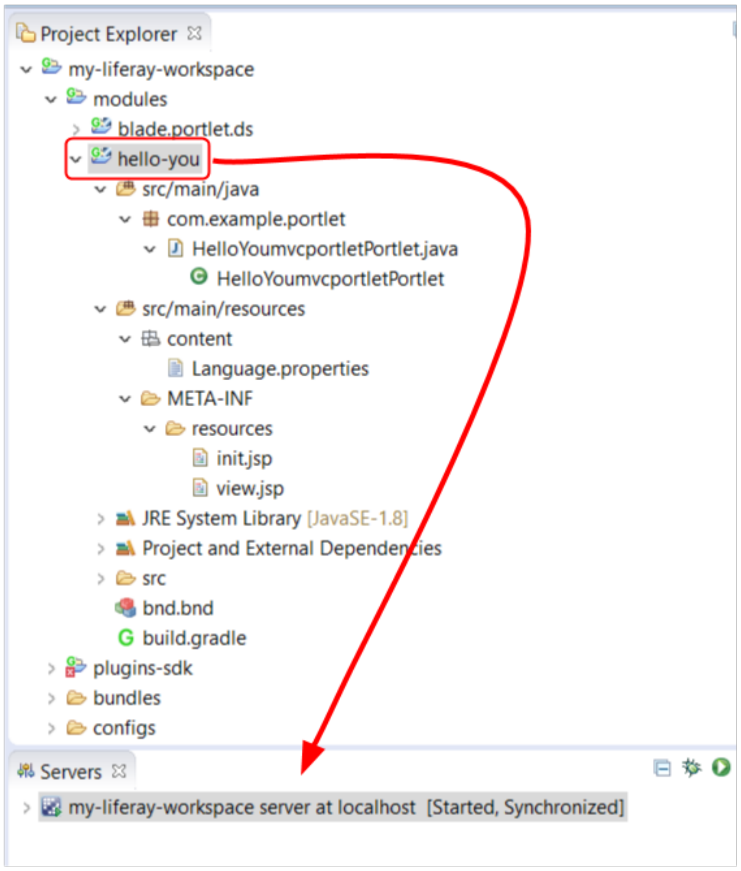


Figure 12.5: Liferay @ide@ lets you deploy modules using drag-and-drop.

12.4 Using Other Build Systems and IDEs

Liferay DXP is tool agnostic–you can use whatever tools you like to develop on it. You can use any IDE and even use Gradle, Bnd, or BndTools if you don't want to use Workspace. The drawback is you lose the Liferay-specific project templates that you get with Blade and Workspace.

Blade lets you create modules to develop anywhere, not only in Liferay Workspace.

Here are some new Gradle features Liferay provides that are independent of Workspace:

- Liferay's Gradle plugins
- Buildship plugins in Liferay @ide@
- Liferay @ide@'s new Gradle views for developing modules and working with Gradle tasks

Liferay has worked hard to make Liferay DXP IDE-agnostic. There are Liferay module developers who use IntelliJ and some enjoy using NetBeans.

Finally, you can copy and modify Liferay sample projects to serve as templates in place of the Blade templates. They're available for these build systems:

- Maven

- Gradle
- Liferay's Gradle environment based on the `com.liferay.plugin` plugin

Liferay's approach to tooling has vastly improved for 7.0. Our tools help you upgrade to 7.0, continue developing traditional plugins the way you have been, and migrate to optimal development environments. Liferay Workspace and the improved Maven support facilitate module development. And developing on Liferay DXP using other tools is easier than ever. Your tool options are wide open.

Chapter 13

Planning Plugin Upgrades and Optimizations

If you've explored 7.0's features and possibly created new portlet modules themes with Liferay's new tooling and techniques, you may be wondering how you'd upgrade existing plugins. The great thing is that Liferay has automated much of the upgrade process. In addition, you can continue developing plugins in traditional ways and adopt new development tooling and techniques when you're ready.

This tutorial guides you through phases of *upgrading* plugins and optionally *optimizing* them.

Upgrade: A process for deploying an existing plugin on 7.0 with minimal changes.

Optimization: An optional but recommended process for modifying a plugin or migrating it to a new environment to improve the plugin or facilitate developing it.

Importantly, you should *upgrade* a plugin before applying any optimizations to it.

The good news is that upgrading plugins to 7.0 is straightforward. For Plugins SDK and Maven projects, Liferay @ide@'s Upgrade Planner automates much of the process. In addition, the upgrade tutorials demonstrate any remaining upgrade steps.

You can deploy plugins to 7.0 as you have for previous releases (e.g., `ant clean deploy`). Since everything in 7.0 runs as OSGi modules, however, you might wonder how traditional WAR-style plugins can run on it. The answer: Liferay's Plugin Compatibility Layer.

The Plugin Compatibility Layer converts standard WARs to Web Application Bundles (WABs). WABs are full-fledged OSGi modules. The Plugin Compatibility Layer's WAB Generator supports deploying traditional plugins and web applications that contain Servlets, JSPs, and other Java web technologies without making any OSGi specific changes to them.

Note, you can still use an application server's mechanisms to deploy regular web applications along with Liferay DXP, without using the Plugin Compatibility Layer.

After upgrading your plugins you can consider optimizations such as these:

- Migrating plugins to Gradle or Maven to leverage their development commands and rich Liferay plugin templates.
- Migrating themes to the Liferay Theme Generator to add Themelets (new) and to leverage Node.js, Yeoman, and Gulp.
- Converting plugins to modules to leverage Declarative Services, extendability, and more modularity benefits.
- Using the Lexicon, to apply a clean consistent application user experience.

See the optimization tutorials for more options and details.

You *can* continue using the Plugins SDK to develop plugins. But the Plugins SDK is deprecated as of 7.0. In light of the deprecation, you should consider migrating plugins from the Plugins SDK to one of the new environments:

- Liferay Workspace is a Gradle environment that supports developing modules and traditional plugins. Blade's `migrateWar command` moves Plugins SDK portlets to Liferay Workspace (Workspace) in a snap.
- Liferay's Maven plugins and archetypes support developing modules and traditional plugins. There's also a Liferay Workspace archetype for generating a Workspace that uses Maven.

Liferay @ide@ supports developing in Workspaces using Gradle or Maven.

In short, there's plenty of time to move plugins out of the Plugins SDK, but you should at least plan for migrating to a new environment that works best for you.

Speaking of planning, properly planned upgrades and optimizations reduce the time and effort they take. To help guide you through the upgrade and optimization tutorials, you get these things:

- Upgrade and optimization phase descriptions
- Upgrade and optimization paths

13.1 Upgrade and Optimization Phases

Follow these upgrade and optimization phases:

1. Read the applicable upgrade tutorials for your plugin. Examine the upgrade and optimization paths.
2. Upgrade the plugin, making only the minimal changes necessary for it to work on 7.0.
3. (Optional) Identify and apply only the most beneficial optimizations for your plugin.
4. (Optional) Apply additional optimizations as desired.

13.2 Upgrade and Optimization Paths

The following tables provide upgrade and optimization paths for 6.2 plugins and features.

Plugin Upgrade and Optimization Paths

Plugin	Upgrade path	Optimizations (optional)
Ext	Customization with Ext Plugins	None
Hook - Language files	- Upgrading Core Language Key Hooks- Upgrading Portlet Language Key Hooks	Same
Hook - Override a Liferay DXP Core JSP	Upgrading Core JSP Hooks	Same
Hook - Override an app's JSP	Upgrading App JSP Hooks	Same

Plugin	Upgrade path	Optimizations (optional)
Hook - Event Actions (Portal/Session/Servlet Service/Shutdown/Startup)	Upgrading Portal Property and Event Action Hooks	None
Hook - Model Listeners	Upgrading Model Listener Hooks	Same
Hook - Portal Properties	Upgrading Portal Property and Event Action Hooks	Same
Hook - Properties	- If the property is now a System Setting, edit it there and/or use a `.config` file- If the property is in the liferay-hook.xml's DTD, then adapt code to API and resolve dependencies	None
Hook - Service Wrappers	Upgrading Service Wrappers	None
Hook - Servlet Filter	Upgrading Servlet Filter Hooks	None
Hook - Struts actions	- StrutsAction → StrutsActionWrapper - processAction → MVCActionCommand - render → MVCRender-Command - serveResource → MVCResourceCom-mand	Same
Layout Template	1. Adapt code to API2. Resolve dependencies3. Update Layout Template	- Migrate to Liferay Theme Generator (Node.js/Gulp/Yeoman)
Portlet - GenericPortlet	Upgrading a GenericPortlet	- Migrate to Workspace/Gradle- Apply Lexicon- Convert to OSGi modules
Portlet - Liferay MVC	Upgrading a Liferay MVC Portlet	- Migrate to Workspace/Gradle- Apply Lexicon- Convert to OSGi modules
Portlet - JSF	Upgrading a Liferay JSF Portlet	None
Portlet - Servlet/JSP	Upgrading a Servlet-based Portlet	None
Portlet - Spring MVC	Upgrading a Spring MVC Portlet	None

Plugin	Upgrade path	Optimizations (optional)
Portlet - Struts 1	Upgrading a Struts Portlet	None
Theme	1. Adapt code to API2. Resolve dependencies3. Upgrade Theme	- Migrate to Liferay Theme Generator (Node.js/Gulp/Yeoman)- Use Themelets
Web plugin	1. Adapt code to API2. Resolve dependencies	Convert to OSGi module, e.g., `portlet-x-web`

Feature Upgrade and Optimization Paths

Feature	Upgrade path	Optimizations (optional)
JNDI data source	Use Liferay DXP's classloader to access the app server's JNDI API	None
Resources Importer	Update the Resources Importer	None
Services - Invoke a service from Liferay DXP Core or another portlet or module	Implement a Service Tracker	Invoke Liferay services from a module
Services - Module dependency	Copy `x-service.jar` to `WEB-INF/lib`	- Migrate to Gradle/Maven and add dependency on the OSGi service
Services - Service Builder	Upgrading Portlets that use Service Builder	Convert to OSGi modules, e.g., `x-api` and `x-service`
Services - Web services	1. Adapt code to API2. Resolve dependencies	Use a Service Builder service with JAX-RS with a REST service in front
Template - FreeMarker	- Adapt code to API- Adapt Theme templates	None
Template - Velocity (deprecated)	Adapt code to API	Convert to FreeMarker

Now you have a game plan and a cheat sheet for upgrading and optimizing plugins with confidence.

13.3 Upgrading Plugins to 7.0

Upgrading to 7.0 involves migrating your installation and code (your custom apps) to the new version. You'll learn how to upgrade your code in this section.

The first upgrade process step is to adapt your existing plugin's code to 7.0's APIs. The great news is that Liferay's Upgrade Planner makes this easier than ever. It identifies Liferay API changes affecting your code, explains the API changes, and offers resolution steps. And the tool offers auto-correction where it can.

You might be tempted to optimize your existing plugins right away to benefit from the new things Liferay DXP offers, but you shouldn't. It's much better to upgrade your plugins according to these tutorials. In this way, you'll get your plugins running in Liferay as fast as possible, and at the same time you'll have prepared the plugins for the optimizations you can implement later.

These tutorials assume you're using the Liferay Upgrade Planner. To follow along with this section, install the planner and step through the upgrade instructions.

For convenience, this tutorial section also references documentation and outlined steps to aid those opting to upgrade their code manually.

Here are the code upgrade steps:

1. Upgrade Your Development Environment

   Legacy project environments should be upgraded to the latest version of Liferay Workspace to ensure you leverage all available features.

   1. Set Up Liferay Workspace
      A Liferay Workspace is a generated environment that is built to hold and manage your Liferay projects. Create/import a workspace to get started.
      1. Create New Liferay Workspace
         If you don't have an existing 7.x Liferay Workspace, you must create one. Skip to the next step if you have an existing workspace.
      2. Import Existing Liferay Workspace
         Import an existing Liferay Workspace. If you don't have one, revisit the previous step.
   2. Configure Liferay Workspace Settings
      Set the Liferay DXP version in workspace's configuration you intend to upgrade to.
      1. Configure Bundle URL
         Configure your bundle URL that the Liferay DXP bundle is downloaded from.
      2. Configure Target Platform Version
         Configure your Target Platform version, which provides the specific artifacts associated with a Liferay DXP release.
      3. Initialize Server Bundle
         Download the Liferay DXP bundle you're upgrading to.

2. Migrate Plugins SDK Projects

   Copy your Plugins SDK projects into workspace and convert them to Gradle/Maven projects.

   1. Import Existing Plugins SDK Projects
      Import your existing Plugins SDK projects.
   2. Migrate Existing Plugins to Workspace
      Migrate your existing plugins to workspace. This involves moving the plugin to workspace and converting it to the workspace's build environment.

3. Upgrade Build Dependencies

   Optimize your workspace's build environment for the most efficient code upgrade experience.

   1. Update Repository URL
      Update your repository URL to Liferay's frequently updated CDN repository.

    2. Update Workspace Plugin Version
       Update your Workspace plugin version to leverage the latest features of Liferay Workspace.
    3. Remove Dependency Versions
       Remove the project's dependency versions since it's leveraging target platform.

4. Fix Upgrade Problems

   Fix common upgrade problems dealing with your project's dependencies and breaking changes.

    1. Auto-Correct Upgrade Problems
       Auto-correct straightforward upgrade problems.
    2. Find Upgrade Problems
       Find upgrade problems. These are problems that cannot be auto-corrected; you can update them manually according to the breaking changes documentation.
    3. Resolve Upgrade Problems
       Mark upgrade problems as resolved after addressing them.
    4. Remove Problem Markers
       After fixing your upgrade problems, remove the problem markers.
    5. Resolving a Plugin's Dependencies
    6. Resolving Breaking Changes

5. Upgrade Customization Plugins

   Upgrade your customization plugins so they're deployable to 7.0.

    1. Upgrade Customization Modules
    2. Upgrade Core JSP Hooks
    3. Upgrade Portlet JSP Hooks
    4. Upgrade Service Wrapper Hooks
    5. Upgrade Core Language Key Hooks
    6. Upgrade Portlet Language Key Hooks
    7. Upgrade Model Listener Hooks
    8. Upgrade Event Action Hooks
    9. Upgrade Servlet Filter Hooks
    10. Upgrade Portal Properties Hooks
    11. Upgrade Struts Action Hooks

6. Upgrade Themes

   Upgrade your themes so they're deployable to 7.0.

    1. Upgrading Your Theme from Liferay Portal 6.1 to 7.0
    2. Upgrading Your Theme from Liferay Portal 6.2 to 7.0

7. Upgrade Layout Templates

   Upgrade your layout templates so they're deployable to 7.0.

8. Upgrade Frameworks & Features

   1. Upgrade JNDI Data Source Usage
      Use Liferay DXP's class loader to access the app server's JNDI API.
   2. Upgrade Service Builder Service Invocation
      For Service Builder logic remaining in a WAR, you must implement a service tracker to call services. For logic divided into OSGi modules, you can leverage Declarative Services.
   3. Upgrade Service Builder
      Adapt your app to account for Service Builder-specific changes.
   4. Migrate Off of Velocity Templates
      Velocity template usage is deprecated for 7.0. You should convert your template to FreeMarker.

9. Upgrade Portlets

   Upgrade your portlets so they're deployable to 7.0.

   1. Upgrade Generic Portlets
   2. Upgrade Liferay MVC Portlets
   3. Upgrade JSF Portlets
   4. Upgrade Servlet-based Portlets
   5. Upgrading Spring MVC Portlets
   6. Upgrade Struts Portlets

10. Upgrade Web Plugins

    Upgrade web plugins previously stored in the `webs` folder of your legacy Plugins SDK.

11. Upgrade Ext Plugins

    Attempt to leverage an extension point instead of upgrading your Ext plugin. If an Ext plugin is necessary, you must review all changes between the previous Liferay Portal instance you were using and 7.0, and then manually modify your Ext plugin to merge your changes with Liferay DXP's.

Once you've finished the code upgrade steps, your custom apps will be compatible with 7.0!

13.4 Upgrading Your Development Environment

A Liferay Workspace is a generated environment that is built to hold and manage your Liferay projects. It is intended to aid in the management of Liferay projects by providing various build scripts and configured properties.

Liferay Workspace is the recommended environment for your code migration; therefore, it will be the assumed development environment in this section.

Continue on to set up a workspace.

Setting Up Liferay Workspace

You must set up your workspace development environment before you begin upgrading your custom apps. If you don't have an existing workspace, follow the step for creating one. If you have an existing workspace, follow the step on importing it into the Upgrade Planner.

Creating New Liferay Workspace

Initiating this step in the Upgrade Planner loads the Liferay Workspace Project wizard.

1. Give your new workspace a name.
2. Choose the build type (Gradle or Maven) you prefer for your workspace environment and future Liferay projects.
3. Click Finish.

You now have a new Liferay Workspace available in the Upgrade Planner!

For more information on creating a Liferay Workspace outside the planner, see the Creating a Liferay Workspace section.

Importing Existing Liferay Workspace

If you already have an existing 7.x Liferay Workspace, you should import it into the planner. Once you initiate this step, you're given a File Explorer/Manager to select your existing workspace. After selecting it, the workspace is imported into the Project Explorer.

For more information importing a workspace into your IDE, see this article.

Configuring Liferay Workspace Settings

You must configure your workspace with the Liferay DXP version you intend to upgrade to. You should verify the workspace's

- Bundle URL
- Target Platform Version

The bundle URL version and target platform version must match.

Visit these steps to begin.

Configuring Bundle URL

The bundle URL points to the Liferay DXP version you want workspace to download. When initiating this step, your workspace's Bundle URL property is updated to the latest release of 7.0.

For more information on configuring a workspace's bundle URL, see the Adding a Liferay Bundle to Liferay Workspace article.

Configuring Target Platform Version

The target platform is the Liferay DXP version you intend to develop for in your workspace. This is used to specify dependencies associated with a specific release. You set the target platform, define your dependencies, and workspace automatically assigns the dependency versions based on the set Liferay DXP version. When initiating this step, your workspace's Target Platform property is updated to the latest release of 7.0.

For more information on this, see the Managing the Target Platform article.

Initializing Server Bundle

Once your workspace is configured for the Liferay DXP version you're upgrading to, you can initialize the server bundle. This involves downloading the bundle and extracting it into its folder (e.g., `bundles`). If you have an existing workspace already equipped with an older Liferay bundle, this deletes the old bundle and initializes the new one.

If you're upgrading your code manually and working in Dev Studio, you can do this by right-clicking the workspace project and selecting *Liferay → Initialize Server Bundle*. See the Installing a Server in IntelliJ article if you use IntelliJ instead. Visit the Managing Your Liferay Server with Blade CLI article for information on how to do this via the command line.

13.5 Migrating Plugins SDK Projects to Liferay Workspace

The Plugins SDK was deprecated for Liferay DXP 7.0. Therefore, to upgrade your custom apps to 7.0, it's recommended to migrate them to a new environment. Liferay Workspace is the recommended environment for your code migration and will be the assumed choice in this section.

There are two steps you must follow to migrate your custom code to workspace:

1. Import the Plugins SDK project into the Upgrade Planner.
2. Convert the Plugins SDK project to a supported workspace build type.

You'll step through importing a Plugins SDK project first.

Importing Existing Plugins SDK Projects

Initiating this step in the Upgrade Planner imports your Plugins SDK projects into the Upgrade Planner. These projects originate from the Plugins SDK you set when the Upgrade Planner process was started.

If you're manually upgrading your code, you can skip this step.

You're now ready to migrate your Plugins SDK projects to your new workspace!

Migrating Existing Plugins to Workspace

Liferay Workspace can be generated as a Gradle or Maven environment, but it does not support the Plugins SDK's Ant build. Because of this, you must convert your projects to one of the supported build tools:

- Gradle
- Maven

When initiating this step for a Gradle-based workspace, your Ant-based Plugins SDK project is copied to the applicable workspace folder based on its project type (e.g., `wars`) and is converted to a Gradle project. There is also a Blade CLI command that completes this via the command line. Visit the Converting Plugins SDK Projects with Blade CLI article for more information.

If you're migrating your Ant project to a Maven workspace, you must manually copy the project to the applicable folder based on the project type (e.g., `wars`). The majority of Plugins SDK projects belong in the workspace's `wars` folder. You can consult the Workspace Anatomy section for a full overview of a workspace's folder structure and choose where your custom app should reside. Once you've made the decision, copy your custom app to the applicable workspace folder.

Then you must convert your project from Ant to Maven. You'll have to complete this conversion manually.

Once you're finished, you should have your project(s) residing in the applicable workspace folders as Gradle/Maven projects.

13.6 Upgrading Build Dependencies

Now that your projects are readily available in a workspace, you must ensure your project build dependencies are upgraded. Your workspace streamlines the build dependency upgrade process by only requiring three modifications:

- Update the repository URL (Gradle only)
- Update the workspace plugin version
- Remove your project's build dependency versions (Gradle only)

If you're upgrading a recently created workspace, only a subset of these tasks may be required.
You'll start by updating the repository URL.

Updating the Repository URL

Initiating this step in the Upgrade Planner updates the repository URL used to download artifacts for your workspace.

If you're using a Gradle-based workspace, the repository URL is updated to point to the latest Liferay CDN repository. This is set in your workspace's `settings.gradle` file within the `buildscript` block like this:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Once the repository URL is set to the proper CDN repository, your build dependencies will be downloaded from Liferay's own managed repo.

For Maven-based workspaces, Maven Central is the default repository, so no action is required.

Updating the Workspace Plugin Version

For the best upgrade experience, you should ensure you're leveraging the latest Liferay Workspace version so all the latest features are available to you. Initiate this step to upgrade the appropriate plugin.

See the Updating Liferay Workspace article to do this for Gradle-based workspaces manually. For Maven-based workspaces, make sure you set the latest Bundle Support plugin version in your root `pom.xml` file.

Removing Your Project's Build Dependency Versions

Note: This step only applies to Gradle-based workspaces since the target platform feature is only available for Gradle projects at this time.

Since your workspace is leveraging the target platform feature, there is no need to set your plugin's dependency versions in its `build.gradle` file. This is because the target platform version you set already defines the artifact versions your project uses. Therefore, if dependency versions are present in any of your projects' `build.gradle` files, you must remove them.

Initiate this step to remove your dependency versions from your project's `build.gradle` file

As an example of what a `build.gradle`'s `dependencies` block should look like, see the below snippet:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel"
    compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib"
    compileOnly group: "javax.portlet", name: "portlet-api"
    compileOnly group: "javax.servlet", name: "javax.servlet-api"
    compileOnly group: "jstl", name: "jstl"
    compileOnly group: "org.osgi", name: "osgi.cmpn"
}
```

If you have not set the target platform feature in your workspace, see the Managing the Target Platform article for more information.

Great! You've successfully upgraded your build dependencies! You likely have compile errors in your project; this is because your dependencies may have changed. You'll learn how to update that and more next.

13.7 Fixing Upgrade Problems

Now that your development environment build configuration is settled, you can start upgrading your project(s). The two most common upgrade problems are

- Broken project dependencies
- Breaking changes

Visit these upgrade problem tutorials for tips on how to fix them.

This tutorial is heavily focused on the Liferay Upgrade Planner. If you're upgrading your code manually, continue to the listed tutorials above to fix your code upgrade problems.

You'll begin auto-correcting upgrade problems first.

Auto-Correcting Upgrade Problems

Initiate this step to auto-correct straightforward updates like

- package imports
- JSP tag names
- Liferay descriptor versions
- XML descriptor content
- etc.

If you choose to preview the auto-correct upgrade problems first, you can view them in the Project Explorer under the *Liferay Upgrade Problems* dropdown. If you click one of the upgrade problems listed with the preview, you're offered documentation in the *Liferay Upgrade Plan Info* window on the proposed change.

Once you've performed this step, the result list is removed.

Finding Upgrade Problems

Initiating this step finds the upgrade problems that were not eligible for auto-correction. The problems are listed under the *Liferay Upgrade Problems* dropdown. If you click one of the upgrade problems listed with the preview, you're offered documentation in the *Liferay Upgrade Plan Info* window on the proposed change.

These upgrade problems are available in the breaking changes for the version upgrade you're performing.

The next step is resolving the reported upgrade problems.

Resolving Upgrade Problems

Now that the upgrade problems have been located, you must resolve them. As you select each upgrade problem, the documentation for how to adapt your code is displayed in the *Liferay Upgrade Plan Info* window.

For each upgrade problem node, you're also given the version the upgrade problem applies to (e.g., when upgrading to Liferay DXP 7.2 from Liferay Portal 6.2, you could have upgrade problems from the 7.0, 7.1, or 7.2 upgrade). As you step through the reported problems, mark them as resolved/skipped using the context menu. You can right-click on the problem in the Project Explorer and choose from four options:

- Mark done
- Mark undone
- Ignore
- Ignore all problems of this type

Leave this step marked as *Incomplete* until you have resolved all upgrade problems accordingly.

Removing Problem Markers

After resolving all the reported upgrade problems, you must remove all previously found markers because, in most cases, the line number and other accompanying marker information are out of date and must be removed before continuing. Initiate this step to remove all the problem markers.

Great! You've fixed all the upgrade problems that could be automatically detected by the Upgrade Planner. Next, you'll take a deeper look at resolving project dependency errors.

13.8 Resolving a Plugin's Dependencies

Now that you've imported your plugin project to Liferay @ide@, you probably see compile errors for some of the Liferay classes it uses. These classes are listed as undefined classes or unresolved symbols because they've been moved, renamed, or removed. As a part of modularization in Liferay DXP, many of these classes reside in new modules.

You must resolve all of these Liferay classes for your plugin. Some of the class changes are quick and easy to fix. Changes involving the new modules require more effort to resolve, but doing so is still straightforward.

Liferay class changes and required adaptations are described here:

1. **Class moved to a package that's in the classpath**: This change is common and easy to fix. Since the module is already on your classpath, you need only update the class import. You can do this by using the Liferay Code Upgrade Tool or by organizing imports in @ide@. The Upgrade Planner reports each moved class for you to address one by one. Organizing imports in @ide@ automatically resolves multiple classes at once.

   It's typically faster to resolve moved classes using the mentioned Eclipse feature. Since Liferay @ide@ is based on Eclipse, you can generate imports of classes in your classpath with the *Organize Imports* keyboard sequence *Ctrl-Shift-o*. Comment out or remove any imports marked as errors, then press *Ctrl-Shift-o*. If there's only one match for the import, @ide@ automatically generates its import statement. Otherwise, a wizard appears that lets you select the correct import.

2. **Class moved to a module that's *not* in the classpath**: You must resolve the new module as a dependency for your project. This requires identifying the module and specifying your project's dependency on it.

3. **Class replaced or removed**: The class has been replaced by another class or removed from the product. The Upgrade Planner (discussed later) explains what happened to the class, how to handle the change, and why the change was made.

Resolving a class that's moved within your classpath is straightforward. Consider resolving such classes first. The remainder of this tutorial explains how to resolve the last two cases and starts with configuring your plugin project to declare the modules it needs.

Identifying Module Dependencies

Before 7.0, all the platform APIs were in `portal-service.jar`. Many of these APIs are now in independent modules. Modularization has resulted in many benefits, as described in the article Benefits of 7.0 for Liferay Portal 6 Developers. One such advantage is that these API modules can evolve separately from the platform kernel. They also simplify future upgrades. For example, instead of having to check all of Liferay's APIs, each module's Semantic Versioning indicates whether the module contains any backwards-incompatible changes. You need only adapt your code to such modules (if any).

As part of the modularization, `portal-service.jar` has been renamed appropriately to `portal-kernel.jar`, as it continues to hold the portal kernel's APIs.

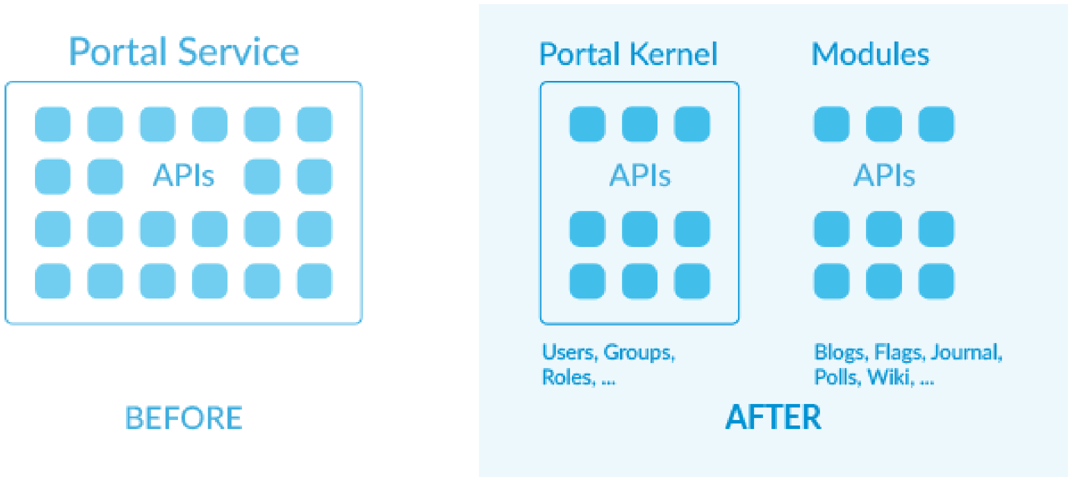

Portal Service
APIs
BEFORE
Portal Kernel
APIs
Users, Groups, Roles, ...
Modules
APIs
Blogs, Flags, Journal, Polls, Wiki, ...
AFTER

Figure 13.1: Liferay refactored the portal-service JAR for 7.0. Application APIs now exist in their own modules, and the portal-service JAR is now *portal-kernel*.

Each app module consists of a set of classes that are highly cohesive and have a specific purpose, such as providing the app's API, implementation, or UI. The app modules are therefore much easier to understand. Next, you'll track down the modules that now hold the classes referenced by your plugin.

The reference article Classes Moved from `portal-service.jar` contains a table that maps each class moved from `portal-service.jar` to its new module. The table includes each class's new package and symbolic name (artifact ID). You'll use this information to configure your plugin's dependencies on these modules.

Your plugin might reference classes that are in Liferay utility modules such as `util-java`, `util-bridges`, `util-taglib`, or `util-slf4j`.

The following table shows each Liferay utility module's symbolic name.

Liferay Utility	Symbolic Name (Artifact ID)
util-bridges	`com.liferay.util.bridges`
util-java	`com.liferay.util.java`
util-slf4j	`com.liferay.util.slf4j`
util-taglib	`com.liferay.util.taglib`

You can use Liferay DXP's App Manager, Felix Gogo Shell, or module JAR file manifests to find versions of modules deployed on your Liferay DXP instance.

Note: Previous versions of the Plugins SDK made `portal-service.jar` available to projects. The 7.0 Plugins SDK similarly makes `portal-kernel.jar` available. If you're using a Liferay DXP bundle (Liferay DXP pre-installed on an app server), the Liferay utility modules are already on your classpath. If you manually installed Liferay DXP on your app server, the Liferay utility modules might not be on your classpath. If a utility module you need is not on your classpath, note its symbolic name (artifact ID) and version.

Resolving Dependencies

Now that you have the module artifact IDs and versions, you can make the modules available to your plugin project. The modules your plugin uses must be available to it at compile time and run time. Here are two options for resolving module dependencies in your traditional plugin project:

Option 1: Use a dependency management tool

Option 2: Manage dependencies manually

The next sections explain and demonstrate these options.

Using a Dependency Management Tool

Dependency management tools such as Ant/Ivy, Maven, and Gradle facilitate acquiring Java artifacts that provide packages your plugins need. They can download artifacts from public repositories or from internal repositories you configure as a proxies. From internal repositories you can audit dependencies.

The following links provide proxy details:

- Ant/Ivy - See documentation on proxy configuration, the `Setproxy` task, and resolvers
- Maven
- Liferay Workspace (Gradle)
- Setting proxies in Liferay @ide@

The Liferay Plugins SDK provides an Ant/Ivy infrastructure. You declare your dependencies in an `ivy.xml` file in your plugin project's root folder. The Plugins SDK's Ant tasks leverage the `ivy.xml` file and the Plugins SDK's Ivy scripts to download the specified modules and their dependencies and make them available to your plugin.

Note: You can use Gradle or Maven in place of Ivy for dependency management, but this isn't in this tutorial's scope. Liferay's Maven and Liferay Workspace tutorials demonstrate using these tools.

Additionally, Liferay Workspace provides a command for migrating Ant/Ivy projects to Gradle-based Liferay Workspace projects. See the tutorial Migrating Traditional Plugins to Workspace Web Applications.

Here's an example dependency element for the Liferay Journal API module, version 2.0.1:

```
<dependency name="com.liferay.journal.api" org="com.liferay" rev="2.0.1" />
```

Each dependency includes the module's name (`name`), organization (`org`), and revision number (`rev`). The Configuring Dependencies tutorial explains how to determine the module's organization (`org`).

At compile time, Ivy downloads the dependency JAR files to a cache folder so you can compile against them.

At deployment, Liferay DXP's WAB Generator creates an OSGi Web Application Bundle (WAB) for the plugin. The WAB generator detects the Java packages your plugin uses and declares dependencies on them. Your plugin can use the packages once a registered OSGi service provides them.

If your project doesn't already have an `ivy.xml` file, you can get one by creating a new plugin project in Liferay @ide@ and copying the `ivy.xml` file it generates.

Here's an example of an `ivy.xml` file from the Liferay Portal 6.2 Knowledge Base portlet:

```
<?xml version="1.0"?>

<ivy-module
    version="2.0"
    xmlns:m2="http://ant.apache.org/ivy/maven"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/schemas/ivy.xsd"
>
    <info module="knowledge-base-portlet" organisation="com.liferay">
        <extends extendType="configurations,description,info" location="${sdk.dir}/ivy.xml" module="com.liferay.sdk" organisation="com.liferay" revision="la
    </info>

    <dependencies defaultconf="default">
        <dependency org="com.liferay" name="com.liferay.markdown.converter" rev="1.0.2" />
    </dependencies>
</ivy-module>
```

The Plugins SDK works with project Ivy files to store artifacts and make them accessible to your plugin projects.

If you don't want to use Ivy or some other dependency management framework, you can store dependency JARs within your plugin project manually. You'll learn about this next.

Managing Dependencies Manually

Plugins rely on their dependencies' availability at compile time and run time. To compile your plugin, you must make sure the dependencies are available in the plugin's `WEB-INF/lib` folder. To run your plugin, the container must be able to find them: either 1) the dependency Java packages must already be active in Liferay DXP's OSGi framework or 2) the dependency JARs must be included in the WAB generated for the plugin. Your plugin can use both the JARs it currently has and the packages Liferay DXP exports.

Using Packages Liferay DXP Exports The Plugins SDK for Liferay Portal 6 provided a way to compile against JARs it had. You'd specify these JARs in the `portal-dependency-jars` property in your `liferay-plugin-package.properties` file. On seeing a plugin's `portal-dependency-jars` list, the Liferay Plugins SDK copied the JARs into the plugin's `WEB-INF/lib`. The Plugins SDK refrained from adding the JARs to the plugin WAR.

This kept the WARs small for deploying faster. It was especially useful for deploying WARs remotely or to cluster nodes.

In 7.0, the `portal-dependency-jars` property is deprecated and behaves differently from previous versions. Because importing and exporting Java packages has replaced wholesale use of JARs, modules and WABs can import packages without concerning themselves with JARs. Liferay DXP exports many third party packages for plugins can use.

If you're still using the `portal-dependency-jars` property, you may run into one of the scenarios below. Follow the instructions below the scenario to fix the issue.

1. **I've specified a JAR, but in 7.0 none of the classes are available to my plugin.**

   Some JARs that Liferay Portal 6.2 used were removed in 7.0. If you specify them in your `portal-dependency-jars`, Liferay DXP can't provide them. If you still need them, remove them from the `portal-dependency-jars` property and add the JARs you need to your plugin's `WEB-INF/lib` folder.

2. **I've specified JARs, and 7.0 also exports all the JAR's packages my plugin imports**

   Keep the JAR in your `portal-dependency-jars` list. The Plugins SDK copies the JAR to your plugin's `WEB-INF/lib` folder at compile time but refrains from adding the JAR to the plugin WAB. The WAB generated for the plugin imports the packages from a registered provider at run time.

3. **7.0 provides the JAR but doesn't export a package my plugin imports**

   Keep the JAR in your `portal-dependency-jars` property. The Plugins SDK copies the JAR to your plugin's `WEB-INF/lib` folder at compile time and adds the JAR to the plugin WAB at deployment.

Understanding Excluded JARs Portal property `module.framework.web.generator.excluded.paths` declares JARs that are stripped from all Liferay DXP generated WABs. These JARs are excluded from WABs because Liferay DXP provides them already. All JARs listed for this property are excluded from the WABs, even if the plugins listed the JAR in their `portal-dependency-jars` property.

If your plugin requires different versions of the packages Liferay DXP exports, you must include them in JARs named differently from the ones `module.framework.web.generator.excluded.paths` excludes.

For example, Liferay DXP's `system.packages.extra.bnd` file exports Spring Framework version 4.1.9 packages:

```
Export-Package:\
    ...
    org.springframework.*;version='4.1.9',\
    ...
```

Liferay DXP uses the `module.framework.web.generator.excluded.paths` portal property to exclude their JARs.

```
module.framework.web.generator.excluded.paths=\
    ...
    WEB-INF/lib/spring-aop.jar,\
    WEB-INF/lib/spring-aspects.jar,\
    WEB-INF/lib/spring-beans.jar,\
    WEB-INF/lib/spring-context.jar,\
    WEB-INF/lib/spring-context-support.jar,\
    WEB-INF/lib/spring-core.jar,\
    WEB-INF/lib/spring-expression.jar,\
    WEB-INF/lib/spring-jdbc.jar,\
    WEB-INF/lib/spring-jms.jar,\
    WEB-INF/lib/spring-orm.jar,\
    WEB-INF/lib/spring-oxm.jar,\
    WEB-INF/lib/spring-tx.jar,\
    WEB-INF/lib/spring-web.jar,\
    WEB-INF/lib/spring-webmvc.jar,\
    WEB-INF/lib/spring-webmvc-portlet.jar,\
    ...
```

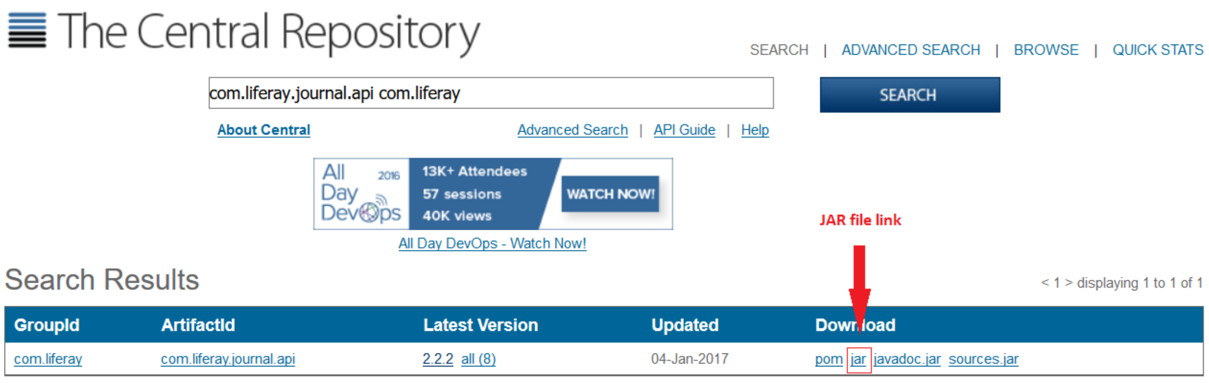
To use a different Spring Framework version in your WAB, you must name the corresponding Spring Framework JARs differently from the glob-patterned JARs `module.framework.web.generator.excluded.paths` lists.

For example, to use Spring Framework version 3.0.7's Spring AOP JAR, include it in your plugin's `WEB-INF/lib` but name it something other than `spring-aop.jar`. Adding the version to the JAR name (i.e., `spring-aop-3.0.7.RELEASE.jar`) differentiates it from the excluded JAR and prevents it from being stripped from the WAB.

Using Packages Liferay DXP Doesn't Export You must download and install to your plugin's `WEB-INF/lib` folder JARs that provide packages Liferay DXP doesn't export that your plugin requires.

Follow these steps to do this:

1. Go to Maven Central at https://search.maven.org/.
2. Search for the module by its artifact ID and group ID.
3. Navigate the search results to find the version of the module you want.
4. Click the *jar* link to download the module's JAR file.
5. Add the JAR to your project's `WEB-INF/lib` folder.


The Central Repository
SEARCH | ADVANCED SEARCH | BROWSE | QUICK STATS
com.liferay.journal.api com.liferay
SEARCH
About Central
Advanced Search | API Guide | Help
All Day DevOps 2016
13K+ Attendees
57 sessions
40K views
WATCH NOW!
All Day DevOps - Watch Now!
JAR file link
Search Results
< 1 > displaying 1 to 1 of 1

GroupId	ArtifactId	Latest Version	Updated	Download
com.liferay	com.liferay.journal.api	2.2.2 all (8)	04-Jan-2017	pom jar javadoc.jar sources.jar

Figure 13.2: After searching Maven Central, download an artifact's JAR file by clicking the *jar* link.

As you manage module JARs, make sure **not** to deploy any OSGi framework JARs or Liferay module JARs (e.g., `com.liferay.journal.api.jar`). If you deploy these, they'll conflict with the JARs already installed in the OSGi framework. Identical JARs in two different classloaders can cause class cast exceptions. The easiest way to exclude such JARs from your plugin's deployment is to list them in a `deploy-excludes` property in your plugin's `liferay-plugin-package.properties`. You must otherwise remove the JARs manually from the plugin WAR file. To exclude JARs in your plugin's `liferay-plugin-package.properties` file, add an entry like the one below, replacing the square-bracketed items with the names of JAR files to exclude:

```
deploy-excludes=\
    **/WEB-INF/lib/[module-artifact.jar],\
    **/WEB-INF/lib/[another-module-artifact.jar]
```

For example, here's an example property that excludes the OSGi framework JAR `osgi.core.jar` and the Liferay app module JAR `com.liferay.journal.api.jar`:

```
deploy-excludes=\
    **/WEB-INF/lib/com.liferay.portal.journal.api.jar,\
    **/WEB-INF/lib/org.osgi.core.jar
```

How do you know what modules are already installed in Liferay DXP? If your Liferay DXP instance has a particular Liferay app suite installed, then don't deploy module JARs you know are in that app suite. For example, if the Web Experience Management App Suite is already installed (which is the case for a Liferay DXP bundle), then don't deploy Web Content module JARs such as `com.liferay.journal.api.jar`. Searching for a module in Liferay DXP's App Manager is a sure-fire way to verify existing module installations.

Using Liferay DXP's Tag Library Definitions

Before adding Tag Library Definition (TLD) files to your plugin, check Liferay DXP for them. The Liferay DXP web application folder `WEB-INF/TLD` has over twenty TLDs, including Struts TLDs.

You can use Liferay DXP's TLDs in a traditional plugin by adding them to the `portal-dependency-tlds` property in the plugin's `liferay-plugin-package.properties` file.

Way to go! You've fixed class imports and resolved dependencies on all the modules and tag libraries your plugin uses.

Related Topics

Importing Packages

Exporting Packages

Development Reference

Modularizing an Existing Portlet

Invoking Local Services

Finding and Invoking Liferay Services

Tooling

13.9 Resolving Breaking Changes

Liferay goes to great lengths to maintain backwards compatibility. Sometimes, breaking changes are necessary to improve Liferay DXP. There may be cases where breaking changes affect your code upgrade process and must be resolved. A breaking change can include

- Functionality that is removed or replaced
- API incompatibilities: Changes to public Java or JavaScript APIs
- Changes to context variables available to templates
- Changes in CSS classes available to Liferay themes and portlets
- Configuration changes: Changes in configuration files, like `portal.properties`, `system.properties`, etc.
- Execution requirements: Java version, J2EE Version, browser versions, etc.

- Deprecations or end of support: For example, warning that a certain feature or API will be dropped in an upcoming version.
- Recommendations: For example, recommending using a newly introduced API that replaces an old API, in spite of the old API being kept in Liferay Portal for backwards compatibility.

Liferay provides a list of breaking changes for every major release to ensure you can easily adapt your code during the upgrade process.

- 7.0 Breaking Changes
- Liferay DXP 7.1 Breaking Changes
- Liferay DXP 7.2 Breaking Changes

The easiest way to resolve breaking changes is by using the Liferay Upgrade Planner. It automatically finds all documented breaking changes and can automatically resolve some of them on its own.

If you're resolving breaking changes manually, make sure to investigate each breaking change document if you're upgrading code across multiple versions. For example, if you're upgrading from Liferay Portal 6.2 to 7.0, you must resolve all the breaking changes listed in the three documents listed above.

Now that you've resolved your breaking changes, you'll learn how to upgrade service builder services next.

CHAPTER 14

UPGRADING HOOK PLUGINS

Liferay DXP has more extension points than ever, and connecting existing hook plugins to them takes very few steps. In most cases, after you upgrade your hook using the Liferay Upgrade Planner, it's ready to run on Liferay DXP. The following tutorials show you how to upgrade each type of hook plugin.

- Override/Extension Modules
- Core JSP Hooks
- App JSP Hooks
- Service Wrapper Hooks
- Core Language Key Hooks
- Portlet Language Key Hooks
- Model Listener Hooks
- Servlet Filter Hooks
- Portal Property and Event Action Hooks
- Struts Action Hooks

Continue on to get started!

14.1 Upgrading Customization Modules

Customization modules include any module extension or override used to customize another module. For examples of these types of modules, visit the `extensions` and `overrides` sample projects.

Getting a customization module running on 7.0 takes two steps:

1. Adapt your code to 7.0's API using the Liferay Upgrade Planner. When you ran the planner's *Fix Upgrade Problems* step, many of the existing issues were autocorrected or flagged. For any remaining errors, consult the Resolving a Plugin's Dependencies article.

2. Deploy your module.

Note: A fragment was a common customization module in past versions of Liferay DXP. Fragments are no longer recommended; you should upgrade a fragment to a dynamic include or portlet filter. For more information on recommended ways of customizing JSPs in 7.0, see the Customizing JSPs section.

Great! Your customization module is upgraded for 7.0!

14.2 Upgrading Core JSP Hooks

Getting a core JSP hook running on 7.0 takes two steps:

1. Adapt your code to 7.0's API using the Liferay Upgrade Planner. When you ran the planner's *Fix Upgrade Problems* step, many of the existing issues were autocorrected or flagged. For any remaining errors, consult the Resolving a Plugin's Dependencies article.

2. Deploy your hook plugin.

Liferay DXP's Plugin Compatibility Layer converts the plugin WAR to a Web Application Bundle (WAB) and installs it to Liferay's OSGi Runtime.

It's just that easy!

Related Topics

JSP Overrides Using Custom JSP Bags

Upgrading App JSP Hooks

Resolving a Plugin's Dependencies

Upgrading the Liferay Maven Build

14.3 Upgrading App JSP Hooks

JSPs in OSGi modules are customized using module fragments. The module fragment attaches to the host module to alter the JSPs. To the OSGi runtime, the fragment is part of the host module. Section 3.14 of the OSGi Alliance's core specification document explains module fragments in detail. This tutorial shows you how to upgrade your app JSP hooks to 7.0.

Liferay @ide@'s Upgrade Planner's *Fixing Upgrade Problems* step generates module fragments from app JSP hook plugins. The tool creates module fragments in the same folder as your Plugins SDK root if your hook is in a Plugins SDK or in the `[liferay_workspace]/modules` folder if your hook is in a Liferay Workspace.

Module fragments follow this name convention: `[plugin_name]-[app]-fragment`. For example, if the plugin's name is `app-jsp-hook` and it modifies a JSP in the Blogs app, the Upgrade Planner generates a module fragment named `app-jsp-hook-blogs-fragment`.

Here are the steps for upgrading app JSP hook plugins:

1. Declare the Fragment Host
2. Update the JSP

Declare the Fragment Host

The module fragment's `bnd.bnd` file must specify an OSGi header `Fragment-Host` set to the host module name and version.

If the host module belongs to one of Liferay DXP's app suites, the Code Upgrade Tool generates a `bnd.bnd` file that specifies an appropriate `Fragment-Host` header automatically.

For example, here's a `Fragment-Host` that attaches a module fragment to the Blogs Web module.

```
Fragment-Host: com.liferay.blogs.web;bundle-version="1.1.9"
```

Updating the JSP is straightforward too.

Update the JSP

The Upgrade Planner creates a module fragment that contains an upgraded version of your custom app JSP.

The following table shows the old and new JSP paths.

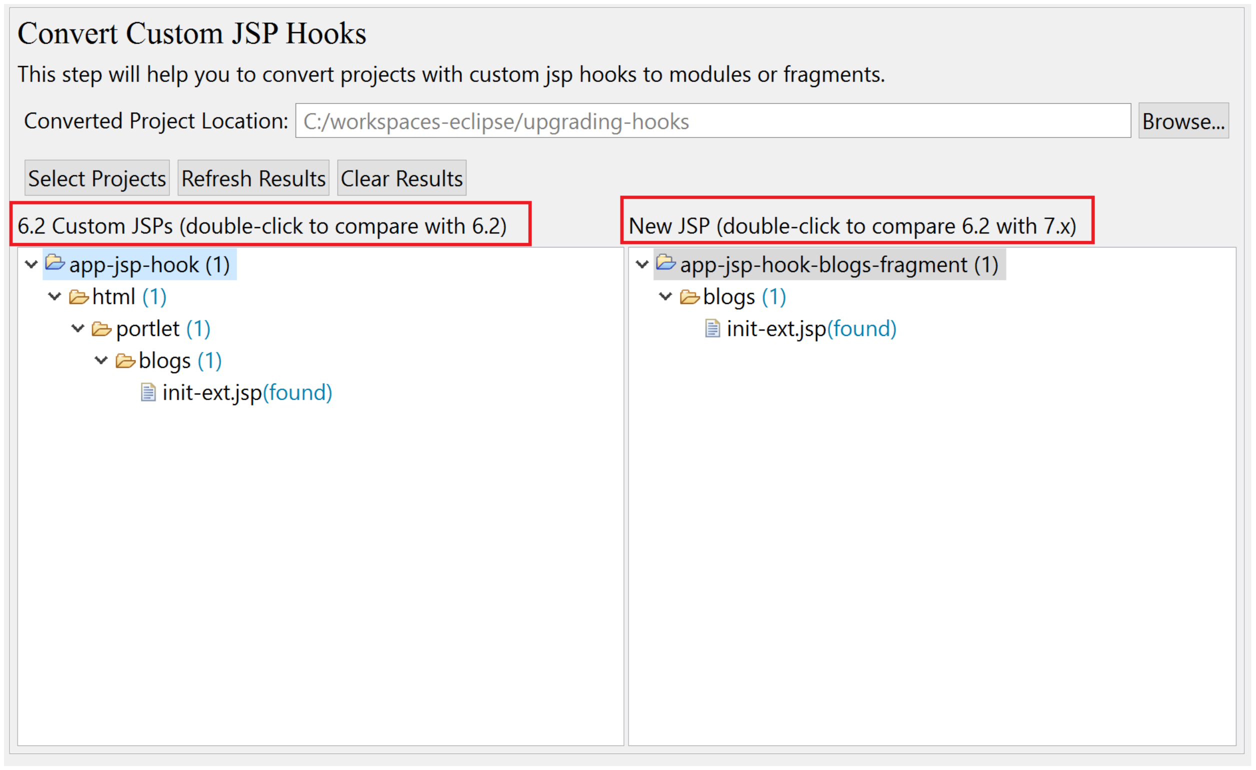
Liferay Portal version	JSP File Path
6.2	`docroot/META-INF/custom_jsps/html/portlet/[jsp_file_path]`
7.0	`src/main/resources/META-INF/resources/[jsp_file_path]`

For example, the Upgrade Planner generates a customized version of the Blogs app's `init-ext.jsp` file here:

```
src/main/resources/META-INF/resources/blogs/init-ext.jsp
```

The tool's *Fixing Upgrade Problems* step lets you compare custom JSPs with originals:

- Compare your custom 6.2 JSP with the original 6.2 JSP.
- Compare your custom 7.0 JSP with your custom 6.2 JSP.


Convert Custom JSP Hooks

This step will help you to convert projects with custom jsp hooks to modules or fragments.

Converted Project Location: C:/workspaces-eclipse/upgrading-hooks

Figure 14.1: The Upgrade Planner lets you compare custom JSPs with originals.

Make any additional needed changes in your 7.0 custom JSP. Then deploy your module fragment. This stops the host module momentarily, attaches the fragment to the host, and then restarts the host module. The console output reflects this process.

Here's output from deploying a module fragment that attaches to the Blogs web module.

```
19:23:11,740 INFO  [Refresh Thread: Equinox Container: 00ce6547-2355-0017-1884-846599e789c4][BundleStartStopLogger:38] STOPPED com.liferay.blogs.web_1.1.9 [
19:23:12,910 INFO  [Refresh Thread: Equinox Container: 00ce6547-2355-0017-1884-846599e789c4][BundleStartStopLogger:35] STARTED com.liferay.blogs.web_1.1.9 [
```

Your custom JSP is live.

Related Topics

JSP Overrides Using OSGi Fragments
Upgrading Core JSP Hooks
Resolving a Plugin's Dependencies
Upgrading the Liferay Maven Build

14.4 Upgrading Service Wrappers

Upgrading traditional service wrapper hook plugins to 7.0 is quick and easy.

1. Adapt your code to 7.0's API using the Liferay Upgrade Planner. When you ran the planner's *Fix Upgrade Problems* step, many of the existing issues were autocorrected or flagged. For any remaining errors, consult the Resolving a Plugin's Dependencies article.

2. Deploy the plugin.

Liferay DXP's Plugin Compatibility Layer converts the plugin WAR to a Web Application Bundle (WAB) and installs it to Liferay's OSGi Runtime.

Related Articles

Overriding Liferay Services (Service Wrappers)
Resolving a Plugin's Dependencies
Upgrading the Liferay Maven Build

14.5 Upgrading Core Language Key Hooks

Here are the steps for upgrading a core language key hook to 7.0.

1. Create a new module based on the Blade sample `resource-bundle` in Gradle or Maven.

   Here are the main parts of the module folder structure:

   - `src/main/java/[resource bundle path]` → Custom resource bundle class goes here
   - `src/main/resources/content`
     - `Language.properties`
     - `Language_xx.properties`
     - ...

2. Copy all your plugin's language properties files into the module folder `src/main/resources/content/`.

3. Create a resource bundle loader.

4. Deploy your module.

Your core language key customizations are deployed to 7.0.

Related Topics

Overriding Global Language Keys
Upgrading Portlet Language Key Hooks
Resolving a Plugin's Dependencies
Upgrading the Liferay Maven Build

14.6 Upgrading Portlet Language Key Hooks

You can upgrade your portlet language key hooks to 7.0 by following these steps:

1. Create a new module based on the Blade sample `resource-bundle` (Gradle or Maven project).

   Here are the module folder structure's main files:

   - `src/main/java/[resource bundle path]` → `ResourceBundleLoader` extension goes here
   - `src/main/resources/content`
     - `Language.properties`
     - `Language_xx.properties`
     - ...

2. Copy your language properties files into module folder `src/main/resources/content/`.

3. In your `bnd.bnd` file, specify OSGi manifest headers that target the portlet module's resource bundle, but prioritize yours.

4. Deploy your module.

Your portlet language key customizations are deployed in your new module on 7.0.

Related Topics

Overriding a Module's Language Keys
Upgrading Core Language Key Hooks
Resolving a Plugin's Dependencies
Upgrading the Liferay Maven Build

14.7 Upgrading Model Listener Hooks

Developers have been creating model listeners for several Liferay Portal versions. Upgrading Model Listener Hooks from previous portal versions has never been easier.

1. Adapt your code to 7.0's API using the Liferay Upgrade Planner. When you ran the planner's *Fix Upgrade Problems* step, many of the existing issues were autocorrected or flagged. For any remaining errors, consult the Resolving a Plugin's Dependencies article.

2. Deploy the plugin.

Liferay DXP's Plugin Compatibility Layer converts the plugin WAR to a Web Application Bundle (WAB) and installs it to Liferay's OSGi Runtime.

Your model listener hook is "all ears" and ready to act.

Related Topics

Creating Model Listeners
Resolving a Plugin's Dependencies
Upgrading the Liferay Maven Build

14.8 Upgrading Servlet Filter Hooks

If you have Servlet Filter Hooks ready to be upgraded, this tutorial's for you. The process is quite simple:

1. Adapt your code to 7.0's API using the Liferay Upgrade Planner. When you ran the planner's *Fix Upgrade Problems* step, many of the existing issues were autocorrected or flagged. For any remaining errors, consult the Resolving a Plugin's Dependencies article.

2. Deploy the plugin.

Liferay DXP's Plugin Compatibility Layer converts the plugin WAR to a Web Application Bundle (WAB) and installs it to Liferay's OSGi Runtime.

Your Servlet Filter is running on 7.0!

Related Topics

Resolving a Plugin's Dependencies
Upgrading the Liferay Maven Build

14.9 Upgrading Portal Property and Event Action Hooks

All portal properties in Liferay Portal 6.2 that are also used in 7.0 can be overridden. Portal property and portal event action hooks that use these properties can be upgraded to 7.0 by following these steps:

1. Adapt your code to 7.0's API using the Liferay Upgrade Planner. When you ran the planner's *Fix Upgrade Problems* step, many of the existing issues were autocorrected or flagged. For any remaining errors, consult the Resolving a Plugin's Dependencies article.

2. Deploy your hook plugin.

Liferay DXP's Plugin Compatibility Layer converts the plugin WAR to a Web Application Bundle (WAB) and installs it to Liferay's OSGi Runtime.

Your custom property values and actions are live.

Related Topics

Liferay @ide@
Resolving a Plugin's Dependencies
Upgrading the Liferay Maven Build

14.10 Converting StrutsActionWrappers to MVCCommands

Since Liferay Portal 6.1, developers could customize the Portal and Portlet Struts Actions using a Hook and `StrutsActionWrappers`. For example, the `liferay-hook.xml` file for a hook that overrode the login portlet's login action had this entry:

```
<struts-action>
    <struts-action-path>/login/login</struts-action-path>
    <struts-action-impl>
    com.liferay.sample.hook.action.ExampleStrutsPortletAction
    </struts-action-impl>
</struts-action>
```

The `liferay-hook.xml` contains the struts mapping and the new class that overrides the default login action.

The wrapper could extend either `BaseStrutsAction` or `BaseStrutsPortletAction`, depending on whether the struts action was a portal or portlet action respectively.

Starting in 7.0, this mechanism no longer applies for most of the portal portlets because they are no longer using Struts Actions, but instead use `MVCCommands`.

This tutorial demonstrates how to convert your existing `StrutsActionWrappers` to `MVCCommands`.

Converting Your old wrapper to MVCCommands

Converting `StrutsActionWrappers` to `MVCCommands` is easier than you may think.

As a review, legacy `StrutsActionWrappers` needed to implement all the methods, such as `processAction`, `render`, and `serveResource`, even if only one method was being customized. Each of these methods can now be customized independently, using different classes, making the logic simpler and easier to maintain. Depending on the method you customized in your `StrutsActionWrapper`, you need to use the matching MVCCommand shown below:

- processAction → MVCActionCommand
- render → MVCRenderCommand
- serveResource → MVCResourceCommand

Look at the ExampleStrutsPortletAction class for a `StrutsActionWrapper` example. Depending on the actions overridden, the user must use different `MVCCommands`. In this example, the action and render were overridden, so in order to migrate to the new pattern, you would need to create two classes: `MVCActionCommand` and `MVCRenderCommand`.

Next you'll need to determine the associated mapping that is used by the `MVCCommand`.

Mapping Your MVCCommand URLs

For most cases, the `MVCCommand` mapping is the same mapping defined in the legacy struts action.

Using the beginning login example once again, the `struts-action-path` mapping, `/login/login`, remains the same for the `MVCCommand` mapping in 7.0, but some of the mappings may have changed. It's best to check Liferay's source code to determine the correct mapping.

Depending on the URL it is a different parameter:

- `RenderURLs` contain a parameter named `mvcRenderCommandName`. For example:

```
<portlet:renderURL var="editEntryURL">
    <portlet:param name="mvcRenderCommandName" value="/hello/edit_entry"
    />
    <portlet:param name="entryId" value="<%= String.valueOf(
    entry.getEntryId()) %>" />
</portlet:renderURL>
```

- `ActionURLs` are contained in the attribute `name` of the tag library or with the parameter `ActionRequest.ACTION_NAME`. For example:

```
<portlet:actionURL name="/blogs/edit_entry" var="editEntryURL" />
```

- `ResourceURLs` are contained as the attribute `id`. For example:

```
<portlet:resourceURL id="/login/captcha" var="captchaURL" />
```

Once you have this information, you can override the `MVCCommand` by following the instructions found in these sections of the Overriding MVC Commands tutorial:

- Overriding MVCActionCommands
- Overriding MVCResourceCommands
- Overriding MVCRenderCommands

Now you know how to convert your `StrutsActionWrappers` to `MVCCommands`!

Related Topics

Overriding MVC Commands

Chapter 15

Upgrading Themes

If you've developed themes in Liferay Portal 6.1 or Liferay Portal 6.2, as part of your upgrade you'll want to use them in 7.0. The upgrade process requires several modifications. To help automate this process you'll use the Liferay Theme Generator.

The following tutorials show you how to upgrade your Liferay Portal 6.1 and Liferay Portal 6.2 themes to 7.0.

15.1 Upgrading Your Theme from Liferay Portal 6.1 to 7.0

This tutorial guides you through the process of upgrading your 6.1 theme to run on 7.0.

For a more in depth tutorial that covers upgrading 6.2 themes, please see the Upgrading Themes tutorial.

Setting Up Your Theme With Liferay Theme Generator

You'll use Liferay Theme Generator to get the upgrade process started. Liferay Theme Generator supplies your theme with the necessary tools to deploy and make quick modifications.

In this tutorial, you'll use Liferay Theme Generator's Import feature to help set up the upgrade process.

Preparing Your Theme

Before you import your theme, you need to make some modifications to your theme to ensure Liferay Theme Generator sets up the correct version and template language.

Make sure these settings are present in your theme:

1. Open your `liferay-look-and-feel.xml` file and make sure the template language (`vm`, `ftl`, `jsp`) is defined:

```
<look-and-feel>
    ...
    <theme id="my-theme-name" name="My Theme Name">
        <template-extension>vm</template-extension>
        ...
```

2. Also in `liferay-look-and-feel.xml`, set the theme version to 6.2+:

```
<look-and-feel>
    <compatibility>
        <version>6.2+</version>
    </compatibility>
    ...
```

3. Finally, open your `liferay-plugin-package.properties` file and update the Liferay Portal version to 6.2+, as well:

   ```
   liferay-versions=6.2+
   ```

Note: Liferay Theme Generator's importer only recognizes 6.2 themes. Therefore, 6.1 themes must be versioned as 6.2+ to be recognized by the importer.

The importer only supports importing 6.2 themes generated with the Plugins SDK. If your theme was not generated with the Plugins SDK, such as those created with Maven, you must modify your theme to match the Plugins SDK structure before importing.

Now that your theme is prepped, you can start the import process next.

Importing Your Theme

Follow these steps to import your theme:

1. Install Liferay Theme Generator and its required dependencies, as explained in the Liferay Theme Generator tutorial.

2. Navigate to the folder where you want your theme imported:

   ```
   $ cd /where/i/want/my-theme/imported
   ```

3. Import your theme by running this command:

   ```
   $ yo liferay-theme:import
   ```

4. Follow the prompts to complete your theme import.

Your theme has been imported into a folder named after your original theme. For example, a theme named `my-awesome-theme-for-61` would be imported into a folder called `my-awesome-theme-for-61-theme`.

Next, you can learn how to update your theme for 7.0.

Updating Your Theme For 7.0

Now that you've successfully imported your theme, you have the tools necessary to develop your theme for 7.0. These tools are provided by the Liferay Theme Tasks, part of the Liferay JS Theme Toolkit.

The first of these tools you'll use is the Upgrade task. This task does a few things:

- Backs up your files so you can revert them later if needed.

- Modifies and renames some files for compatibility.

- Creates a `css.diff` file under `/_backup`, so you can easily see what's been changed.
- Updates NPM dependencies.
- Creates a "compatibility" file for Compass mixins, which may have been used in your 6.1 theme.

Important: Before you run the Update task, read the section below titled An Important Note About Core Stylesheets for advice on renaming (effectively overriding) "core" stylesheets.

Upgrade your theme, by running these commands:

```
$ cd my-awesome-theme-for-62-theme
```

```
$ gulp upgrade
```

If the Upgrade task didn't update your files like you expected, you can easily revert them back using the Upgrade Revert task:

```
$ gulp upgrade:revert
```

The Upgrade task automates some of the steps for you, but some manual changes are still required. These are covered next.

Additional Manual Changes To Your Theme

When the Upgrade task finishes, the console may output additional manual changes you may need to make to your theme.

Note: Because Liferay Portal 7.0 uses Bootstrap 3, the default box model has been changed to box-sizing: border-box. If you were using width or height and padding together on an element, you may need to make changes, or those elements may have unexpected sizes.

There are likely changes that you still need to make, which these tools and this guide can't realistically cover. Therefore, it's recommended that you make changes gradually, rather than all at once. This allows you to more easily identify any issues your theme may have.

Please refer to the Appendix below for ways on how to update your theme manually.

Follow the instructions in the next section to build and deploy your theme.

Building And Deploying Your Theme

Follow these steps to build and deploy your theme:

1. To check that everything is running smoothly, run this command:

   ```
   $ gulp build
   ```

2. If all goes well, you can now deploy your theme by running this command:

   ```
   $ gulp deploy
   ```

If you're using Velocity as your template language, you'll notice this message on your server console when deploying your theme:

Support of Velocity is deprecated. Update your theme to use FreeMarker for forward compatibility.

It's recommended that you migrate to using Freemarker for your theme templates. For more details, please see the Upgrading Theme Templates section in the Appendix.

Additional tasks and info can be found in the Liferay Gulp Tasks reference guide.

Congratulations! You've now completed a major step in upgrading your theme.

Appendix: Making Manual Compatibility Adjustments To Your Theme

This appendix covers the changes you need to make to your theme if you manually update your files, or if you need to make additional changes after using the Liferay JS Theme Toolkit's Upgrade task.

Upgrading Theme Metadata

Your theme contains metadata about its capabilities and requirements. This metadata needs to be updated to ensure your theme is available for use in Liferay DXP.

Liferay Portal Version Your theme must target the proper Liferay DXP version: in your case, 7.0.0+. You must update the theme version in two places:

1. In the file `liferay-plugin-package.properties`, add the following property to ensure Liferay DXP deploys your theme:

   ```
   liferay-versions=7.0.0+
   ```

2. Add the following version information to your `liferay-look-and-feel.xml` to ensure Liferay DXP makes your theme available for use:

   ```
   <look-and-feel>
       <compatibility>
           <version>7.0.0+</version>
       </compatibility>
       ...
   ```

Template Extension If you're not using the default Theme Template language, Freemarker (.ftl), you must add a `template-extension` tag to your `liferay-look-and-feel.xml` file:

```
<look-and-feel>
    ...
    <theme id="my-theme-name" name="My Theme Name">
        <template-extension>vm</template-extension>
        ...
```

Required Deployment Contexts If you have any Required Deployment Contexts listed in your `liferay-plugin-package.properties` file, these will likely need to be removed.

- `resources-importer-web`: Portal Compatibility Hook is for previous versions of Liferay Portal.
- `portal-compat-hook`: Because this tutorial is focused on upgrading a theme rather than importing content and assets, it does not cover the Resources Importer. You can learn more about the Resources Importer in the Importing Resources with a Theme tutorial.

Upgrading Stylesheets

An Important Note About "Core" Stylesheets If you've made customizations to "core" Liferay DXP stylesheets, it's recommended that you not use these customizations right away. Liferay DXP's UI has changed significantly since version 6.1, and any customizations you may have are either not likely to work, or will cause problems in the UI.

These "core" files would be:

- `application.css`
- `aui.css`
- `base.css`
- `dockbar.css`
- `extras.css`
- `layout.css`
- `main.css`
- `navigation.css`
- `portal.css`
- `portlet.css`
- `taglib.css`

Important: Rename these files to something like `my_application.css`, `my_aui.css`, etc. If you don't rename these files, you may get an error when building your theme. This is because Sass is importing ambiguous names like application, aui, etc.

It's recommended that you get your theme working in 7.0 first, and then audit the styles in these "core" CSS files to see if they're still necessary and don't cause any issues.

Renaming CSS Files 7.0 now uses the `.scss` extension for Sass stylesheets.

1. Under `my-imported-theme/src/css`, rename your `custom.css` file to `_custom.scss`.
2. If you have any additional CSS files that need to take advantage of Sass and Bourbon, you must rename those as well.

   For example, `my-feature.css` becomes `_my-feature.scss`, and `fonts.css` becomes `_fonts.scss`.

Imported Stylesheet References If you've imported any renamed stylesheets into another, you'll want to update those references.

For example `_my-fonts.scss` needs to be updated from:

```
@import url(my-fonts.css);
```

to:

```
@import "my-fonts";
```

Sass Framework Import Beginning with version 7.0, Liferay Portal now uses Bourbon for mixins, rather than Compass. Bourbon is being imported in a core stylesheet, _imports.scss, so you shouldn't need to import it yourself.

Also, files with the `.scss` extension are processed by Sass automatically. But you must still import or link to these to use them.

Remove `@import "compass"` from all your Sass files, and don't worry about importing Bourbon.

Mixins When Liferay Portal switched from using Compass to Bourbon, some mixins changed syntax, became unsupported, or became unnecessary in modern browsers. These mixins should be updated or removed.

For example:

```
.my-component {

    @include border-radius(8px);

    @include opaque;

    @include scale(0.9);

}

input[type="text"] {

    @include input-placeholder {

        color: #bfbfbf;

    }

}
```

becomes:

```
.my-component {

    border-radius: 8px;

    opacity: 1;

    @include transform(scale(0.9));

input[type="text"] {

    @include placeholder {

        color: #bfbfbf;

    }

}
```

Please refer to Compass and Bourbon documentation for more details.

Bootstrap 3 If you've used Bootstrap 2.3.2 in your theme, a migration guide is available.

Remove `.aui` Class Used In Descendant Selectors The class `.aui` is no longer in use. Selectors that contain `.aui` as a parent qualifier must be modified, or those selectors will not match the expected elements.

For example:

```
.aui{

    my-component {

        // ...

    }

}

.aui my-other-component {

    // ...

}
```

becomes:

```
html {

    my-component {

        // ...

    }

}

html my-other-component {

    // ...

}
```

or maybe:

```
my-component {

    // ...

}

my-other-component {

    // ...
}
```

Upgrading JavaScript

In August 2014 Yahoo stopped all new development of YUI, which AlloyUI was built on. Because of this announcement, Liferay has decided to sunset AlloyUI and deprecate it as of Liferay Portal 7.

Although AlloyUI is deprecated, it's still available by default, and existing AlloyUI 3 based code continues to work.

To further research changes to an AlloyUI module, you can find a `HISTORY.md` file under each component's directory.

Moving forward, it's recommended to migrate your AlloyUI/YUI code to either Metal.js or the framework of your choice. Below, you'll find some libraries used in Liferay DXP. Feel free to use them as well.

jQuery jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler. It's also popular among front-end developers, making it ideal for getting started quickly.

For more in depth coverage, see jQuery's documentation.

Lodash Lodash is a modern JavaScript utility library delivering modularity, performance & extras. It's used in Liferay DXP to fill the void left by YUI's utility modules.

For more in depth coverage, see Lodash's documentation.

Metal.js Written by Liferay to specifically to meet the needs of Liferay DXP, Metal.js is a JavaScript library for building UI components in a solid, flexible way.

For more in depth coverage, see Metal.js's documentation

Upgrading Theme Templates

In prior versions of Liferay Portal, Velocity templates (`.vm` files) were the default template language for writing theme markup. But Velocity had limitations. In 7.0, Freemarker templates (`.ftl` files) are the default template language, and Velocity templates are deprecated.

With that in mind, if your themes are written in `.vm` files, they should still work, but you're missing out on additional theme features. In a future release, support for Velocity templates will be removed.

Migrating to Freemarker Templates For a typical theme, migrating to Freemarker should be simple. The syntaxes of both languages are relatively similar.

Here's a comparison:

Velocity:

```
$theme.include($my_template_include)

#if ($show_site_name)

    <span title="#language_format ("go-to-x", [$site_name])">

        $site_name

    </span>

#end

#parse ("$full_templates_path/navigation.vm")

#breadcrumbs()

$theme.wrapPortlet("portlet.vm", $content_include)

#foreach ($nav_item in $nav_items)

    $nav_item.getName()

#end
```

Freemarker:

```
<@liferay_util["include"] page=my_template_include />

<#if show_site_name>

    <span title="<@liferay.language_format arguments="${site_name}" key="go-to-x" />">
```

```
        ${site_name}

    </span>

</#if>

<#include "${full_templates_path}/navigation.ftl" />

<@liferay.breadcrumbs />

<@liferay_theme["wrap-portlet"] page="portlet.ftl">

    <@liferay_util["include"] page=content_include />

</@>

<#list nav_items as nav_item>

    ${nav_item.getName()}

</#list>
```

Whichever language you decide to use, there are several convenient macros included with Liferay DXP:

- Freemarker macros on LDN
- Freemarker macros source
- Velocity macros source

Noteworthy Breaking Changes

There are many changes since Liferay Portal 6.1 which break in version 7.0. Some of these changes are documented in Liferay DXP's Breaking Changes. Below are some notable ones.

Dockbar Is Now Called Control Menu In `portal_normal.vm`, admin controls are now called using `#control_menu()`.

Liferay Portal 6.1:

```
#if ($is_signed_in)

    #dockbar()

#end
```

7.0:

```
#control_menu()
```

Portlet Configuration Options In `portlet.vm`, portlet menu icons have been condensed into a single call and are now controlled using the `PortletConfigurationIcon` classes.

Liferay Portal 6.1:

```
$theme.iconOptions()

$theme.iconMinimize()

$theme.iconMaximize()

$theme.iconClose()
```

7.0:

```
$theme.portletIconOptions()
```

See Liferay Portal's Breaking Changes for more details:

- Removed the Tags that Start with portlet:icon-
- Portlet Configuration Options May Not Always Be Displayed

Navigation Item Icons In `navigation.vm`, access to icons have been changed, to minimize dependency on taglibs.

Liferay Portal 6.1:

```
nav_item.icon()
```

7.0:

```
$theme.layoutIcon($nav_item.getLayout())
```

See Liferay Portal's Breaking Changes for more details.

Upgrading Layout Templates

The only significant change to Layout Templates has been the addition of Bootstrap's Grid system.

If you're using any custom Layout Templates, you'll want to integrate Bootstrap into them.

A comparison between Layout Templates in Liferay Portal 6.1 and Liferay Portal 7.0 is shown in the example below:

Liferay Portal 6.1:

```
<div class="portlet-layout">

    <div class="portlet-column portlet-column-only" id="column-1">

        $processor.processColumn("column-1", "portlet-column-content portlet-column-content-only")

    </div>

</div>
```

Liferay Portal 7.0:

```
<div class="portlet-layout row">

    <div class="col-md-12 portlet-column portlet-column-only" id="column-1">

        $processor.processColumn("column-1", "portlet-column-content portlet-column-content-only")

    </div>

</div>
```

Related Topics

Themes Generator

Themelets

Importing Resources with a Theme

15.2 Upgrading Your Theme from Liferay Portal 6.2 to 7.0

This tutorial guides you through the process of upgrading your 6.2 theme to run on 7.0. While you're at it, you should leverage theme improvements, including support for Sass, Bootstrap 3, and Lexicon (Liferay's UI design language). This tutorial demonstrates upgrading a Liferay Portal 6.2 theme to 7.0.

Theme upgrades involve these steps:

- Updating project metadata
- Updating CSS
- Updating theme templates
- Updating resources importer configuration and content
- Applying Lexicon design patterns

As an example, this tutorial applies the steps to a Liferay Portal 6.2 theme called the Lunar Resort theme—developed in the Liferay Portal 6.2 Learning Path Developing a Liferay Theme. It's similar to many Liferay Portal 6.2 themes as it extends the `_styled` theme, adding configurable settings and incorporating a responsive design that leverages Font Awesome icons and Bootstrap. The theme ZIP file contains its original source code.

Before upgrading a theme, consider migrating the theme to use the Liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator. 7.0 doesn't require this migration, but the Liferay JS Theme Toolkit's Gulp `upgrade` task automates many upgrade steps. Themes that use the Liferay JS Theme Toolkit can also leverage exclusive new features, such as the Liferay Theme Generator's sub-generators, and Themelets. If you migrate your theme, return here afterward to upgrade it.

No matter the environment in which you're developing your theme, this tutorial explains everything required to upgrade it. The easiest option is to use the Liferay JS Theme Toolkit's Gulp `upgrade` task, so you'll see that first. Then you'll see *all* upgrade steps, in case you want to run them manually.

Running the Upgrade Task for Themes that Use Liferay JS Theme Toolkit

A Liferay Portal 6.2 theme can be upgraded to 7.0, regardless of its project environment (Liferay JS Theme Toolkit, Plugins SDK, Maven, etc.). But a theme that's been migrated to use the Liferay JS Theme Toolkit can leverage the Gulp `upgrade` task. If you're not going to leverage Liferay JS Theme Toolkit, skip to the *Updating Project Metadata* section.

Here's what the Upgrade task does:

- Updates the theme's Liferay version
- Updates the CSS
- Suggests specific code updates

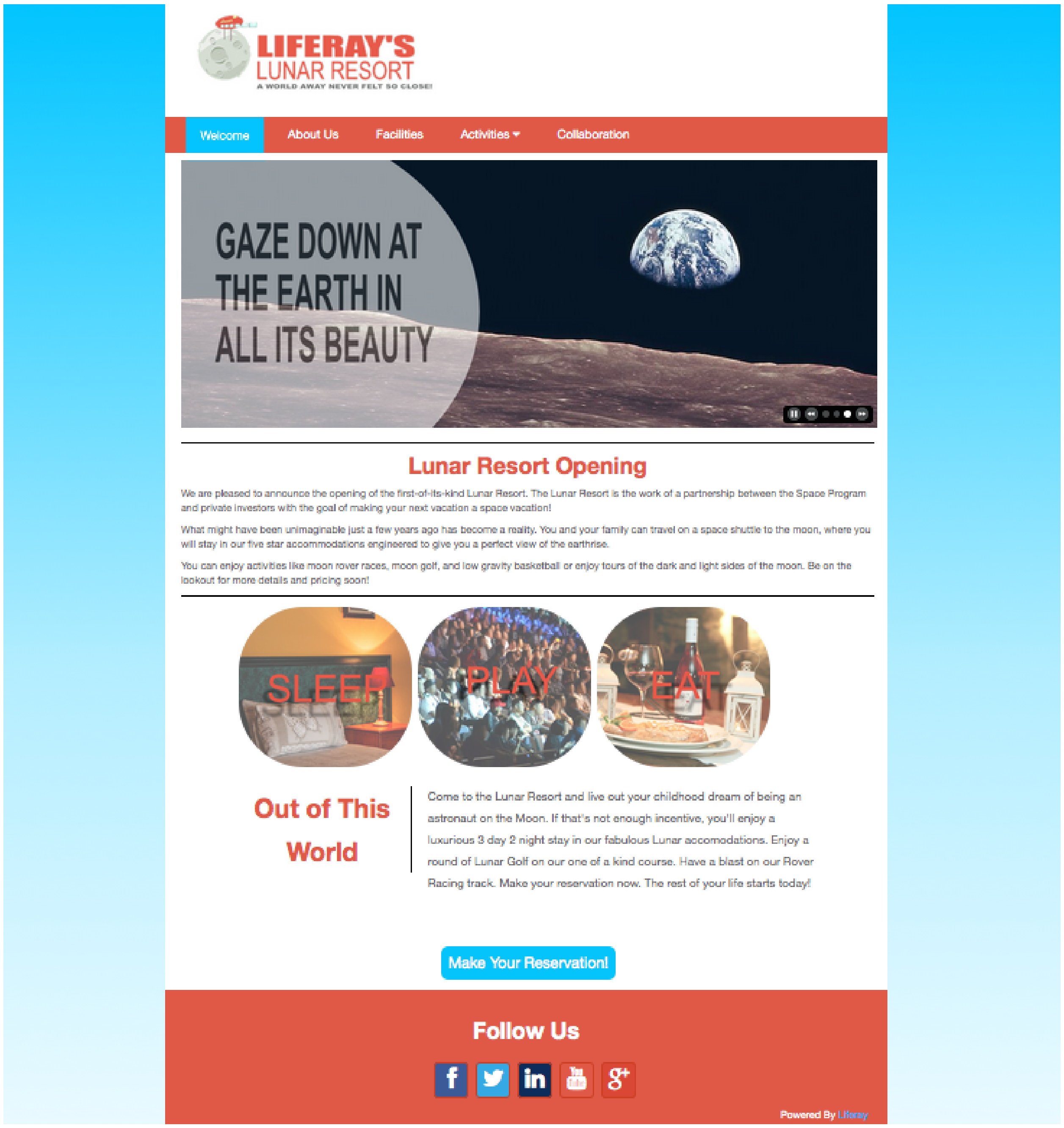
The Upgrade task automatically upgrades CSS code that it can identify. For everything else, it suggests manual upgrades that you can make.

Here are the steps for using the theme `gulp upgrade` task:

1. In your theme's root directory, run this command:

   ```
   gulp upgrade
   ```

   Here's what it does initially:


LIFERAY'S
LUNAR RESORT
A WORLD AWAY NEVER FELT SO CLOSE!
Welcome
About Us
Facilities
Activities
Collaboration
GAZE DOWN AT THE EARTH IN ALL ITS BEAUTY

Lunar Resort Opening

We are pleased to announce the opening of the first-of-its-kind Lunar Resort. The Lunar Resort is the work of a partnership between the Space Program and private investors with the goal of making your next vacation a space vacation!

What might have been unimaginable just a few years ago has become a reality. You and your family can travel on a space shuttle to the moon, where you will stay in our five star accommodations engineered to give you a perfect view of the earthrise.

You can enjoy activities like moon rover races, moon golf, and low gravity basketball or enjoy tours of the dark and light sides of the moon. Be on the lookout for more details and pricing soon!

Out of This World

Come to the Lunar Resort and live out your childhood dream of being an astronaut on the Moon. If that's not enough incentive, you'll enjoy a luxurious 3 day 2 night stay in our fabulous Lunar accomodations. Enjoy a round of Lunar Golf on our one of a kind course. Have a blast on our Rover Racing track. Make your reservation now. The rest of your life starts today!

Figure 15.1: The Lunar Resort example theme upgraded in this tutorial uses a clean, minimal design.

- Copies the existing theme to a folder called `backup`
- Creates core code for generating theme base files
- Updates Liferay version references

```
**Note**: An upgraded theme can be restored to its original state by
executing `gulp upgrade:revert`.
```

```
The task continues upgrading CSS files, prompting you to update CSS file
names.
```

2. For 7.0, Sass files should use the `.scss` extension and file names for Sass partials should start with an underscore (e.g., `_custom.scss`). The `upgrade` task prompts you for each CSS file to rename.

The Upgrade task makes a best effort to upgrade the theme's Bootstrap code from version 2 to 3. For other areas of the code it suspects might need updates, it logs suggestions (covered later). The task also reports changes that may affect theme templates.

A *breaking change* is a code modification between versions of Liferay DXP that might be incompatible with existing plugins, including themes. Liferay minimized the number of breaking changes, but couldn't avoid some. The Breaking Changes reference document describes them. The theme's `gulp upgrade` command and the Upgrade Planner, in Liferay @ide@ identify and address these changes.

The Gulp `upgrade` task jump-starts the upgrade process, but it doesn't complete it. Manual updates are required.

The rest of this tutorial explains all the theme upgrade steps, regardless of whether the Gulp `upgrade` task performs them. Steps the Upgrade task performs are noted in context. Even if you've already executed the Upgrade task, it's best to learn all the steps and make sure they're applied to your theme.

The next step is to update the theme's metadata.

Updating Project Metadata

If you're developing your theme in an environment other than the Plugins SDK, skip this section.

A theme's Liferay version must be updated to `7.0.0+` for the theme to run on 7.0.

If you're using the Plugins SDK, open the `liferay-plugin-package.properties` file and change the `liferay-versions` property value to `7.0.0+`:

```
liferay-versions=7.0.0+
```

If you're using the Liferay JS Theme Toolkit, open the `liferay-look-and-feel.xml` file and specify `liferay-look-and-feel_7_0_0.dtd` as the DTD and `7.0.0+` as the compatibility version:

```
<?xml version="1.0"?>
<!DOCTYPE look-and-feel PUBLIC "-//Liferay//DTD Look and Feel 7.0.0//EN"
"http://www.liferay.com/dtd/liferay-look-and-feel_7_0_0.dtd">

<look-and-feel>
        <compatibility>
                <version>7.0.0+</version>
        </compatibility>

        ...

</look-and-feel>
```

3. If your theme uses the Liferay JS Theme Toolkit and has a `package.json` file, update the file's Liferay version references to `7.0`.

Your theme's Liferay version references are updated for 7.0. Next, you'll update the CSS.

Updating CSS Code

7.0's UI improvements required these CSS-related changes:

- Adding new CSS files
- Removing unneeded CSS files
- Class variable changes (required by Bootstrap 3)
- Modifications to CSS responsiveness tokens

The theme upgrade process involves conforming to these changes.

In this section, you'll update your theme's CSS to leverage the styling improvements. Start with updating CSS file names for Sass.

Updating CSS File Names for Sass

Although Sass was available in Liferay Portal 6.2, only Sass partial files followed the Sass naming convention (using file suffix `.scss`). In 7.0 themes, all Sass files must end in scss.

Note: The Gulp `upgrade` task renames Sass files automatically.

For each CSS file you've modified in your theme, except `main.scss` and `aui.scss`, change its suffix from `.css` to `.scss`.

Then prepend an underscore (_) to all Sass partial file names.

For example, rename `custom.css` to `_custom.scss`.

Here are the Lunar Resort theme's renamed CSS files:

- `css/`
  - `_aui_variables.scss`
  - `_custom.scss`

Refer to the Theme Reference Guide for a complete list of expected theme CSS files.

Next, the CSS rules must be updated to use Bootstrap 3 syntax.

Updating CSS Rules

7.0 uses Bootstrap 3's CSS rule syntax. The new syntax lets developers leverage Bootstrap 3 features and improvements.

If your theme does not use the Liferay JS Theme Toolkit, you can refer to the Migrating from 2.x to 3.0 guide for updating CSS rules to Bootstrap 3.

If your theme uses the Liferay JS Theme Toolkit, the Gulp `upgrade` task reports automatic CSS updates and suggested manual updates. For example, here is part of the task log for the Lunar Resort theme:

```
----------------------------------------------------------------
 Bootstrap Upgrade (2 to 3)
----------------------------------------------------------------

File: src/css/_aui_variables.scss
    Line 5: "$white" has been removed
    Line 11: "$baseBorderRadius" has changed to "$border-radius-base"
    Line 15: "$btnBackground" has changed to "$btn-default-bg"
    Line 16: "$btnBackgroundHighlight" has been removed
    Line 17: "$btnBorder" has changed to "$btn-default-border"
    Line 18: "$btnDangerBackground" has changed to "$btn-danger-bg"
    Line 19: "$btnDangerBackgroundHighlight" has been removed
    Line 21: "$btnInfoBackgroundHighlight" has been removed
    Line 21: "$btnInfoBackground" has changed to "$btn-info-bg"
    Line 22: "$btnPrimaryBackground" has changed to "$btn-primary-bg"
    Line 23: "$btnPrimaryBackgroundHighlight" has been removed
    Line 24: "$btnSuccessBackground" has changed to "$btn-success-bg"
    Line 25: "$btnSuccessBackgroundHighlight" has been removed
    Line 26: "$btnWarningBackground" has changed to "$btn-warning-bg"
    Line 27: "$btnWarningBackgroundHighlight" has been removed
    Line 29: "$dropdownLinkBackgroundActive" has changed to
    "$dropdown-link-active-bg"
    Line 30: "$dropdownLinkBackgroundHover" has changed to
    "$dropdown-link-hover-bg"
    Line 31: "$dropdownLinkColorActive" has changed to
    "$dropdown-link-active-color"
    Line 31: "$white" has been removed
    Line 34: "$navbarBackgroundHighlight" has been removed
    Line 35: "$navbarBorder" has changed to "$navbar-default-border"
    Line 36: "$navbarBackground" has changed to "$navbar-default-bg"
    Line 36: "$navbarLinkBackgroundActive" has changed to
    "$navbar-default-link-active-bg"
    Line 38: "$linkColorHover" has changed to "$link-hover-color"
    Line 38: "$navbarLinkColorHover" has changed to
    "$navbar-default-link-hover-color"
    Line 39: "$navbarLinkColor" has changed to
    "$navbar-default-link-color"
    Line 39: "$navbarText" has changed to "$navbar-default-color"
    Line 41: "$errorBackground" has changed to "$error-bg"
    Line 45: "$infoBackground" has changed to "$info-bg"
    Line 47: "$successBackground" has changed to "$success-bg"
    Line 50: "$warningBackground" has changed to "$warning-bg"
File: src/css/custom.css
    Line 201: Padding no longer affects width or height, you may need to
    change your rule (lines 201-227)
    Line 207: Padding no longer affects width or height, you may need to
    change your rule (lines 207-226)
    Line 212: You would change height from "62px" to "82px"
    Line 305: Padding no longer affects width or height, you may need to
    change your rule (lines 305-314)
    Line 308: You would change height from "39px" to "46px"
    Line 403: "nav-collapse" has changed to "navbar-collapse"
    Line 409: Padding no longer affects width or height, you may need to
    change your rule (lines 409-418)
    Line 490: "btn-navbar" has changed to "navbar-btn"
    Line 490: "btn" has changed to "btn btn-default"
    Line 586: "nav-collapse" has changed to "navbar-collapse"
```

For each update performed and suggested, the task reports a file name and line number range.

Since Bootstrap 3 adopts the `box-sizing: border-box` property for all elements and pseudo-elements (e.g., `:before` and `:after`), padding no longer affects dimensions. Bootstrap's documentation describes the box sizing changes. Consider the padding updates the `upgrade` task reports for CSS rules.

Note: For individual elements, you can overwrite the `box-sizing: border-box` rule with `box-sizing: content-box`.

In all CSS rules that use padding, make sure to update the width and height.

For example, examine the `height` value change in this CSS rule from the Lunar Resort theme's `_custom.scss` file.

Old way:

```
#reserveBtn {
    background-color: #00C4FB;
    border-radius: 10px;
    color: #FFF;
    font-size: 1.5em;
    height: 62px;
    margin: 30px;
    padding: 10px 0;
    ...
}
```

New way:

```
#reserveBtn {
    background-color: #00C4FB;
    border-radius: 10px;
    color: #FFF;
    font-size: 1.5em;
    height: 82px;
    margin: 30px;
    padding: 10px 0;
    ...
}
```

After updating your theme's CSS rules, you should update its CSS responsiveness.

Updating the Responsiveness

In 7.0, Bootstrap 3 explicit media queries replace Bootstrap 2 `respond-to` mixins for CSS responsiveness. Follow these steps to update CSS responsiveness:

1. Open your `_custom.scss` file.

2. Replace all `respond-to` mixins with corresponding media queries shown below:

   Media Query Replacements

Liferay Portal 6.2 Mixin	&nbsp;7.0 Media Query
`@include respond-to(phone)`	`@include media-query(null, $screen-xs-max)`
`@include respond-to(tablet)`	`@include media-query(sm, $screen-sm-max)`
`@include respond-to(phone, tablet)`	`@include media-query(null, $breakpoint_tablet - 1)`
`@include respond-to(desktop, tablet)`	`@include sm`
`@include respond-to(desktop)`	`@include media-query($breakpoint_tablet, null)`

For example, here is a responsiveness update to the Lunar Resort's `_custom.scss` file:

Old:

```
@include respond-to(phone, tablet) {
    html #wrapper #banner #navigation {
    ...
    }
    ...
}
```

New:

```
@include media-query(null, $breakpoint_tablet - 1) {
    html #wrapper #banner #navigation {
    ...
    }
    ...
}
```

The new media query `@include media-query(null, $breakpoint_tablet - 1)` replaces the old mixin `@include respond-to(phone, tablet)`.

The Liferay JS Theme Toolkit's Gulp `upgrade` task generates a file `_deprecated_mixins.scss`. The file provides deprecated compass mixins that your migrated theme might be using. Consider upgrading your use of these mixins. Keep the `_deprecated_mixins.scss` file if you're using any of its mixins, but delete all unused mixins. If you're not using any of the mixins, delete the `_deprecated_mixins.scss` file.

You've updated the theme's responsiveness. Next, you'll update its Font Awesome settings.

Updating Font Awesome Icons

Liferay DXP uses Font Awesome icons extensively. For example, the Lunar Resort theme's design incorporates Font Awesome icons in its social media links.

Figure 15.2: Font Awesome icons facilitate creating social media links.

The icons are easy to use in themes too.

In Liferay Portal 6.2, the CSS file `aui.css` defined the Font Awesome icon paths. In 7.0, the Sass file `_aui_variables.scss` defines them.

Note: In 7.0, the `aui.css` file holds the `lexicon-base` style import. The Theme Reference Guide describes all the Liferay DXP theme files.

The top of the `_aui_variables.scss` file must start with the Font Awesome Icons imports. If you modified the `_aui_variables.scss` file in your Liferay Portal 6.2 theme, add these Font Awesome imports to the top of it:

```
// Icon paths
$FontAwesomePath: "aui/lexicon/fonts/alloy-font-awesome/font";
$font-awesome-path: "aui/lexicon/fonts/alloy-font-awesome/font";
$icon-font-path: "aui/lexicon/fonts/";
```

Next, you'll update the theme templates.

Updating Theme Templates

7.0 theme templates are essentially the same as Liferay Portal 6.2 theme templates. Here are the main changes:

- Velocity templates are now deprecated in favor of FreeMarker templates.
- The Dockbar has been replaced and reorganized into a set of three distinct menus.

Key reasons for using FreeMarker templates and deprecating Velocity templates are these:

- FreeMarker is developed and maintained regularly, while Velocity is no longer actively being developed.
- FreeMarker is faster and supports more sophisticated macros.
- FreeMarker supports using taglibs directly rather than requiring a method to represent them. You can pass body content to them, parameters, etc.

The menus that replace the Dockbar supports a more flexible and responsive design for creating better user experiences.

You should start by addressing the Velocity templates. Since Velocity templates have been deprecated, **you should convert your Velocity theme templates to FreeMarker**.

If you're using the Liferay JS Theme Toolkit, the `gulp upgrade` command reports the required theme template changes in the log.

For example, here is the command's output for the Lunar Resort theme:

```
----------------------------------------------------------------
Liferay Upgrade (6.2 to 7)
----------------------------------------------------------------

File: portal_normal.ftl
    Warning: <@liferay.dockbar /> is deprecated, replace with
    <@liferay.control_menu /> for new admin controls.
    Warning: not all admin controls will be visible without
    <@liferay.control_menu />
    Warning: ${theme} variable is no longer available in FreeMarker
    templates, see https://goo.gl/9fXzYt for more information.
```

For all the theme's templates, it suggests replacement code for deprecated code.

Next, you'll learn how to update various theme templates to 7.0. If you didn't modify any theme templates, you can skip these sections.

Updating Portal Normal FTL

The first one to update is the `portal_normal.ftl` theme template. If you didn't customize `portal_normal.ftl`, you can skip this section.

In FreeMarker templates, the new syntax for including taglibs lets you use them directly rather than accessing them via the `theme` variable. The change is described in the Breaking Changes reference document. All modified `portal_normal.ftl` theme templates must be updated to use the new syntax.

1. Open your modified `portal_normal.ftl` file and replace the following 6.2 directives with the corresponding 7.0 directives:

   FreeMarker Theme Variable Replacements

6.2	7.0
`${theme.include(top_head_include)}`	`<@liferay_util["include"] page=top_head_include />`
`${theme.include(body_top_include)}`	`<@liferay_util["include"] page=body_top_include />`
`${theme.include(content_include)}`	`<@liferay_util["include"] page=content_include />`
`${theme.wrapPortlet("portlet.ftl", content_include)}`	`<@liferay_theme["wrap-portlet"] page="portlet.ftl"> <@liferay_util["include"] page=content_include /> </@>`
`${theme.include(body_bottom_include)}`	`<@liferay_util["include"] page=body_bottom_include />`
`${theme.include(bottom_include)}`	`<@liferay_util["include"] page=bottom_include />`
`${theme.getSetting("my-theme-setting")}`	`${theme_settings["my-theme-setting"]}`
`${theme.runtime("56", "articleId=" + my_article_id)}`	`<@liferay_portlet["runtime"] portletName="com_liferay_journal_content_web_portlet_JournalContentPortlet" queryString="articleId=" + my_article_id />`

2. Replace the following link type.

   Old:

   `<a href="#main-content" id="skip-to-content"><@liferay.language key="skip-to-content" /></a>`

   New:

   `<@liferay_ui["quick-access"] contentId="#main-content" />`.

   The `liferay-ui:quick-access` tag provides a keyboard shortcut to the page's main content.

3. Replace all Dockbar references with Control Menu references.

   Old:

```
<#if is_signed_in>
	<@liferay.dockbar />
</#if>
```

   New:

```
<@liferay.control_menu />
```

   The Dockbar was an all-in-one component that contained the page administration menus and the user/portal administration menus. This UI has since been split and reorganized into three menus:

- *The Product Menu*: Manage site and page navigation, content, settings and pages for the current site, and navigate to user account settings, etc.
- *The Control Menu*: Configure and add content to the page and view the page in a simulation window.
- *The User Personal Bar*: Display notifications and the user's avatar and name.

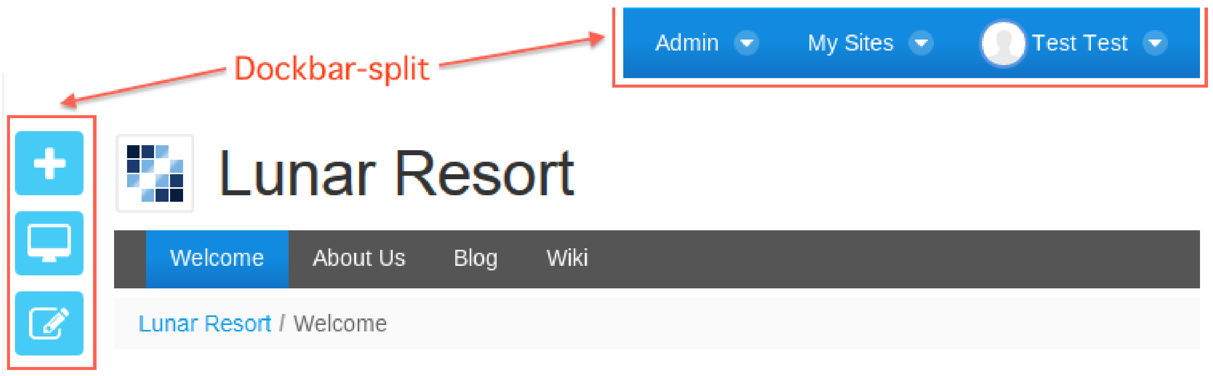


Figure 15.3: The Dockbar was removed in 7.0 and must be replaced with the new Control Menu.

The new design enhances the user experience by providing clear and purposeful menus.

4. If you used the split Dockbar in your Liferay Portal 6.2 theme, remove `dockbar-split` from the `body` element's `class` value.

   For example, remove `dockbar-split` from `<body class="... dockbar-split">`. .

5. Remove the page title code shown below:

```
<h2 class="page-title">
    <span>${the_title}</span>
</h2>
```

   Rather than include the page title on every page, it was decided that this decision should be left up to developers. With the introduction of modularization in themes, this feature can easily be implemented however you like.

6. To ensure navigation is only rendered when there are pages, wrap the `<#include "${full_templates_path}/navigation.f` `/>` include with an `if` statement as demonstrated below:

```
<#if has_navigation && is_setup_complete>
    <#include "${full_templates_path}/navigation.ftl" />
</#if>
```

7. Finally, replace content `div` elements (e.g., `<div id="content">...<div>`) with HTML 5 `section` elements.

   The `div` element works but the `section` element is more accurate and provides better accessibility for screen readers.

   For example, here's a new content `section` element:

```
<section id="content">
    <h1 class="hide-accessible">${the_title}</h1>
    ...
</section>
```

To support accessibility, consider adding an `h1` element like the one above.

Note: The Liferay JS Theme Toolkit's `gulp upgrade` command reports suggested theme template changes.

If you modified the navigation template for your theme, follow the steps in the next section.

Updating Navigation FTL

Follow these steps to update your modified `navigation.ftl` file:

1. Below the `<nav class="${nav_css_class}" id="navigation" role="navigation">` element, add the following hidden heading for accessibility screen readers:

   ```
   <h1 class="hide-accessible">
       <@liferay.language key="navigation" />
   </h1>
   ```

2. To access the layout, add the following variable declaration below the `<#assign nav_item_css_class = "" />` variable declaration:

   ```
   <#assign nav_item_layout = nav_item.getLayout() />
   ```

   This variable grabs the layout for navigation. You can use this variable to retrieve an icon for the navigation menu next.

3. To retrieve an icon for the navigation menu, replace the `${nav_item.icon()}` variable in the `<a aria-labelledby="layout_${nav_item.getLayoutId()}"...</a>` anchor with the following element:

   ```
   <@liferay_theme["layout-icon"] layout=nav_item_layout />
   ```

The navigation template is updated.

That covers most, if not all, of the required theme template changes. If you modified any other FreeMarker theme templates, you can compare them with templates in the `_unstyled` theme. And if your theme uses the Liferay JS Theme Toolkit, refer to the suggested changes that the `gulp upgrade` command reports.

After updating the theme templates, you can update your theme's resources importer code.

Updating the Resources Importer

Liferay's resources importer is now an OSGi module in Liferay's Web Experience application suite. Since the suite is bundled with Liferay DXP, developers no longer need to download the resources importer separately.

API changes and upgrades to Bootstrap 3 affect the following resources importer components:

- Plugin properties
- Web content article files and directory structure
- Sitemap

This section shows you how to update these components.

Note: The example Lunar Resort theme's resources importer web content articles have been modified to avoid known issue LPS-64859. Articles in the Liferay Portal 6.2 theme link to pages in the site's layout. Due to the page and article import order, the links cause a null pointer exception. To avoid this issue with the example theme, the offending links have been removed from its articles.

Start updating the plugin properties for the resources importer.

Updating liferay-plugin-package.properties

If you're upgrading a Plugins SDK theme, follow these instructions to update resources importer properties. Otherwise, skip this section.

Make the following updates to the theme's `liferay-plugin-package.properties` file:

1. Remove the `required-deployment-contexts` property.

   The plugin no longer needs this property as the resources importer is now an OSGi module built-in and deployed with 7.0.

2. Since the group model class's fully-qualified class name has changed, replace the `resources-importer-target-class-name` property's value with the following class name:

   ```
   com.liferay.portal.kernel.model.Group
   ```

Now that the resources importer's properties are configured properly, you can update your theme's web content.

Updating Web Content

All 7.0 web content articles must be written in XML and have a structure and template. Article creation requires a structure and article content rendering requires a template. Follow these steps to update your web content:

1. In the `/resources-importer/journal/articles/` folder, create a subfolder, for example `BASIC_WEB_CONTENT`, to hold the basic HTML articles.

2. Move all basic HTML articles into the folder you just created in step 1.

3. In the `/resources-importer/journal/templates/` and `/resources-importer/journal/structures/` folders, create a subfolder with the same name as the folder you created in step 1.

   For the web content to work properly, the articles, structure, and template folder names must match.

4. In previous Liferay versions, article structures were written in XML. Now they're written in JSON.

   Create a file `[structure-name].json`, for example `BASIC_WEB_CONTENT.json`, in the structure subfolder you created in the previous step.

   For web content articles that use complicated structures and templates, create the structures and templates in Liferay DXP.

5. In the JSON file you just created, add a JSON structure for the web content. For example, you can use a JSON structure like the one below for basic web content articles.

```
{
    "availableLanguageIds": [
        "en_US"
    ],
    "defaultLanguageId": "en_US",
    "fields": [
        {
            "label": {
                "en_US": "Content"
            },
            "predefinedValue": {
                "en_US": ""
            },
            "style": {
                "en_US": ""
            },
            "tip": {
                "en_US": ""
            },
            "dataType": "html",
            "fieldNamespace": "ddm",
            "indexType": "keyword",
            "localizable": true,
            "name": "content",
            "readOnly": false,
            "repeatable": false,
            "required": false,
            "showLabel": true,
            "type": "ddm-text-html"
        }
    ]
}
```

This structure identifies the articles' language and field settings and specifies a name value to identify the content. This example structure's content is named content.

6. In the template subfolder you created in step 3 (e.g., /resources-importer/journal/templates/[template-folder-name]/), create a FreeMarker template file (e.g., [template-folder-name].ftl) and add a method in it to get the article's data.

   For example, this method accesses content from the variable named content and renders it as HTML:

```
${content.getData()}
```

   You've created the basic web content structure and template.

7. Follow this pattern for basic web content articles you convert from HTML to XML:

```
<?xml version="1.0"?>

<root available-locales="en_US" default-locale="en_US">
        <dynamic-element name="content" type="text_area"
        index-type="keyword" index="0">
                <dynamic-content language-id="en_US">
                        <![CDATA[
                        HTML CONTENT GOES HERE
                        ]]>
                </dynamic-content>
        </dynamic-element>
</root>
```

For example, the `2 column description.html` Lunar Resort article's HTML content should be converted to an XML file (e.g., `2 column description.xml`) whose content looks like this:

```
<?xml version="1.0"?>

<root available-locales="en_US" default-locale="en_US">
    <dynamic-element name="content" type="text_area"
    index-type="keyword" index="0">
        <dynamic-content language-id="en_US">
            <![CDATA[
                <div class="container-fluid">
    <div class="span4" id="columnLeft">Out of This World</div>
    <div class="span8" id="columnRight">Come to the Lunar Resort and
    live out your childhood dream of being an astronaut on the Moon. If
    that's not enough incentive, you'll enjoy a luxurious 3 day 2 night
    stay in our fabulous Lunar accommodations. Enjoy a round of Lunar
    Golf on our one of a kind course. Have a blast on our Rover Racing
    track. Make your reservation now. The rest of your life starts
    today!</div>
</div>
]]>
        </dynamic-content>
    </dynamic-element>
</root>
```

8. 7.0's migration from Bootstrap 2 to Bootstrap 3 requires that you replace all `div` element `class` attribute values of Bootstrap 2 format span[`number`] with values that use the Bootstrap 3 format:

```
col-[device-size]-[number]
```

`device-size` can be `xs`, `sm`, `md`, or `lg`. `md` works for most cases. Bootstrap's site at https://getbootstrap.com/docs/3.3/css/#grid explains the Bootstrap 3 grid system

Continuing with the `2 column description.xml` article example, here is its updated content:

```
<?xml version="1.0"?>

<root available-locales="en_US" default-locale="en_US">
    <dynamic-element name="content" type="text_area"
    index-type="keyword" index="0">
        <dynamic-content language-id="en_US">
            <![CDATA[
                <div class="container-fluid">
    <div class="col-md-4" id="columnLeft">Out of This World</div>
    <div class="col-md-8" id="columnRight">Come to the Lunar Resort and
    live out your childhood dream of being an astronaut on the Moon. If
    that's not enough incentive, you'll enjoy a luxurious 3 day 2 night
    stay in our fabulous Lunar accommodations. Enjoy a round of Lunar
    Golf on our one of a kind course. Have a blast on our Rover Racing
    track. Make your reservation now. The rest of your life starts
    today!</div>
</div>
]]>
        </dynamic-content>
    </dynamic-element>
</root>
```

That's all that is needed for most basic web content articles. If you're following along with the Lunar Resort example, the updated XML articles are in the ZIP file's `/resources-importer/journal/articles/Basic Web Content/` folder.

Note: Although Liferay Portal 6.2 used AlloyUI 2.0.x, 7.0 uses AlloyUI 3.0.x. As a result, you may need to update your code that uses AlloyUI. Refer to AlloyUI's examples and API docs for details.

Next, you must update your resources importer's sitemap file.

Updating the Sitemap

In Liferay Portal 6.2, portlet IDs were incremental numbers. In 7.0, they're explicit class names. The new IDs are intuitive and unique. But you must update your `sitemap.json` file with the new portlet IDs.

Some of common portlet IDs are specified in the `sitemap.json` example in the Importing Resources with a Theme tutorial.

You can also retrieve a portlet's ID from the UI:

1. In the portlet's *Options* menu, select *Look and Feel Configuration*.

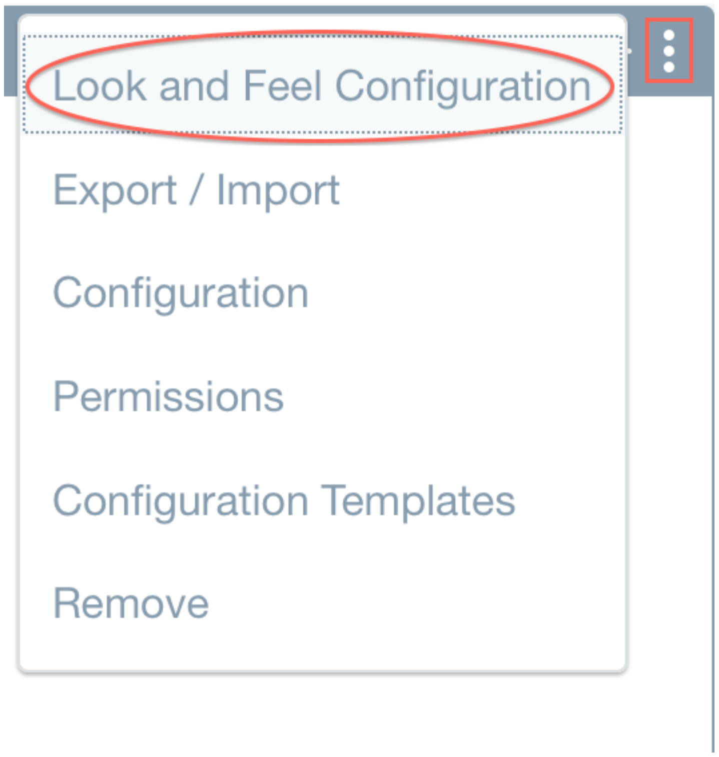


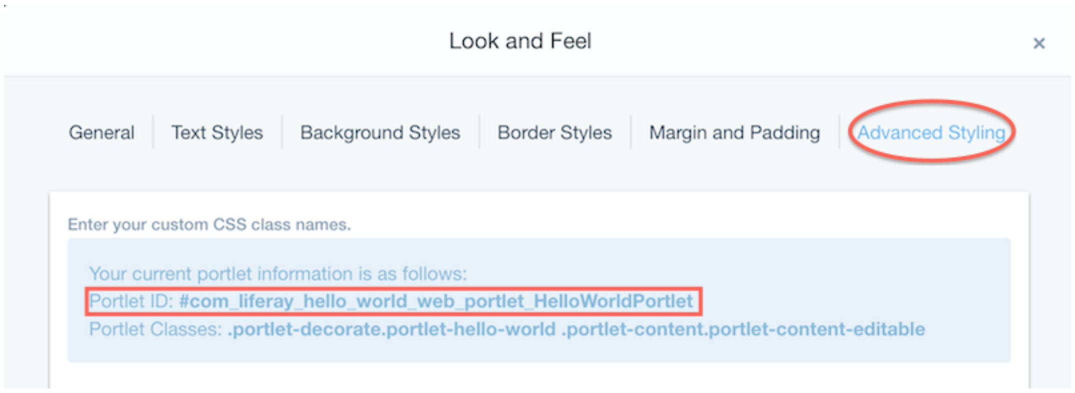
Figure 15.4: You can find the portlet ID in the the *Look and Feel Configuration* menu.

2. Select the *Advanced Styling* tab.

   The `Portlet ID` value is listed in the blue box.

The Portlet ID quick reference lists all the default portlet IDs.

Next, you can learn how to update your theme's UI to follow Lexicon design patterns.



Enter your custom CSS class names.

Your current portlet information is as follows:
Portlet ID: #com_liferay_hello_world_web_portlet_HelloWorldPortlet
Portlet Classes: .portlet-decorate.portlet-hello-world .portlet-content.portlet-content-editable

Figure 15.5: The portlet ID is listed within the blue box in the *Advanced Styling* tab.

Applying Lexicon Design Patterns

7.0 uses Lexicon, a web implementation of Liferay's Lexicon Experience Language. The Lexicon Experience Language provides styling guidelines and best practices for application UIs. While Lexicon's CSS, HTML, and JavaScript components enable developers to build fully-realized UIs quickly and effectively. This section demonstrates how to apply Lexicon to a form.

For example, this is the Liferay Portal 6.2 Lunar Resort's reservation form:

```
<p>
Thanks for choosing to stay at the Liferay Lunar Resort! Please fill out the
form below to book your stay. We know you have a choice in where to stay on
the
Moon... oh wait no you don't. Thanks for picking us anyways. We'll see you
soon on the Moon!
</p>

<form class="form-horizontal">
        <fieldset>
          <legend>Reservation Form</legend>
          <div class="control-group">
              <label class="control-label" for="inputName">Name</label>
              <div class="controls">
                      <input type="text" id="inputName"
                      placeholder="Enter your Name here" required="required">
                  </div>
          </div>
          <div class="control-group">
              <label class="control-label" for="inputEmail">Email</label>
              <div class="controls">
                  <input type="email" id="inputEmail"
                  placeholder="Enter your E-Mail here" required="required">
              </div>
          </div>
          <div class="control-group">
              <div class="controls">
                  <button type="submit" class="btn">Submit</button>
              </div>
          </div>
        </fieldset>
</form>

<p style="padding-bottom:25px;">
Thanks again for booking with Liferay. When you book with Liferay, you
remember your stay. Please take a moment to fill out our guestbook below.
</p>
```

The HTML code above uses Bootstrap 2's markup and CSS classes.

Lexicon extends Bootstrap 3. Here's the Lunar Resort form updated to Lexicon:

```
<?xml version="1.0"?>

<root available-locales="en_US" default-locale="en_US">
        <dynamic-element name="content" type="text_area"
        index-type="keyword" index="0">
                <dynamic-content language-id="en_US">
                        <![CDATA[
                                <p>Thanks for choosing to stay at the
                                Liferay Lunar Resort! Please fill out the
                                form below to book your stay. We know you
                                have a choice in where to stay on the Moon...
                                oh wait no you don't. Thanks for picking us
                                anyways. We'll see you soon on the Moon!</p>
<form role="form-horizontal">
        <fieldset>
          <legend>Reservation Form</legend>
          <div class="form-group">
              <label for="inputName">Name</label>
              <input type="text" id="inputName" class="form-control"
              placeholder="Enter your Name here" required="required">
          </div>
          <div class="form-group">
              <label for="inputEmail">Email</label>
              <input type="email" id="inputEmail" class="form-control"
              placeholder="Enter your E-Mail here" required="required">
          </div>
          <div class="form-group">
                  <button type="submit" class="btn btn-primary">Submit
                  </button>
          </div>
        </fieldset>
</form>

<p style="padding-bottom:25px;">Thanks again for booking with Liferay. When
you book with Liferay, you remember your stay. Please take a moment to fill
out our guestbook below.</p>
                        ]]>
                </dynamic-content>
        </dynamic-element>
</root>
```

The Lexicon updates applied to the form are as follows:

- The `control-group` classes were updated to `form-group` classes.
- The `control-label` classes were removed from the `label` elements.
- The `<div class=""controls>` elements were removed.
- The `form-control` class was added to each input element.
- To emphasize the form's submit button, the `btn-primary` class was added to it.

You can apply similar Lexicon design patterns to your theme's HTML files.

You've updated your theme to 7.0! You can deploy it from your theme project.

Liferay JS Theme Toolkit-based project:

```
gulp deploy
```

Plugins SDK project:

```
ant deploy
```

Now your users can continue enjoying the visual styles you've created in your upgraded themes.

Related Topics

Liferay Theme Generator
Migrating a theme to 7.0
[Upgrading to 7.0] (/docs/7-0/deploy/-/knowledge_base/d/upgrading-to-liferay-7)

15.3 Upgrading Layout Templates

Layout templates for 7.0 differ slightly from layout templates for Liferay Portal 6. The layout template's rows and columns are affected by Bootstrap's new grid system syntax.

This tutorial demonstrates the following:

- How to upgrade your layout template to 7.0

Upgrading a layout template involves updating its Liferay version and updating the class syntax for its rows and columns.

Follow these steps to upgrade your layout template:

1. Open your `liferay-plugin-package.properties` file and update the `liferay-versions` property to `7.0.0+`:

   ```
   liferay-versions=7.0.0+
   ```

2. Open your layout template's `.tpl` file and replace `row-fluid` with `row`, in each row's class value.

3. Previously, column size was denoted using a class value of format `span[number]`. The new Bootstrap grid system uses the format `col-[device-size]-[number]`.

   The `[device-size]` value must be `xs`, `sm`, `md`, or `lg`. In most cases, an `md` device size works well. You can read more about the Bootstrap grid system on their site at https://getbootstrap.com/docs/3.3/css/#gri d.

   The `[number]` value must be an integer from 1 to 12. A row's width is divisible by twelve; so the combined width of a row's columns must equal 12.

   Inside the `.tpl` file, replace each `span-[number]` class value with `col-[device-size]-[number]`, where `[device-size]` is `xs`, `sm`, `md`, or `lg` and `[number]` is an integeter from 1 to 12.

   Here's an example column that uses the `md` device size and a column that is a third (4/12) of the row's total width:

   ```
   <div class="portlet-column portlet-column-last col-md-4" id="column-3">
   ```

As an example, here's Liferay Portal 6 layout template 1_2_1_columns.tpl upgraded to 7.0:

```
<div class="columns-1-2-1" id="main-content" role="main">
        <div class="portlet-layout row">
                <div class="col-md-12 portlet-column portlet-column-only"
                id="column-1">
                        $processor.processColumn("column-1",
                        "portlet-column-content portlet-column-content-only")
                </div>
        </div>

        <div class="portlet-layout row">
```

```
                <div class="col-md-6 portlet-column portlet-column-first"
                id="column-2">
                        $processor.processColumn("column-2",
                        "portlet-column-content portlet-column-content-first")
                </div>

                <div class="col-md-6 portlet-column portlet-column-last"
                id="column-3">
                        $processor.processColumn("column-3",
                        "portlet-column-content portlet-column-content-last")
                </div>
        </div>

        <div class="portlet-layout row">
                <div class="col-md-12 portlet-column portlet-column-only"
                id="column-4">
                        $processor.processColumn("column-4",
                        "portlet-column-content portlet-column-content-only")
                </div>
        </div>
</div>
```

Your layout template is ready to use in 7.0!

Related Topics

Planning Plugin Upgrades and Optimizations
Benefits of 7.0 for Liferay Portal 6 Developers
Liferay Upgrade Planner

15.4 Upgrading Frameworks and Features

Your upgrade process not only relies on portlet technology, themes, and customization plugins, but also the frameworks your project leverages. The following frameworks and their upgrade processes are discussed in this section:

- JNDI data source usage
- Service Builder service invocation
- Service Builder
- Velocity templates

Continue on to learn more about upgrading these frameworks.

15.5 Upgrading JNDI Data Source Usage

In Liferay DXP's OSGi environment, you must use the portal's class loader to load the application server's JNDI classes. An OSGi bundle's attempt to connect to a JNDI data source without using Liferay DXP's class loader results in a `java.lang.ClassNotFoundException`.

For more information on how to do this, see the Connecting to JNDI Data Sources article.

15.6 Upgrading Service Builder Service Invocation

When upgrading a portlet leveraging Service Builder, you must first decide if you're building your Service Builder logic as a WAR or modularizing it.

Note: Service Builder portlets automatically migrated to Liferay Workspace using the Upgrade Planner or Blade CLI's `convert` command automatically have its Service Builder logic converted to API and implementation modules. This is a best practice for 7.0.

If you prefer keeping your Service Builder logic as a WAR, you must implement a service tracker to call services. See the Service Trackers article for more information.

If you're optimizing your Service Builder logic to invoke Liferay services from a module, see the Invoking Local Services article for more information.

15.7 Upgrading Service Builder

7.0 continues to use Service Builder, so you can focus on your application's business logic instead of its persistence details. It still generates model classes, local and remote services, and persistence.

Upgrading most Service Builder portlets involves these steps:

1. Adapt the code to 7.0's API
2. Resolve dependencies
3. Build the services

Start by adapting the code.

Step 1: Adapt the Code to 7.0's API

Adapt the portlet to 7.0's API using the Upgrade Planner. When running the planner's *Fix Upgrade Problems* step, many of the existing issues are autocorrected. For remaining issues, the planner identifies code affected by the new API and ways to adapt it.

For example, consider an example portlet with the following compilation error:

```
/html/guestbook/view.jsp(58,1) PWC6131: Attribute total invalid for tag search-container-results according to TLD
```

The `view.jsp` file specifies a tag library attribute `total` that doesn't exist in 7.0's `liferay-ui` tag library. Notice the second attribute `total`.

```
<liferay-ui:search-container-results
    results="<%=EntryLocalServiceUtil.getEntries(scopeGroupId,
                    guestbookId, searchContainer.getStart(),
                    searchContainer.getEnd())%>"
    total="<%=EntryLocalServiceUtil.getEntriesCount(scopeGroupId,
                    guestbookId)%>" />
```

Remove the `total` attribute assignment to make the tag like this:

```
<liferay-ui:search-container-results
    results="<%=EntryLocalServiceUtil.getEntries(scopeGroupId,
                    guestbookId, searchContainer.getStart(),
                    searchContainer.getEnd())%>" />
```

Resolve these error types and others until your code is adapted to the new API.

Step 2: Resolve Dependencies

To adapt your app's dependencies, refer to the Resolving a Plugin's Dependencies tutorial. Once your dependencies are upgraded, rebuild your services!

Step 3: Build the Services

To rebuild your portlet's services, see the Running Service Builder article.

An example change where upgrading legacy Service Builder code can produce differing results is explained below.

A Liferay Portal 6.2 portlet's `service.xml` file specifies exception class names in `exception` elements like this:

```
<service-builder package-path="com.liferay.docs.guestbook">
    ...
    <exceptions>
        <exception>GuestbookName</exception>
        <exception>EntryName</exception>
        <exception>EntryMessage</exception>
        <exception>EntryEmail</exception>
    </exceptions>
</service-builder>
```

In Liferay Portal 6.2, Service Builder generates exception classes to the path attribute `package-path` specifies. In 7.0, Service Builder generates them to `[package-path]/exception`.

Old path:

```
[package-path]
```

New path:

```
[package-path]/exception
```

For example, the example portlet's package path is `com.liferay.docs.guestbook`. Its exception class for `exception` element `GuestbookName` is generated to `docroot/WEB-INF/service/com/liferay/docs/guestbook/exception`. Classes that use the exception must import `com.liferay.docs.guestbook.exception.GuestbookNameException`. If this upgrade is required in your Service Builder project, you must update the references to your portlet's exception classes.

Once your Service Builder portlet is upgraded, deploy it.

Note: Service Builder portlets automatically migrated to Liferay Workspace using the Upgrade Planner or Blade CLI's `convert` command automatically has its Service Builder logic converted to API and implementation modules. This is a best practice for 7.0.

The portlet is now available on Liferay DXP. Congratulations on upgrading a portlet that uses Service Builder!

15.8 Migrating Off of Velocity Templates

Velocity templates were deprecated in Liferay Portal 7.0 and are now removed in favor of FreeMarker templates in 7.0. Below are the key reasons for this move:

- FreeMarker is developed and maintained regularly, while Velocity is no longer actively being developed.
- FreeMarker is faster and supports more sophisticated macros.
- FreeMarker supports using taglibs directly rather than requiring a method to represent them. You can pass body content to them, parameters, etc.

Although Velocity templates still work in 7.0, we highly recommend migrating to FreeMarker templates. For more information on this topic, see the Upgrading Layout Templates section.

Chapter 16

Upgrading Portlet Plugins

All portlet plugin types developed for Liferay Portal 6 can be upgraded and deployed to 7.0.

Upgrading most portlets involves these steps:

1. Adapt the code to 7.0's API
2. Resolve dependencies

Liferay's Upgrade Planner helps you adapt your code to 7.0's API. And resolving a portlet's dependencies is straightforward. In most cases, after you finish the above steps, you can deploy your portlet to Liferay DXP.

The portlet upgrade tutorials show you how to upgrade the following common portlets:

- GenericPortlet
- Servlet-based portlet
- Liferay MVC Portlet
- Portlet that uses Service Builder
- Liferay JSF Portlet
- Struts Portlet
- Spring MVC Portlet

The tutorials provide example portlet source code from before and after upgrading the example portlet. Each tutorial's steps were applied to the example portlet. You can refer to example code as you upgrade your portlet.

Let's get your portlet running on 7.0!

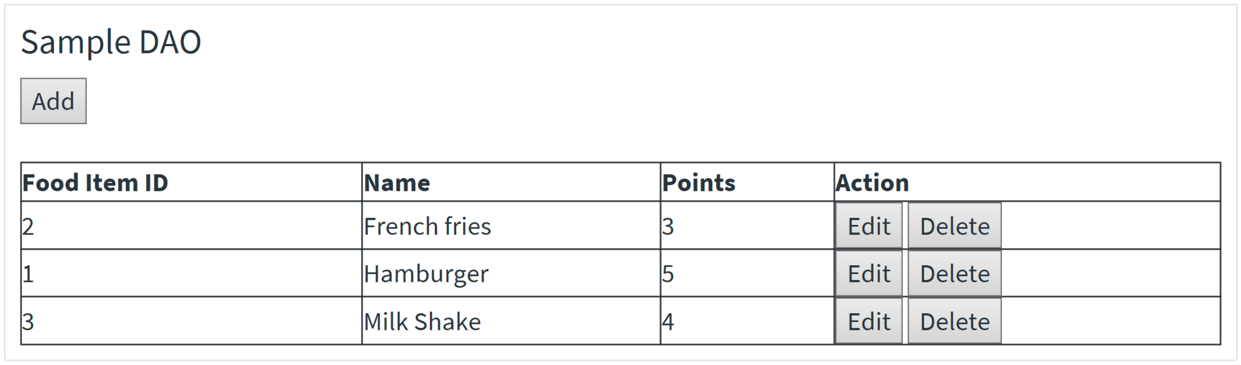
16.1 Upgrading a GenericPortlet

It's common to create portlets that extend `javax.portlet.GenericPortlet`. After all, `GenericPortlet` provides a default `javax.portlet.Portlet` interface implementation. Upgrading a `GenericPortlet` is straightforward and takes only two steps:

1. Adapt the portlet to 7.0's API using the Upgrade Planner.

2. Resolve its dependencies.

This tutorial demonstrates upgrading a Liferay Plugins SDK 6.2 sample portlet called *Sample DAO* (project `sample-dao-portlet`).



Sample DAO

Add

Food Item ID	Name	Points	Action
2	French fries	3	Edit Delete
1	Hamburger	5	Edit Delete
3	Milk Shake	4	Edit Delete

Figure 16.1: The `sample-dao-portlet` lets users manage food items.

The sample portlet lets users view, add, edit, and delete food items from a listing. For reference, you can download the pre-upgraded portlet code and the upgraded code.

The sample portlet has the following characteristics:

- Extends `GenericPortlet`
- View layer implemented by JSPs
- Persists models using the Data Access Object (DAO) design pattern
- Specifies database connection information in a properties file
- Manages dependencies via Ant/Ivy
- Developed in a Liferay Plugins SDK 6.2

The portlet uses a traditional Plugins SDK portlet project folder structure.

Upgrading most `GenericPortlet` portlets involves these steps:

1. Adapt the code to 7.0's API
2. Resolve dependencies

Since the sample portlet's dependencies haven't changed, upgrading it involves only adapting the code to 7.0's API. The Upgrade Planner facilitates updating the code and resolving compilation issues quickly.

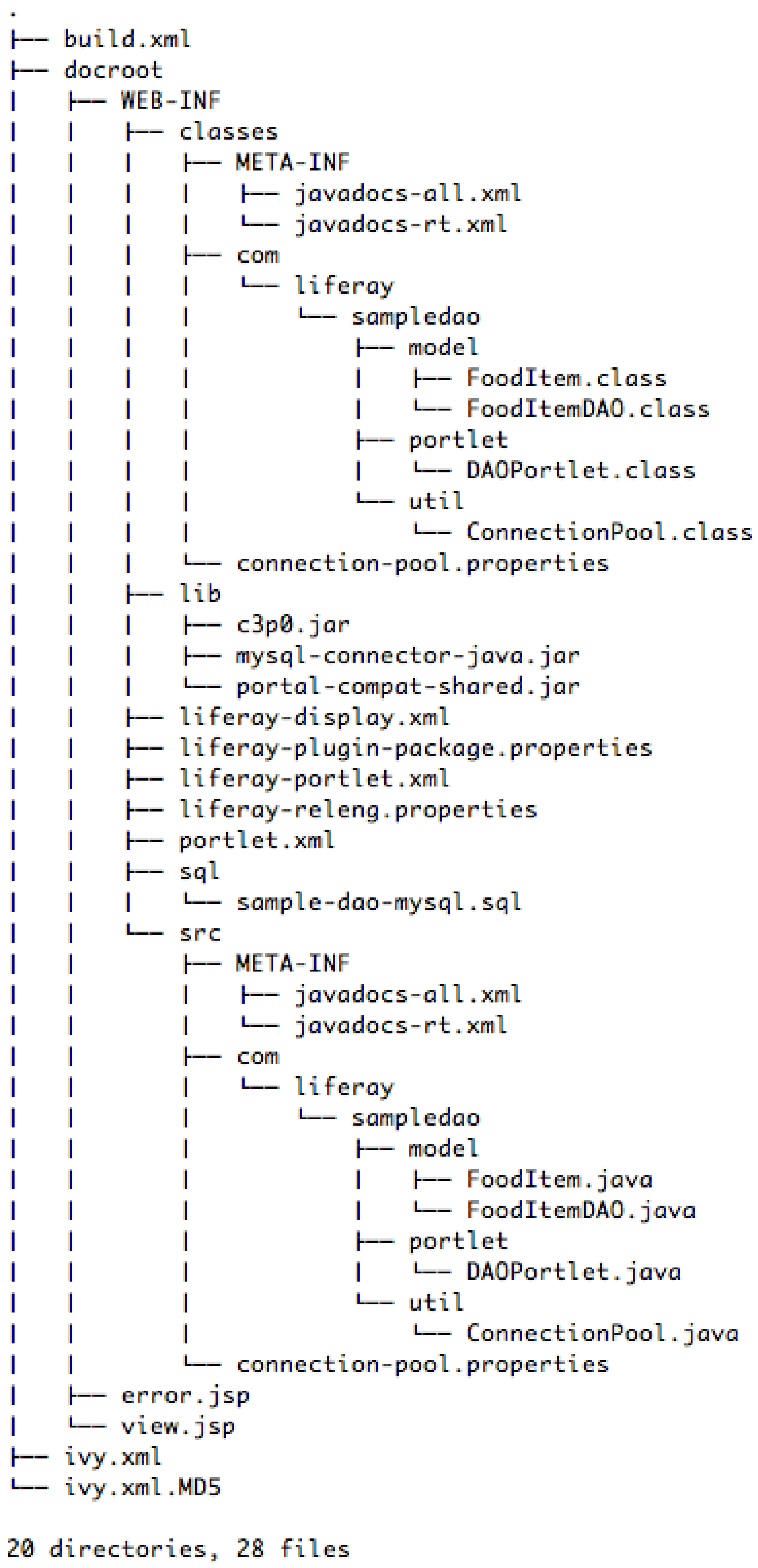
Note: Refer to tutorial Resolving a Plugin's Dependencies if you need to adapt to dependency changes.

You deploy a `GenericPortlet` to 7.0 in the same way you deploy to Portal 6.x. When the plugin WAR file lands in the `[Liferay_Home]/deploy` folder, Liferay DXP's Plugin Compatibility Layer converts the WAR to a Web Application Bundle (WAB) and installs the portlet as a WAB to Liferay DXP's OSGi runtime.

On deploying an upgraded portlet, the server prints messages that indicate the following portlet status:

- WAR processing
- WAB startup
- Availability to users

Deploying the sample portlet produces messages like these:



```
.
├── build.xml
├── docroot
│   ├── WEB-INF
│   │   ├── classes
│   │   │   ├── META-INF
│   │   │   │   ├── javadocs-all.xml
│   │   │   │   └── javadocs-rt.xml
│   │   │   ├── com
│   │   │   │   └── liferay
│   │   │   │       └── sampledao
│   │   │   │           ├── model
│   │   │   │           │   ├── FoodItem.class
│   │   │   │           │   └── FoodItemDAO.class
│   │   │   │           ├── portlet
│   │   │   │           │   └── DAOPortlet.class
│   │   │   │           └── util
│   │   │   │               └── ConnectionPool.class
│   │   │   └── connection-pool.properties
│   │   ├── lib
│   │   │   ├── c3p0.jar
│   │   │   ├── mysql-connector-java.jar
│   │   │   └── portal-compat-shared.jar
│   │   ├── liferay-display.xml
│   │   ├── liferay-plugin-package.properties
│   │   ├── liferay-portlet.xml
│   │   ├── liferay-releng.properties
│   │   ├── portlet.xml
│   │   ├── sql
│   │   │   └── sample-dao-mysql.sql
│   │   └── src
│   │       ├── META-INF
│   │       │   ├── javadocs-all.xml
│   │       │   └── javadocs-rt.xml
│   │       ├── com
│   │       │   └── liferay
│   │       │       └── sampledao
│   │       │           ├── model
│   │       │           │   ├── FoodItem.java
│   │       │           │   └── FoodItemDAO.java
│   │       │           ├── portlet
│   │       │           │   └── DAOPortlet.java
│   │       │           └── util
│   │       │               └── ConnectionPool.java
│   │       └── connection-pool.properties
│   ├── error.jsp
│   └── view.jsp
├── ivy.xml
└── ivy.xml.MD5

20 directories, 28 files
```

Figure 16.2: The `sample-dao-portlet` project uses a typical Plugins SDK portlet folder structure

```
20:57:02,571 INFO ... [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:252] Processing sample-dao-portlet-
7.0.0.1.war
...
20:57:12,639 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][BundleStartStopLogger:35] STARTED sample-
dao-portlet_7.0.0.1 [996]
...
20:57:13,480 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][PortletHotDeployListener:313] 1 portlet for sample-
dao-portlet is available for use
```

The portlet is now available on Liferay DXP.

You've learned how to upgrade and deploy a portlet that extends `GenericPortlet`. You adapt the code, resolve dependencies, and deploy the portlet as you always have. It's just that easy!

Related Topics

Migrating Plugins SDK Projects to Workspace and Gradle

Using Dependency Management Tools

Using the WAB Generator

Migrating Data Upgrade Processes

16.2 Upgrading a Servlet-based Portlet

Servlet-based portlets have little overhead and are easy to upgrade. This tutorial shows you how to upgrade them and refers to code from before and after upgrading a sample servlet-based portlet called *Sample JSON* (project `sample-json-portlet`). The portlet shows a *Click me* link. When users click the link, the Liferay logo appears.

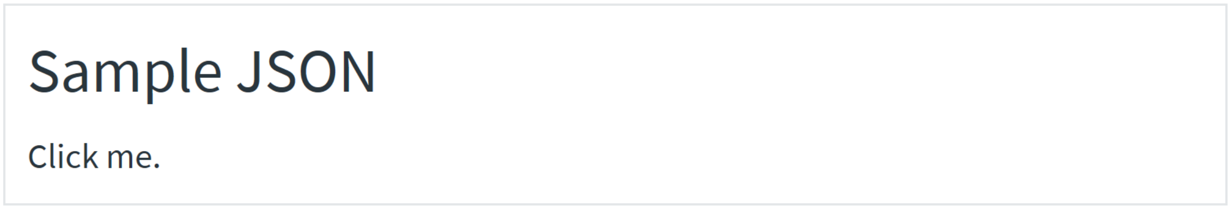


Figure 16.3: The Sample JSON portlet displays text stating *Click me* that you can click to initiate an action.

To get the most from this tutorial, you can download and refer to the original sample portlet source code and the upgraded source code.

Here are the sample portlet's characteristics:

- Processes requests using a servlet that extends `javax.servlet.HttpServlet`
- View layer implemented by JSPs
- Processes data using JSON objects
- Relies on manual dependency management
- Depends on third-party libraries that Liferay Portal 6.2 provides
- Embeds additional dependencies in its `WEB-INF/lib` folder
- Developed in a Liferay Plugins SDK 6.2

Follow these steps to upgrade a servlet-based portlet:

1. Adapt the code to 7.0's API

2. Resolve dependencies

The Upgrade Planner makes adapting a portlet's code straightforward, and it automates much of the process.

The sample portlet relied on Liferay Portal to provide several dependency JAR files. Here's the `portal-dependency-jars` property from the portlet's `liferay-plugin-package.properties` file:

```
portal-dependency-jars=\
    dom4j.jar,\
    jabsorb.jar,\
    json-java.jar
```

Instructions for using packages that Liferay DXP exports are found here. 7.0's core system exports the package this portlet needs from each of the above dependency JARs.

The upgraded sample portlet continues to specify these JARs in the `portal-dependency-jars` property. They're made available to the portlet at compile time. But to avoid packages from compile time conflicting with the core system's exported packages, the Liferay Plugins SDK 7.0 excludes the JARs from the plugin WAR.

Next, deploy your portlet as you always have.

The server prints messages that indicate the following portlet status:

- WAR processing
- WAB startup
- Availability to users

Note: On deploying the sample upgraded portlet, the WAB processor warns that the `portal-dependency-jars` property is deprecated.

```
 21:40:25,347 WARN  [fileinstall-...][WabProcessor:564] The property "portal-dependency-jars" is deprecated. Specified JARs may not be included in the class
```

For running on 7.0, it's fine to specify the`portal-dependency-jars` property per the instructions for using packages that @portal@ exports. After upgrading, consider using a dependency management tool in your project. This helps prepare it for future Liferay DXP versions and facilitates managing dependencies.

The portlet is installed to Liferay DXP's OSGi runtime and is available to users.

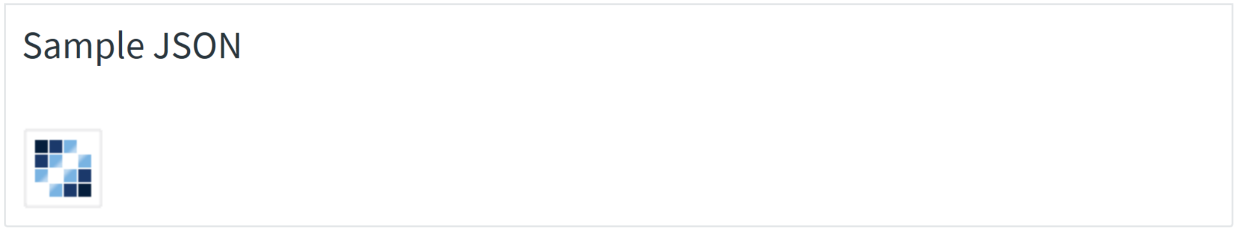


Figure 16.4: Clicking on the sample portlet's *Click me* link shows the Liferay logo.

Congratulations! You've upgraded and deployed your servlet-based portlet to 7.0.

Related Topics

Migrating Plugins SDK Projects to Workspace and Gradle

Using Dependency Management Tools
Using the WAB Generator
Migrating Data Upgrade Processes

16.3 Upgrading a Liferay MVC Portlet

Liferay's MVC Portlet framework is used extensively in Liferay's portlets and is a popular choice for Liferay Portal 6.2 portlet developers. The `MVCPortlet` class is a lightweight extension of `javax.portlet.GenericPortlet`. Its `init` method saves you from writing a lot of boilerplate code. MVC portlets can upgraded to 7.0 without a hitch.

Upgrading a Liferay MVC Portlet involves these steps:

1. Adapt the code to 7.0's API

2. Resolve dependencies

Liferay's Upgrade Planner identifies code affected by the new API, explains the API changes and how to adapt to them, and in many cases, provides options for adapting the code automatically.

After upgrading your portlet, deploy it the same way you always do.

The server prints messages that indicate the following portlet status:

- WAR processing
- WAB startup
- Availability to users

You've upgraded and deployed your Liferay MVC Portlet on your 7.0 instance. Have fun showing off your upgraded portlet!

Related Topics
Migrating Plugins SDK Projects to Workspace and Gradle
Using Dependency Management Tools
Using the WAB Generator
Migrating Data Upgrade Processes

16.4 Upgrading Portlets that use Service Builder

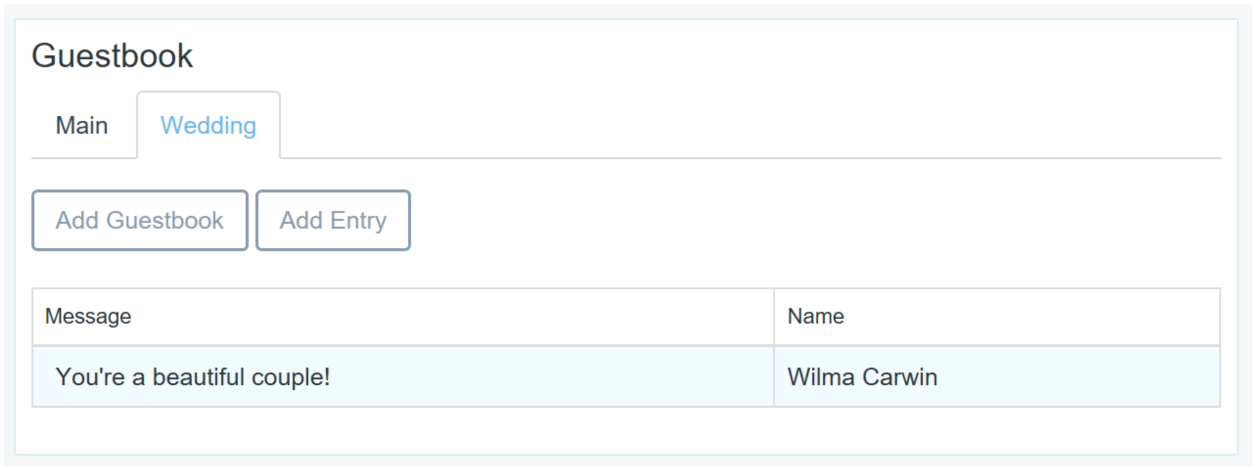
7.0 continues to use Service Builder, so you can focus on your application's business logic instead of its persistence details. It still generates model classes, local and remote services, and persistence.

This tutorial demonstrates upgrading a Liferay Plugins SDK 6.2 portlet called Guestbook portlet (project `guestbook-portlet`). It's from the Writing a Data-Driven Application section of the Liferay Portal 6.2 Learning Path Writing a Liferay MVC Application.

To get the most from this tutorial, you can download and refer to the original portlet source code and the upgraded source code.

The Guestbook portlet has the following characteristics:

- Extends `MVCPortlet`
- Separate Model, View, and Controller layers
- Persistence by Hibernate under Service Builder
- View layer implemented by JSPs



Message	Name
You're a beautiful couple!	Wilma Carwin

Figure 16.5: The Guestbook portlet to model guestbooks and guestbook entries.

- Relies on manual dependency management
- Developed in a Liferay Plugins SDK 6.2

Upgrading most Service Builder Portlets involves these steps:

1. Adapt the code to 7.0's API
2. Resolve dependencies
3. Build the services

Start by adapting the code.

1. Adapt the code to 7.0's API

Use the Upgrade Planner to update the code and resolve compilation issues quickly. Then fix any remaining compilation errors manually.

The Guestbook portlet has the following compilation error:

```
/html/guestbook/view.jsp(58,1) PWC6131: Attribute total invalid for tag search-container-results according to TLD
```

The `view.jsp` file specifies a tag library attribute `total` that doesn't exist in 7.0's `liferay-ui` tag library. Notice the second attribute `total`.

```
<liferay-ui:search-container-results
    results="<%=EntryLocalServiceUtil.getEntries(scopeGroupId,
                    guestbookId, searchContainer.getStart(),
                    searchContainer.getEnd())%>"
    total="<%=EntryLocalServiceUtil.getEntriesCount(scopeGroupId,
                    guestbookId)%>" />
```

Remove the total attribute assignment to make the tag like this:

```
<liferay-ui:search-container-results
    results="<%=EntryLocalServiceUtil.getEntries(scopeGroupId,
                    guestbookId, searchContainer.getStart(),
                    searchContainer.getEnd())%>" />
```

That's the Guestbook portlet's only compilation error you need to fix manually.

2. Resolve dependencies

Since the Guestbook portlet's dependencies haven't changed, there aren't any dependencies to resolve.
If you need to adapt a portlet's dependencies, refer to tutorial Resolving a Plugin's Dependencies.

3. Build the services

Build the services as you did in Liferay Portal 6.2.
The Guestbook portlet's `service.xml` file specifies exception class names in `exception` elements.

```
<service-builder package-path="com.liferay.docs.guestbook">
    ...
    <exceptions>
        <exception>GuestbookName</exception>
        <exception>EntryName</exception>
        <exception>EntryMessage</exception>
        <exception>EntryEmail</exception>
    </exceptions>
</service-builder>
```

In Liferay Portal 6.2, Service Builder generates exception classes to the path attribute `package-path` specifies. In 7.0, Service Builder generates them to `[package-path]/exception`.
Old path:

```
[package-path]
```

New path:

```
[package-path]/exception
```

For example, the Guestbook portlet's package path is `com.liferay.docs.guestbook`. Its exception class for `exception` element `GuestbookName` is generated to `docroot/WEB-INF/service/com/liferay/docs/guestbook/exception`. Classes that use the exception must import `com.liferay.docs.guestbook.exception.GuestbookNameException`.
Update references to your portlet's exception classes.
Deploy the portlet as you normally would. The server prints messages indicating the following portlet status:

- WAR processing
- WAB startup
- Availability to users

Deploying the Guestbook portlet produces these messages:

```
19:36:27,591 INFO  [RMI TCP Connection(27)-192.168.1.110][BaseAutoDeployListener:42] Copying portlets for C:\portals\liferay-ce-portal-
7.0-ga3-upgrading-portlets\tomcat-8.0.32\temp\20170710193627556LFQNRFGO\guestbook-portlet-7.0.0.1.war
19:36:27,973 INFO  [RMI TCP Connection(27)-192.168.1.110][BaseDeployer:873] Deploying guestbook-portlet-7.0.0.1.war
19:36:29,449 INFO  [RMI TCP Connection(27)-192.168.1.110][BaseAutoDeployListener:50] Portlets for C:\portals\liferay-ce-portal-7.0-ga3-
upgrading-portlets\tomcat-8.0.32\temp\20170710193627556LFQNRFGO\guestbook-portlet-7.0.0.1.war copied successfully
19:36:31,231 INFO  [pool-23-thread-2][BundleStartStopLogger:35] STARTED guestbook-portlet_7.0.0.1 [496]
19:36:31,459 INFO  [pool-23-thread-2][HotDeployImpl:226] Deploying guestbook-portlet from queue
19:36:31,459 INFO  [pool-23-thread-2][PluginPackageUtil:1006] Reading plugin package for guestbook-portlet
10-Jul-2017 19:36:31.470 INFO [pool-23-thread-2] org.apache.catalina.core.ApplicationContext.log Initializing Spring root WebApplicationContext
19:36:31,934 INFO  [pool-23-thread-2][PortletHotDeployListener:202] Registering portlets for guestbook-portlet
19:36:32,008 INFO  [pool-23-thread-2][PortletHotDeployListener:331] 1 portlet for guestbook-portlet is available for use
```

The portlet is now available on Liferay DXP.
Congratulations on upgrading and deploying a portlet that uses Service Builder.

Related Topics

Migrating Plugins SDK Projects to Workspace and Gradle
Using Dependency Management Tools
Using the WAB Generator
Migrating Data Upgrade Processes

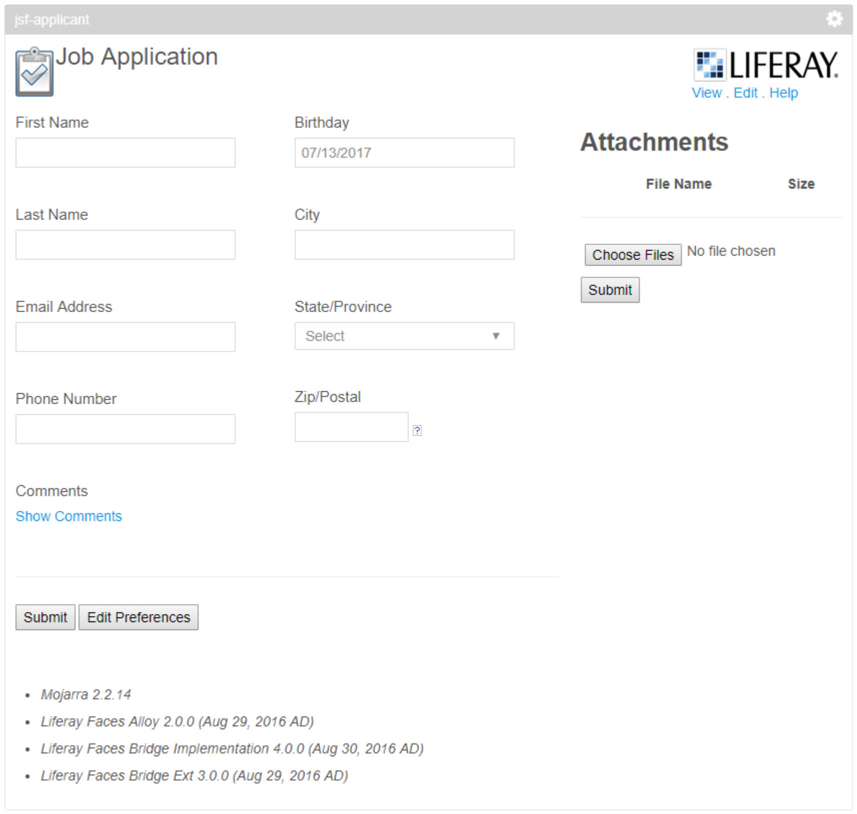
16.5 Upgrading a Liferay JSF Portlet

Liferay JSF portlets are easy to upgrade and require few changes. They interface with the Liferay Faces project, which encapsulates Liferay DXP's Java API and JavaScript code. Because of this, upgrading JSF portlets to 7.0 requires only updating dependencies.

There are two ways to find a JSF portlet's dependencies for 7.0:

- The http://liferayfaces.org/ home page lets you look up the dependencies (Gradle or Maven) by Liferay DXP version, JSF version, and component suites.
- The Liferay Faces Version Scheme article's tables list artifacts by Liferay DXP version, JSF version, portlet version, and AlloyUI and Metal component suite version.

In this tutorial, you'll see how easy it is to upgrade a Liferay Portal 6.2 JSF portlet (JSF 2.2) to 7.0 by upgrading the sample JSF Applicant portlet. This portlet provides a job application users can submit.



- Mojarra 2.2.14
- Liferay Faces Alloy 2.0.0 (Aug 29, 2016 AD)
- Liferay Faces Bridge Implementation 4.0.0 (Aug 30, 2016 AD)
- Liferay Faces Bridge Ext 3.0.0 (Aug 29, 2016 AD)

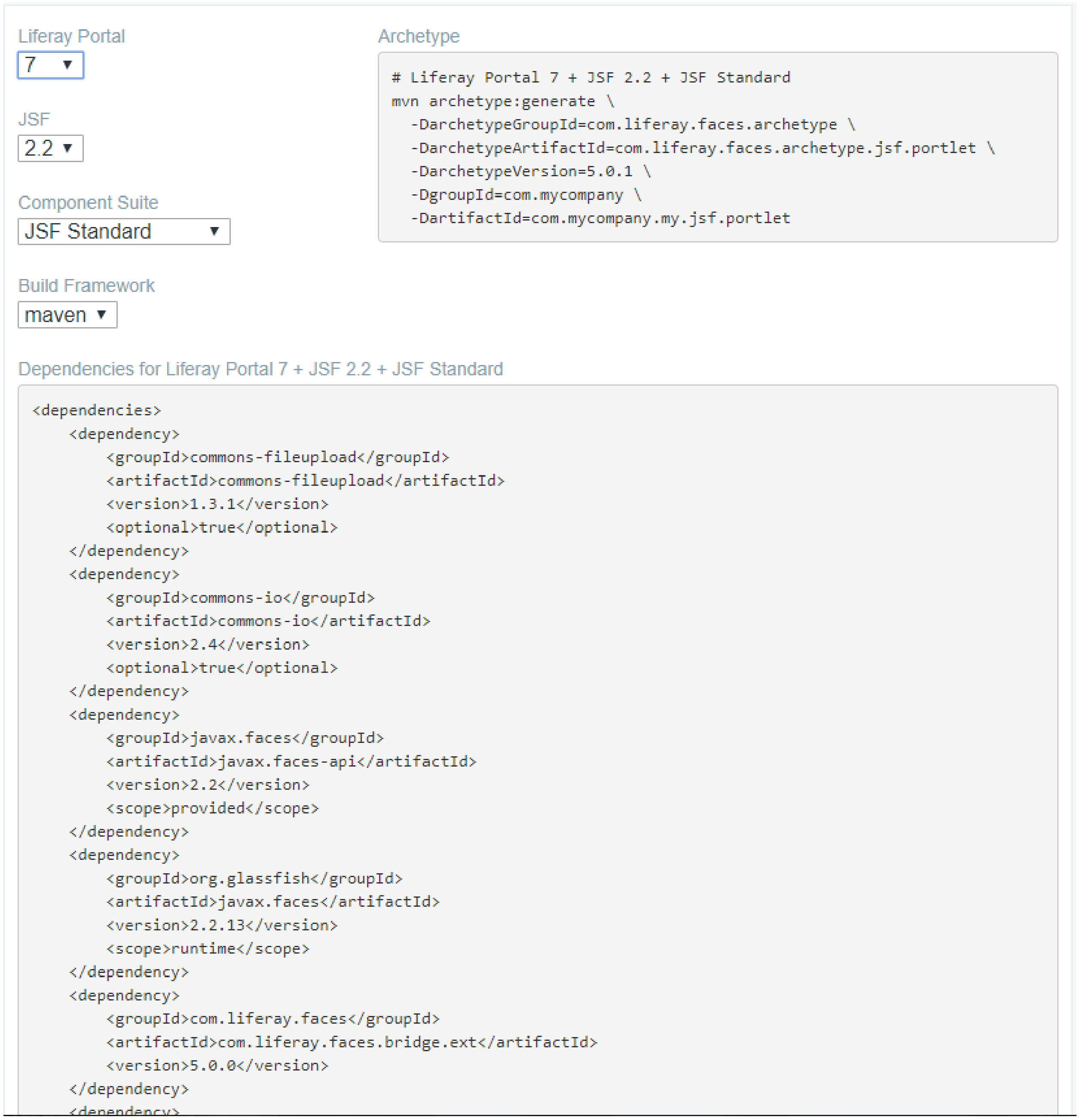
Figure 16.6: The JSF Applicant portlet provides a job application for users to submit.

For reference, you can download the pre-upgraded portlet code and the upgrade portlet code. This sample project uses Maven.

Follow these steps to upgrade your Liferay JSF portlet.

1. Open your Liferay JSF portlet's build file (e.g., `pom.xml`, `build.gradle`) to where the dependencies are configured.

2. Navigate to the http://liferayfaces.org/ site and generate a dependency list by choosing the environment to which you want to upgrade your portlet.



Liferay Portal: 7

JSF: 2.2

Component Suite: JSF Standard

Build Framework: maven

Archetype

```
# Liferay Portal 7 + JSF 2.2 + JSF Standard
mvn archetype:generate \
  -DarchetypeGroupId=com.liferay.faces.archetype \
  -DarchetypeArtifactId=com.liferay.faces.archetype.jsf.portlet \
  -DarchetypeVersion=5.0.1 \
  -DgroupId=com.mycompany \
  -DartifactId=com.mycompany.my.jsf.portlet
```

Dependencies for Liferay Portal 7 + JSF 2.2 + JSF Standard

```
<dependencies>
    <dependency>
        <groupId>commons-fileupload</groupId>
        <artifactId>commons-fileupload</artifactId>
        <version>1.3.1</version>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>commons-io</groupId>
        <artifactId>commons-io</artifactId>
        <version>2.4</version>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>javax.faces</groupId>
        <artifactId>javax.faces-api</artifactId>
        <version>2.2</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.glassfish</groupId>
        <artifactId>javax.faces</artifactId>
        <version>2.2.13</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>com.liferay.faces</groupId>
        <artifactId>com.liferay.faces.bridge.ext</artifactId>
        <version>5.0.0</version>
    </dependency>
    <dependency>
```

Figure 16.7: The Liferay Faces site gives you options to generate dependencies for many environments.

3. Compare the generated dependencies with your portlet's dependencies and make any necessary updates. For the sample JSF Applicant portlet, only one Liferay Faces dependency requires updating:

   ```
   <dependency>
       <groupId>com.liferay.faces</groupId>
       <artifactId>com.liferay.faces.bridge.ext</artifactId>
       <version>3.0.0</version>
   </dependency>
   ```

   The Liferay Faces Bridge EXT dependency must be updated to version 5.0.0 for 7.0. The updated dependency should look like this:

   ```
   <dependency>
       <groupId>com.liferay.faces</groupId>
       <artifactId>com.liferay.faces.bridge.ext</artifactId>
       <version>5.0.0</version>
   </dependency>
   ```

That's it! Your Liferay JSF portlet is upgraded and deployable to 7.0!

You deploy a Liferay JSF portlet to 7.0 the same way you deploy to Portal 6.x. When the plugin WAR file lands in the `[Liferay_Home]/deploy` folder, Liferay DXP's Plugin Compatibility Layer converts the WAR to a Web Application Bundle (WAB) and installs the portlet as a WAB to Liferay DXP's OSGi runtime.

On deploying an upgraded portlet, the server prints messages that indicate the following portlet status:

- WAR processing
- WAB startup
- Availability to users

Deploying a Liferay JSF portlet produces messages like these:

```
13:41:43,690 INFO ... [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:252] Processing com.liferay.faces.demo.jsf.applicant.portlet-
1.0.war
...
13:42:03,522 INFO  [fileinstall-C:/liferay-ce-portal-7.0-ga4/osgi/war][BundleStartStopLogger:35] STARTED com.liferay.faces.demo.jsf.applicant.portlet-
1.0_4.1.0 [503]
...
13:42:05,169 INFO  [fileinstall-C:/liferay-ce-portal-7.0-ga4/osgi/war][PortletHotDeployListener:293] 1 portlet for com.liferay.faces.demo.jsf.applicant.port
1.0 is available for use
```

After the portlet deployment is complete, it's available on Liferay DXP.

You've learned how to upgrade and deploy a Liferay JSF portlet. You resolved dependencies and deployed the portlet as you always have. It's just that easy!

16.6 Upgrading a Struts Portlet

Struts is a stable, widely adopted framework that implements the Model View Controller (MVC) design pattern. If you have a Struts portlet for Liferay Portal 6.2, you can upgrade it to 7.0.

Upgrading Struts portlets to 7.0 is easier than you might think. Liferay DXP lets you continue working with Struts portlets as Java EE web applications. On deploying a Struts portlet Web Application aRchive (WAR), Liferay DXP's Web Application Bundle (WAB) Generator creates an OSGi module (bundle) for the portlet and installs it to Liferay's OSGi framework. The Struts portlet behaves just as it did in 6.2 on your 7.0 site.

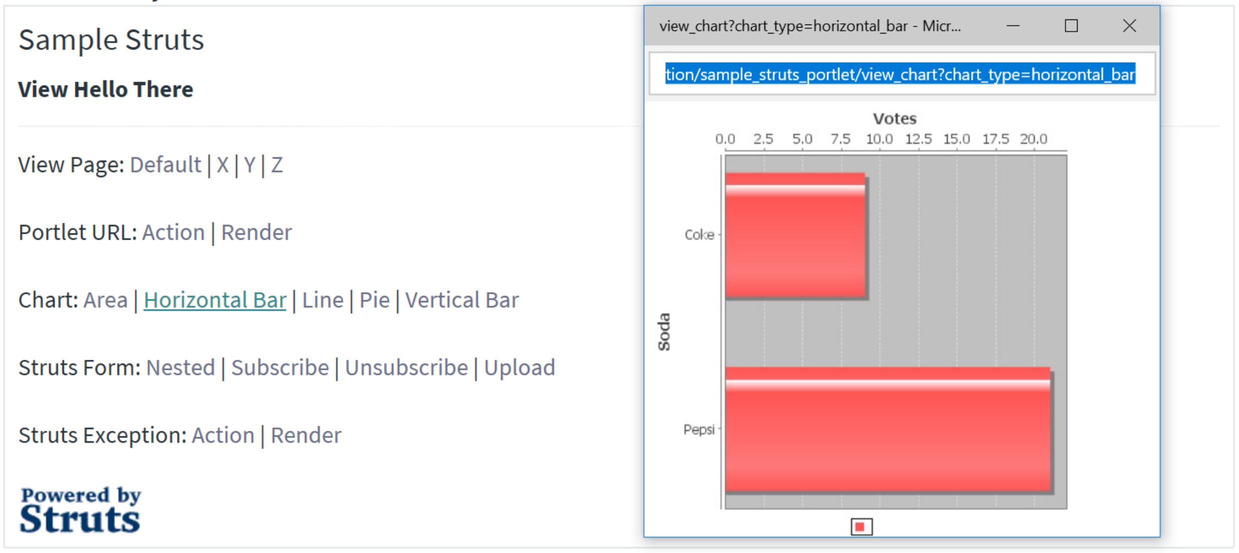

Sample Struts
View Hello There
View Page: Default | X | Y | Z
Portlet URL: Action | Render
Chart: Area | Horizontal Bar | Line | Pie | Vertical Bar
Struts Form: Nested | Subscribe | Unsubscribe | Upload
Struts Exception: Action | Render
Powered by Struts
view_chart?chart_type=horizontal_bar - Micr...
tion/sample_struts_portlet/view_chart?chart_type=horizontal_bar
Votes
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0
Soda
Coke
Pepsi

Figure 16.8: The Sample Struts portlet's charts compare fictitious soft drink survey results.

This tutorial demonstrates how to upgrade a portlet that uses the Struts 1 Framework and refers to Liferay's Sample Struts portlet (Sample Struts) as an example. Sample Struts uses several Struts features to show page navigation, `Action` and `ActionForm` Controller classes, exceptions, and more.

Here are the sample portlet's characteristics:

- Model class (`Book`)
- View comprised of JSPs that leverage Struts tag libraries
- `Action` classes (Controllers) for handling requests and responses
- `ActionForm` classes for interacting with models and forwarding requests to `Action`s
- Visual component reuse with Tiles
- Internationalization using Action Messages and resource bundles
- Form validation using the Validator Framework
- Error management through Action Errors

You can follow this tutorial to upgrade your Struts portlet. Along the way, you can examine the Sample Struts portlet source code from before and after its upgrade:

- Liferay Portal 6.2 Sample Struts portlet code

- 7.0 Sample Struts portlet code

Here's the Sample Struts portlet's folder structure:

- `sample-struts-portlet`
  - `docroot/`
    * `html/portlet/sample_struts_portlet/` → JSPs
    * `WEB-INF/`
      · `lib/` → Required third-party libraries unavailable in the Liferay DXP system

- src/
- com/liferay/samplestruts/model/ → Model classes
- com/liferay/samplestruts/servlet/ → Test servlet and servlet context listener
- com/liferay/samplestruts/struts/
- action/ → Action classes that return View pages to the client
- form/ → ActionForm classes for model interaction
- render/ → Action classes that present additional pages and handle input
- SampleException.java → Exception class
- content/test/ → Resource bundles
- META-INF/ → Javadoc
- tld/ → Tag library definitions
- liferay-display.xml → Sets the application category
- liferay-plugin-package.properties → Sets metadata and portal dependencies
- liferay-portlet.xml → Maps descriptive role names to roles
- liferay-releng.properties → (internal) Release properties
- portlet.xml → Defines the portlet and its initialization parameters and security roles
- struts-config.xml → Struts configuration
- tiles-defs.xml → Struts Tile definitions
- validation.xml → Defines form inputs for validation
- validation-rules.xml → Struts validation rules
- web.xml → Web application descriptor

– build.xml → Apache Ant build file

Upgrading a Struts portlet involves these steps:

1. Adapt the code to Liferay 7.0's API
2. Resolve dependencies

Adapting the code to Liferay 7.0's API

Liferay's Upgrade Planner identifies code affected by the new API, explains the API changes and how to adapt to them, and in many cases, provides options for adapting the code automatically.

Adapting the Sample Struts portlet's code is straightforward. Resolving its dependencies is more involved.

Resolving dependencies

The Liferay Portal 6.2 Sample Struts portlet depends on Liferay Portal to provide required third-party libraries and tag library definitions (TLDs). The portal-dependency-jars and portal-dependency-tlds properties in the portlet's liferay-plugin-package.properties specifies them:

```
portal-dependency-jars=\
    antlr2.jar,\
    commons-beanutils.jar,\
    commons-collections.jar,\
    commons-digester.jar,\
    commons-fileupload.jar,\
    commons-io.jar,\
    commons-lang.jar,\
    commons-validator.jar,\
    jcommon.jar,\
    jfreechart.jar,\
    oro.jar,\
    portals-bridges.jar,\
    struts.jar

portal-dependency-tlds=\
    struts-bean.tld,\
    struts-bean-el.tld,\
    struts-html.tld,\
    struts-html-el.tld,\
    struts-logic.tld,\
    struts-logic-el.tld,\
    struts-nested.tld,\
    struts-tiles.tld,\
    struts-tiles-el.tld
```

Resolving the tag libraries is easy.

Resolving Tag Library Definitions

7.0 continues to provide many of the same TLDs Liferay Portal 6.2 provided.

If the 7.0 application's `WEB-INF/tld` folder contains a TLD you need, add it to your portlet's `portal-dependency-tlds` property in the `liferay-plugin-package.properties` file. If the folder doesn't contain the TLD, find the TLD on the web, download it, and add it to your portlet's `WEB-INF/tld` folder.

Resolving Third-Party Libraries

Third-party libraries listed as `portal-dependency-jars` in a 6.x portlet's `liferay-plugin-package.properties` file might not be provided by 7.0. Providing fewer libraries streamlines Liferay DXP. Liferay DXP has replaced some of its libraries with newer ones.

7.0 exposes (exports) Java packages instead of sharing JAR content wholesale. If you need packages Liferay DXP doesn't export, you can find and download the artifact (JAR) that provides them and add it to your portlet's `docroot/WEB-INF/lib` folder.

Here are steps for resolving the Sample Struts portlet's Java package dependencies:

1. Liferay DXP doesn't export Antlr packages. Replace the portlet's JAR file `antlr2.jar` with newer JAR `antlr.jar` from Maven Central.

2. Liferay DXP doesn't export packages from `portals-bridges.jar`. Replace the portlet's JAR file `portals-bridges.jar` with these JARs from Maven Central:

    - `portals-bridges-common.jar`
    - `portals-bridges-struts.jar`

3. Liferay DXP doesn't export packages from `struts.jar`. Replace the portlet's JAR file `struts.jar` with the following ones from Maven Central:

    - `struts-taglib.jar`
    - `struts-tiles.jar`

4. The `Import-Packages` heading in the `com.liferay.portal.bootstrap` module's `system.packages.extra.bnd` file lists the following JAR files Sample Struts requires. Since the bootstrap module exports packages from these JARs, add their names to the `portal-dependency-jars` property in the portlet's `liferay-plugin-package.properties` file. :

- `commons-beanutils.jar`
- `commons-collections.jar`
- `commons-lang.jar`

5. Add all the rest of the JARs the Sample Struts portlet depends on to the portlet's `WEB-INF/lib` folder.

The following table summarizes the sample portlet's Java dependency resolution.
Sample Struts Portlet's Dependency Resolution:
JAR | System exports the packages theportletneeds from this JAR? | Resolution | `antlr.jar` | No | add to `WEB-INF/lib` `commons-beanutils.jar` | Yes | List in `portal-dependency-jars` | `commons-collections.jar` | Yes | List in `portal-dependency-jars` | `commons-digester.jar` | No | add to `WEB-INF/lib` | `commons-fileupload.jar` | No | add to `WEB-INF/lib` | `commons-io.jar` | No | add to `WEB-INF/lib` | `commons-lang.jar` | Yes | List in `portal-dependency-jars` | `commons-validator.jar` | No | Add to `WEB-INF/lib` | `jcommon.jar` | No | Add to `WEB-INF/lib` | `jfreechart.jar` | No | Add to `WEB-INF/lib` | `oro.jar` | No | Add to `WEB-INF/lib` | `portals-bridges-common.jar` | No | Add to `WEB-INF/lib` | `portals-bridges-struts.jar` | No | Add to `WEB-INF/lib` | `struts-core.jar` | No | Add to `WEB-INF/lib` | `struts-extras.jar` | No | Add to `WEB-INF/lib` | `struts-taglib.jar` | No | Add to `WEB-INF/lib` | `struts-tiles.jar` | No | Add to `WEB-INF/lib` |

Note: The official Sample Struts portlet for 7.0 uses Apache Ant/Ivy to manage dependencies.

For more details on resolving dependencies, see the tutorial Resolving a Plugin's Dependencies.
You've resolved the Sample Struts portlet's dependencies. It's ready to deploy.

Important: Setting Portal property `jsp.page.context.force.get.attribute` (described in the JSP section) to `true` (default) forces calls to `com.liferay.taglib.servlet. PageContextWrapper#findAttribute(String)` to use `getAttribute(String)`. Although this improves performance by avoiding unnecessary fall-backs, it can cause attribute lookup problems in Struts portlets. To use Struts portlets in your sites, makes sure to set the Portal property `jsp.page.context.force.get.attribute` to `false` in a file `[Liferay-Home]/portal-ext.properties`.

```
jsp.page.context.force.get.attribute=false
```

Deploy the Struts portlet as you normally would. The server prints messages indicating the following portlet status:

- WAR processing
- WAB startup
- Availability to users

Deploying the Sample Struts portlet produces these messages:

```
00:15:20,344 INFO  [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:252] Processing sample-struts-portlet-
7.0.0.1.war
00:15:26,871 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][BaseAutoDeployListener:42] Copying portlets for C:\portals\lifer
dxp-digital-enterprise-7.0-sp1\tomcat-8.0.32\temp\20170727241526847GURDCOLU\sample-struts-portlet-7.0.0.1.war
00:15:27,282 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][BaseDeployer:863] Deploying sample-struts-
portlet-7.0.0.1.war
00:15:29,627 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][BaseAutoDeployListener:50] Portlets for C:\portals\liferay-
dxp-digital-enterprise-7.0-sp1\tomcat-8.0.32\temp\20170727241526847GURDCOLU\sample-struts-portlet-7.0.0.1.war copied successfully
00:15:29,644 WARN  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][WabProcessor:564] The property "portal-
dependency-jars" is deprecated. Specified JARs may not be included in the class path.
```

```
00:15:33,123 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][BundleStartStopLogger:35] STARTED sample-
struts-portlet_7.0.0.1 [1230]
00:15:34,063 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][HotDeployImpl:226] Deploying sample-struts-
portlet from queue
00:15:34,065 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][PluginPackageUtil:1007] Reading plugin package for sample-
struts-portlet
00:15:34,106 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][PortletHotDeployListener:201] Registering portlets for sample-
struts-portlet
00:15:34,424 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][PortletHotDeployListener:313] 1 portlet for sample-
struts-portlet is available for use
```

The Struts portlet is available on your Liferay DXP instance.

Congratulations on upgrading your Struts portlet to 7.0!

Related Topics

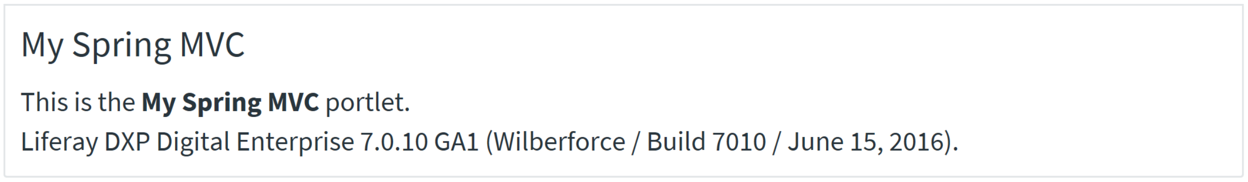
Using the WAB Generator

Using Dependency Management Tools

16.7 Upgrading a Spring MVC portlet

The Spring Portlet MVC framework facilitates injecting dependencies and implementing the Model View Controller pattern in portlets. If you use this framework in a portlet for Liferay Portal 6.x, you can upgrade it to 7.0.

This tutorial demonstrates upgrading a Spring MVC portlet called My Spring MVC (project `my-spring-mvc-portlet`). It's a bare-bones portlet created from the Plugins SDK's `spring_mvc` template.



My Spring MVC

This is the **My Spring MVC** portlet.
Liferay DXP Digital Enterprise 7.0.10 GA1 (Wilberforce / Build 7010 / June 15, 2016).

Figure 16.9: My Spring MVC portlet shows its name and Liferay DXP's information.

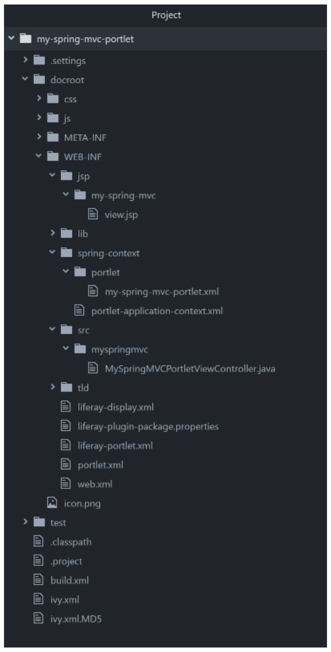
To follow along, download and refer to the original source code and the upgraded source code.

The figure below shows the `my-spring-mvc-portlet` project.

These files have Spring-related content:

- `view.jsp` → Shows the portlet's name and Liferay DXP's release information.
- `my-spring-mvc-portlet.xml` → Liferay DXP uses this context file for the portlet.
- `portlet-applications-context.xml` → Spring's `SpringContextLoaderListener` class uses this context file.
- `MySpringMVCPortletviewController` → Maps `VIEW` requests to the `view.jsp` and assigns Liferay DXP release information to a model attribute.
- `portlet.xml` → References context configuration file `my-spring-mvc-portlet.xml` and specifies a dispatcher for registered portlet request handlers.
- `web.xml` → References context configuration file `portlet-application-context.xml` and specifies a `ViewRendererServlet` to convert portlet requests and responses to HTTP servlet requests and responses.

Here are the Spring MVC portlet upgrade steps:



```
Project
my-spring-mvc-portlet
  .settings
  docroot
    css
    js
    META-INF
    WEB-INF
      jsp
        my-spring-mvc
          view.jsp
      lib
      spring-context
        portlet
          my-spring-mvc-portlet.xml
        portlet-application-context.xml
      src
        myspringmvc
          MySpringMVCPortletViewController.java
      tld
      liferay-display.xml
      liferay-plugin-package.properties
      liferay-portlet.xml
      portlet.xml
      web.xml
    icon.png
  test
  .classpath
  .project
  build.xml
  ivy.xml
  ivy.xml.MD5
```

Figure 16.10: The `my-spring-mvc-portlet` project has traditional Liferay plugin files, Spring Portlet MVC application contexts (in `spring-context/`), and a controller class `MySpringMVCPortletviewController`.

1. Adapt the code to Liferay 7.0's API
2. Resolve dependencies

Adapt the code to Liferay 7.0's API

The Upgrade Planner facilitates updating the code and resolving compilation issues quickly.

The Upgrade Planner detects if the value of the `liferay-versions` property in your plugin's `liferay-plugin-package.properties` file needs updating and it provides an option to fix it automatically. This is the only code adaptation required by `my-spring-mvc-portlet`.

Resolve Dependencies

In Liferay Portal 6.2, `my-spring-mvc-portlet` leveraged Portal's JARs by specifying them in the `liferay-plugin-package.properties` file's `portal-dependency-jars` property. Since the property is deprecated in 7.0, you should acquire dependencies using a dependency management framework, such as Gradle, Maven, or Apache Ant/Ivy.

Converting the sample portlet plugin from a traditional plugin to a Liferay Workspace web application facilitated resolving its dependencies.

Here's the updated `my-spring-mvc-portlet`'s `build.gradle` file:

```
dependencies {
    compileOnly group: 'aopalliance', name: 'aopalliance', version: '1.0'
    compileOnly group: 'commons-logging', name: 'commons-logging', version: '1.2'
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compile group: 'org.jboss.arquillian.junit', name: 'arquillian-junit-container', version: '1.1.3.Final'
    compile group: 'org.jboss.arquillian.container', name: 'arquillian-tomcat-remote-7', version: '1.0.0.CR6'
    compile group: 'com.liferay', name: 'com.liferay.ant.arquillian', version: '1.0.0-SNAPSHOT'
    compile group: 'org.springframework', name: 'spring-aop', version: '4.1.9.RELEASE'
    compile group: 'org.springframework', name: 'spring-beans', version: '4.1.9.RELEASE'
    compile group: 'org.springframework', name: 'spring-context', version: '4.1.9.RELEASE'
    compile group: 'org.springframework', name: 'spring-core', version: '4.1.9.RELEASE'
    compile group: 'org.springframework', name: 'spring-expression', version: '4.1.9.RELEASE'
    compile group: 'org.springframework', name: 'spring-web', version: '4.1.9.RELEASE'
    compile group: 'org.springframework', name: 'spring-webmvc', version: '4.1.9.RELEASE'
    compile group: 'org.springframework', name: 'spring-webmvc-portlet', version: '4.1.9.RELEASE'
}
```

Some of `my-spring-mvc-portlet`'s dependency artifacts have new names.

Old name	New name
`spring-web-portlet`	`spring-webmvc-portlet`
`spring-web-servlet`	`spring-webmvc`

Maven Central provides artifact dependency information.

Note: If the Spring Framework version you're using differs from the version Liferay DXP uses, you must name your Spring Framework JARs differently from Liferay DXP's Spring Framework JARs. If you don't rename your JARs, Liferay DXP assumes you're using its Spring Framework JARs and excludes yours from the generated WAB (Web Application Bundle). Portal property `module.framework.web.generator.excluded.paths` lists Liferay DXP's Spring Framework JARs. Understanding Excluded JARs explains how to detect the Spring Framework version Liferay DXP uses.

Note: If a dependency is an OSGi module JAR and Liferay DXP already exports your plugin's required packages, *exclude* the JAR from your plugin's WAR file. This prevents your plugin from exporting the same package(s) that Liferay is already exporting. This prevents class loader collisions. To exclude a JAR from deployment, add its name to the your project's `liferay-plugin-package.properties` file's `deploy-excludes` property.

```
deploy-excludes=\
    **/WEB-INF/lib/module-a.jar,\
    **/WEB-INF/lib/module-b.jar
```

Since `my-spring-mvc-portlet`'s dependencies aren't OSGi modules, no JARs must be excluded.

To import class packages referenced by your portlet's descriptor files, add the packages to an `Import-Package` header in the `liferay-plugin-package.properties` file. See Packaging a Spring MVC Portlet for details.

If you depend on a package from Java's `rt.jar` other than its `java.*` packages, override portal property `org.osgi.framework.bootdelegation` and add it to the property's list. Go here for details.

Note: Spring MVC portlets whose embedded JARs contain properties files (e.g., `spring.handlers`, `spring.schemas`, `spring.tooling`) might be affected by issue LPS-75212. The last JAR that has properties files is the only JAR whose properties are added to the resulting WAB's classpath. Properties in other JARs aren't added.

Packaging a Spring MVC Portlet explains how to add all the embedded JAR properties.

The portlet is ready to deploy. Deploy it as you always have.

Liferay DXP's WAB Generator converts the portlet WAR to a Web Application Bundle (WAB) and installs the WAB to Liferay's OSGi Runtime Framework.

```
21:12:23,775 INFO  [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:252] Processing my-spring-mvc-portlet-
7.0.0.1.war
...
21:12:36,159 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][PluginPackageUtil:1007] Reading plugin package for my-
spring-mvc-portlet
07-Aug-2017 21:12:36.170 INFO [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war] org.apache.catalina.core.ApplicationContext.log Initi
07-Aug-2017 21:12:36.181 INFO [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war] org.apache.catalina.core.ApplicationContext.log Initi
21:12:36,365 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][PortletHotDeployListener:201] Registering portlets for my-
spring-mvc-portlet
21:12:36,707 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][PortletHotDeployListener:313] 1 portlet for my-
spring-mvc-portlet is available for use
21:12:36,868 INFO  [fileinstall-C:/portals/liferay-dxp-digital-enterprise-7.0-sp1/osgi/war][BundleStartStopLogger:35] STARTED my-
spring-mvc-portlet_7.0.0.1 [1309]
```

You've upgraded a Spring MVC portlet to 7.0. Way to go!

Related Topics

Spring MVC

Migrating Plugins SDK Projects to Workspace and Gradle

Using Dependency Management Tools

Using the WAB Generator

16.8 Upgrading Web Plugins

Web plugins are regular Java EE web modules designed to work with Liferay DXP. These plugins were stored in the `webs` folder of the legacy Plugins SDK.

Upgrading a Liferay web plugin involves these steps:

1. Adapt the plugin to 7.0's API using the Liferay Upgrade Planner. When running the planner's *Fix Upgrade Problems* step, many of the existing issues are autocorrected. For remaining issues, the planner identifies code affected by the new API and ways to adapt it.

2. Resolve its dependencies

3. Deploy it.

After deploying the upgraded portlet, the server prints messages that indicate the following portlet status:

- WAR processing
- WAB startup
- Availability to users

You've upgraded and deployed your Liferay web plugin on your 7.0 instance. Great job!

16.9 Upgrading Ext Plugins

Ext plugins let you use internal APIs and even let you overwrite Liferay DXP core files. This puts your deployment at risk of being incompatible with security, performance, or feature updates released by Liferay. When upgrading to a new version of Liferay DXP, you must review all changes and manually modify your Ext projects to merge your changes with Liferay DXP's.

During your upgrade to 7.0, it's highly recommended to leverage an extension point to customize Liferay DXP instead of using you existing Ext plugin, if possible. 7.0 provides many extension points that let you customize almost every detail of Liferay DXP. If there's a way to customize what you want with an extension point, do it that way instead. See the More Extensible, Easier to Maintain section for more details on the advantages of using Liferay DXP's extension points.

For more information on Ext projects, how to decide if you need one, and how to manage them, see the Customization with Ext section.

16.10 Upgrading the Liferay Maven Build

If you're an avid Maven user and have been using it for Liferay Portal 6.2 project development, you must upgrade your Maven build to be compatible with 7.0 development. There are two main parts of the Maven environment upgrade process that you must address:

- Upgrading to new 7.0 Maven plugins
- Updating Liferay Maven artifact dependencies

For more information on using Maven with 7.0, see the Maven tutorial section. For a guided and expedited upgrade process for your Maven build, try the Upgrade Planner.

You'll start off by upgrading your Maven environment's Liferay Maven plugins.

Upgrading to New 7.0 Maven Plugins

The biggest change for your project's build plugins is the removal of the `liferay-maven-plugin`. Liferay now provides several individual Maven plugins that accomplish specific tasks. For example, you can configure Maven plugins for Liferay's CSS Builder, Service Builder, Theme Builder, etc. With smaller plugins available to accomplish specific tasks in your project, you no longer have to rely on one large plugin that provides many things you may not want.

For example, suppose your Liferay Portal 6.2 project was using the `liferay-maven-plugin` for Liferay CSS Builder only. First, you should remove the legacy `liferay-maven-plugin` plugin dependency from your project's `pom.xml` file:

```
<plugin>
    <groupId>com.liferay.maven.plugins</groupId>
    <artifactId>liferay-maven-plugin</artifactId>
    <version>${liferay.version}</version>
    <configuration>
        <autoDeployDir>${liferay.auto.deploy.dir}</autoDeployDir>
        <liferayVersion>${liferay.version}</liferayVersion>
        <pluginType>portlet</pluginType>
    </configuration>
</plugin>
```

Then, add the CSS Builder plugin dependency to your project's `pom.xml` file:

```
<plugin>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.css.builder</artifactId>
    <version>1.0.21</version>
    <executions>
        <execution>
            <id>default-build</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
        <configuration>
            <docrootDirName>src/main/webapp</docrootDirName>
        </configuration>
</plugin>
```

Some common Liferay Maven plugins are listed below, with their corresponding artifact IDs and tutorials explaining how to configure them:

Common Liferay Maven Plugins

Name	Artifact ID	Tutorial
CSS Builder	com.liferay.css.builder	Compiling SASS Files in a Maven Project
Lang Builder	com.liferay.lang.builder	Coming Soon
Service Builder	com.liferay.portal.tools.service.builder	Using Service Builder in a Maven Project
Theme Builder	com.liferay.portal.tools.theme.builder	Building Themes in a Maven Project

In Liferay Portal 6.2, you were also required to specify all your app server configuration settings. For example, your parent POM probably contained settings similar to these:

```
<properties>
    <liferay.app.server.deploy.dir>
        E:\liferay-portal-6.2.0-ce-ga1\tomcat-7.0.42\webapps
    </liferay.app.server.deploy.dir>

    <liferay.app.server.lib.global.dir>
        E:\liferay-portal-6.2.0-ce-ga1\tomcat-7.0.42\lib\ext
    </liferay.app.server.lib.global.dir>

    <liferay.app.server.portal.dir>
        E:\liferay-portal-6.2.0-ce-ga1\tomcat-7.0.42\webapps\root
    </liferay.app.server.portal.dir>

    <liferay.auto.deploy.dir>
        E:\liferay-portal-6.2.0-ce-ga1\deploy
    </liferay.auto.deploy.dir>

    <liferay.version>
        6.2.0
    </liferay.version>

    <liferay.maven.plugin.version>
        6.2.0
    </liferay.maven.plugin.version

</properties>
```

This is no longer required in 7.0 because Liferay's Maven tools no longer rely on your Liferay DXP installation's specific versions. You should remove them from your POM file.

Awesome! You've learned about the new Maven plugins available to you for 7.0 development. Next, you'll learn about updating your Liferay Maven artifacts.

Updating Liferay Maven Artifact Dependencies

Many Liferay Portal 6.2 artifact dependencies you were using have changed in 7.0. See the table below for popular Liferay Maven artifacts that have changed:

Liferay Portal 6.2 Artifact ID	7.0 Artifact ID
`portal-service`	`com.liferay.portal.kernel`
`util-bridges`	`com.liferay.util.bridges`
`util-java`	`com.liferay.util.java`
`util-slf4j`	`com.liferay.util.slf4j`
`util-taglib`	`com.liferay.util.taglib`

For more information on resolving dependencies in 7.0, see the Resolving a Plugin's Dependencies tutorial.

Of course, you must also update the artifacts you're referencing for your projects. If you're using the Central Repository to install Liferay Maven artifacts, you won't need to do anything more than update the artifacts in your POMs. If, however, you're working behind a proxy or don't have Internet access, you must update your company-shared or local repository with the latest 7.0 Maven artifacts. See the Installing Liferay Maven Artifacts tutorial for instructions.

With these updates, you can easily upgrade your Liferay Maven build so you can begin developing traditional plugin projects for 7.0.

Chapter 17

Optimizing Plugins for 7.0

Once you've upgraded your plugin to 7.0, you can optimize it to take advantage of all @product-ver@ offers. If you haven't yet familiarized yourself with what's changed from Liferay Portal 6, the new benefits for developers, OSGi and modularity, and the improved tooling, make sure to do so as they'll help you understand and appreciate the optional improvements for plugins and plugin development the optimization tutorials demonstrate.

Here are some common optimizations to consider:

- Migrating to environments that help you develop and test more quickly, such as the Liferay Theme Generator for themes and Liferay Workspace for modules and web applications.
- Adapting plugins to 7.0's modular architecture and updated frameworks, such as Service Builder.
- Styling your app consistently using Lexicon –the web implementation of Liferay's Lexicon Experience Language.
- Modularizing apps to reap the benefits of modularity and all that 7.0 offers.

Several optimization tutorials are here and more are coming soon.

17.1 Migrating Traditional Plugins to Workspace Web Applications

After you've adapted your traditional plugin to Liferay DXP's API, you can continue maintaining it in the Plugins SDK. The Plugins SDK, however, is deprecated as of 7.0. Maintaining plugins in the SDK will become increasingly difficult. Liferay Workspace replaces the Plugins SDK, providing a comprehensive Gradle development environment and more. A simple command migrates traditional plugins (such as portlets) to Gradle-based web application projects. From there you can build and deploy them to 7.0 as Web ARchives (WARs).

Running the Migration Command

Blade CLI's `convert` command migrates Plugins SDK plugins to web application projects in Workspace's `wars` folder. Plugin files are re-organized to follow the standard web application folder structure.

Standard Web Application Anatomy:

- `[project root]`

- `src`
    * `main`
        - `webapp`
        - `WEB-INF`
        - `classes`
        - `lib` → Libraries
        - descriptor files
        - `css` → CSS files
        - `js` → JavaScript files
        - `icons`
        - JSP files
    * `java` → Java source
- build files

In a terminal, navigate to the Liferay Workspace root folder. Then pass your Plugins SDK project's name to Blade's `convert` command:

```
blade convert [PLUGIN_PROJECT_NAME]
```

Blade extracts the plugin from the Plugins SDK and reorganizes it in a standard web application project in Workspace's `wars` folder.

Note: You can execute `blade convert -l` to show a list of projects that can be migrated in your Plugins SDK. Run `blade convert --all` to migrate all plugin projects in your Plugins SDK to Workspace.

The image below shows the plugin files before and after they're migrated to Workspace.

The following table maps traditional plugin source files to the standard web application folder structure Workspace uses.

Plugins SDK folders to web application folders:

Files	Plugins SDK folder (old)	Web app folder (new)
Java	`docroot/WEB-INF/src`	`src/main/java`
JSPs	`docroot`	`src/main/webapp`
icons	`docroot`	`src/main/webapp`
CSS	`docroot/css`	`src/main/webapp/css`
JS	`docroot/js`	`src/main/webapp/js`
descriptors	`docroot/WEB-INF`	`src/main/webapp/WEB-INF`
libraries	`docroot/WEB-INF/lib`	`src/main/webapp/lib`

From your plugin's new location, you can invoke Workspace Gradle tasks on it and build its `.war` file.

```
blade gw war
```

To deploy the `.war`, copy it from the plugin's `build/libs` folder to the `[LIFERAY_HOME]/deploy` folder.

Welcome to your plugin's new home in Workspace!

Related Topics

Workspace Development lifecycle

Workspace Gradle Tasks

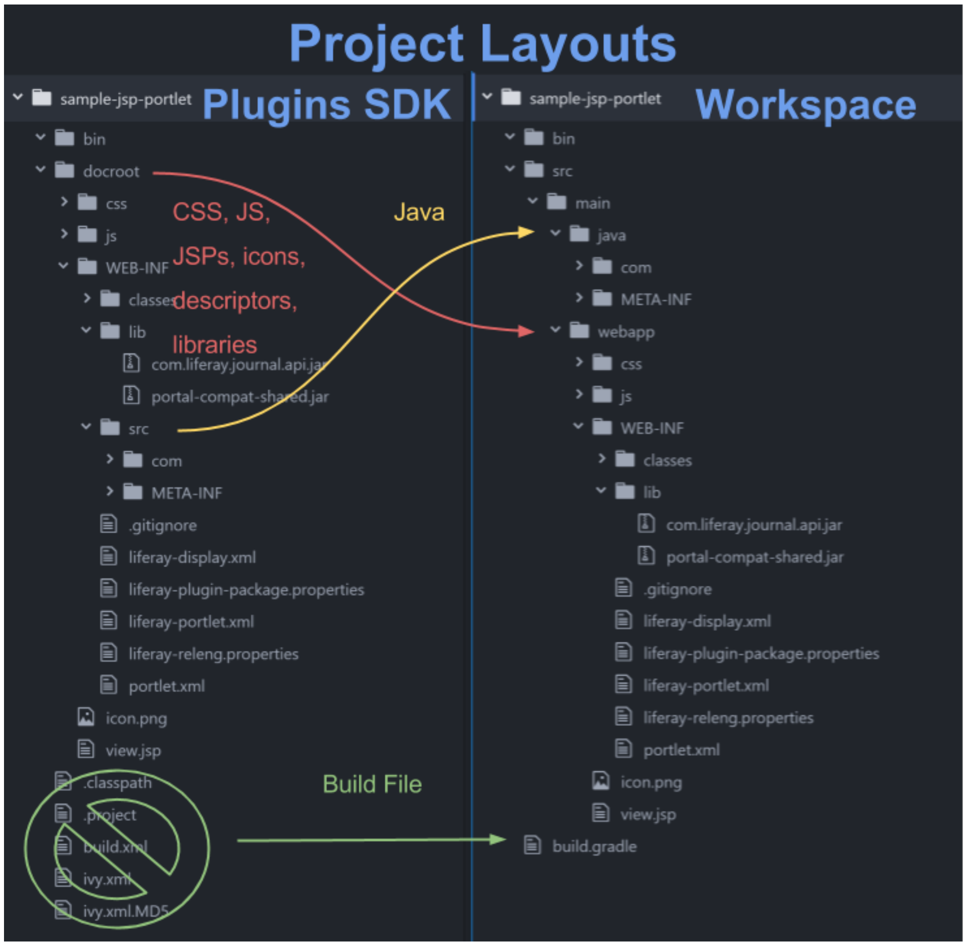


Figure 17.1: The `convert` command migrates a Plugins SDK project to a Workspace web application project. It moves Java source files to `src/main/java` and all other files/folders to `src/main/webapp`.

Chapter 18

Modularizing Plugins

As described in Benefits of Liferay 7 for Liferay 6 Developers, applications that comprise OSGi modules offer considerable advantages over monolithic applications.

The main benefit is that modular development practices structure code in ways that reduce maintenance costs. These practices involve, for example, defining contracts (such as APIs) more clearly, hiding internal classes, and handling dependencies more carefully. Related to this, module dependencies are explicitly listed within a module. Modules run only when all their dependencies are met–this can eliminate many obscure run time errors.

Splitting large applications into small independent modules lets you focus on smaller release cycles for those modules. Individual modules can be updated independently of the others. For instance, you might fix a JSP's security issue in an application's web (client) module. The issue only affects that module–none of the application's other modules need change.

The scenarios described below can help you decide whether to convert an application to modules.

When not to convert?

- You have a portlet that's JSR-168/286 compatible and you still want to be able to deploy it to another portlet container. In this case, it's best to stay with the traditional WAR model. (To eliminate this reason for not converting, Liferay is discussing with other vendors the possibility of making modular portlets a standard.)
- You're using a complex web framework that is heavily tied to the Java EE programming model and the amount of effort necessary to make it work with OSGi is more than you feel is necessary or warranted.
- You want to minimize effort to get your application to working on 7.0.

When to convert?

- You have a very large application with many lines of code. If you've got lots of developers making changes, separating the code into modules can make it easier and faster to get releases out.
- Your application has reusable parts that you want to consume outside of it. For instance, you have business logic that you're reusing in different projects. Modules let you consume their services from other modules.
- In general, you want to start reaping the benefits of modular development.

You can now make an informed decision on whether to stick with your upgraded traditional application as is or modularize it to leverage 7.0's modularity features.

18.1 Modularizing an Existing Portlet

An application with properly modularized plugins offers several benefits. You can release individually its plugins without releasing the entire application. External clients can consume services from particular plugins, without having to depend on an entire application. And by splitting up large amounts of code into concise modules, teams can more easily focus on particular areas of the application. These are just a few reasons to modularize application plugins.

In this tutorial, you'll learn how to convert your traditional application into modules. Before getting started, it's important to reiterate that the module structure shown in this tutorial is just one of many ways for structuring your application's modules. It's also important to remember that applications come in all different shapes and sizes. There may be special actions that some applications require. This tutorial provides the general process for converting to modules using Liferay's module structure.

Here's what's involved:

- Converting portlet classes and the UI
- Converting Service Builder interfaces and implementations
- Building and deploying modules

The instructions covered in this tutorial apply to both the commercial and open source versions of Liferay.

The first thing you'll do is create your application's web (client) module.

Converting Your Application's Portlet Classes and UI

The first thing you'll do is create your application's parent directory and the directory structure for your application's *web* client module. This module holds your application's portlet classes and is responsible for its UI. Before you start creating a skeleton structure for your application's modules, you should determine which modules will comprise this version of your application. If your application provides service and API classes (which is the case for all Liferay Service Builder applications), you should create separate modules for your service implementation and service API classes. This tutorial assumes the Maven project model, although any build tools or directory setup is permissible.

Note: It's recommended that you use the build plugin versions that support the latest OSGi features. The following Gradle or Maven build plugin versions should be used in their respective build frameworks:

Gradle - biz.aQute.bnd:biz.aQute.bnd.gradle:3.1.0 **or** - org.dm.gradle:gradle-bundle-plugin:0.8.1

Maven - biz.aQute.bnd:bnd-maven-plugin:3.1.0 **or** - org.apache.felix:maven-bundle-plugin:3.0.1

---

1. Create your application's parent folder. It is home for your application's independent modules and configuration files. For example, if your application's name is *Tasks*, then your parent folder could be *tasks*.

   If your application uses Liferay Service Builder, use the following Blade CLI command to generate the parent folder and service implementation and service API modules in it. If the parent folder already exists, it must be empty. This command names the parent folder after the `APPLICATION_NAME`:

   ```
   blade create -t service-builder -p [ROOT_PACKAGE] [APPLICATION_NAME]
   ```

   The `*-service` and `*-api` module folders are described later in this tutorial.

2. Create the folder structure for your web client module. You can do this automatically by using Blade CLI. The portlet tutorials demonstrate creating all different kinds of portlets.

   Navigate to your parent directory (e.g., tasks) and run the following Blade CLI command to generate a generic web client module structure:

   ```
   blade create -t mvc-portlet -p [ROOT_PACKAGE] [APPLICATION_NAME]-web
   ```

3. Replace the `/src/main/java/[APPLICATION_NAME]` folder with your root package. For instance, if your application's root package name is `com.liferay.tasks.web`, your class's directory should be `/src/main/java/com/liferay/tasks/web`. Also, remove the `init.jsp` and `view.jsp` files located in the `src/main/resources/META-INF/resources` folder. You'll insert your traditional application's Java code and JSPs, so the generated default code is not necessary.

4. Verify that your current directory structure for your application's `*-web` module matches the structure listed below:

   - `tasks`
     - `tasks-web`
       - `src`
         - `main`
         - `java`
         - `[ROOT_PACKAGE]`

         - `resources`
         - `content`
         - `Language.properties`

         - `META-INF`
         - `resources`
       - `bnd.bnd`
       - `build.gradle`

   The instructions in the rest of this sub-section only affect your application's web client module.

5. Open the `bnd.bnd` file. This is used to generate your module's `MANIFEST.MF` file that is generated when you build your project. Edit your module's `bnd.bnd` file to fit your application. For more information about configuring your module's `bnd.bnd`, visit http://bnd.bndtools.org/. You can view the `dictionary-web` module's `bnd.bnd` for a simple example below:

   ```
   Bundle-Name: Liferay Dictionary Web
   Bundle-SymbolicName: com.liferay.dictionary.web
   Bundle-Version: 1.0.6
   ```

   For a more advanced example, examine the `journal-web` module's `bnd.bnd`:

```
Bundle-Name: Liferay Journal Web
Bundle-SymbolicName: com.liferay.journal.web
Bundle-Version: 2.0.0
Export-Package:\
    com.liferay.journal.web.asset,\
    com.liferay.journal.web.dynamic.data.mapping.util,\
    com.liferay.journal.web.social,\
    com.liferay.journal.web.util
Liferay-JS-Config: /META-INF/resources/js/config.js
Liferay-Releng-Module-Group-Description:
Liferay-Releng-Module-Group-Title: Web Content
Web-ContextPath: /journal-web
```

6. Open the `build.gradle` file. This is used to specify all your module's dependencies. The `build.gradle` file that was generated for you is pre-populated with content and default dependencies related to OSGi and Liferay DXP. In the `dependencies {...}` block, you need to add the web client module's dependencies. To learn how to find and specify dependencies on Liferay API modules, refer to the reference document Finding Liferay API Modules. When deploying your module into the OSGi container, OSGi checks if the dependencies are available in the container. If the dependencies are not available in the container, your module will be unavailable. Therefore, your dependencies are not bundled with your module. Instead, they're available from Liferay's OSGi container.

7. Copy your traditional application's JSP files into the `/src/main/resources/META-INF/resources` directory. In most cases, all of your application's JSP files should reside in the web client module.

8. Your next task is to add your portlet classes, non-service classes, and non-implementation classes into your client module. Copy your portlet classes into their respective directories and ensure their package names within the class are specified correctly. Your client module can hold one class or many classes, depending on how large your application is. It's a good practice to organize your classes into sub-packages of the main package, to more easily manage them. You'll examine the `journal-web` module for an example of a client module holding many different Java classes:

- `journal-web`
    - ...
    - `src/main/java/com/liferay/journal/web/`
        - `asset`
            - [classes]
        - `configuration`
            - [classes]
        - `dynamic/data/mapping/util`
            - [classes]
        - `internal`
            - `application/list`
            - [classes]

            · custom/attributes
            · [classes]

            · dao/search
            · [classes]

            · ...

        * social

            · [classes]

        * util

            · [classes]

    - ...

```
**Note:** Many applications have service and API classes. These classes
need to live in separate implementation and API modules. You'll learn more
about creating these later in this tutorial.
```

---

9. Now that you have the necessary classes in your client module, you need to edit these classes to be compliant with OSGi. First, you need to choose a component framework to work with. Using a component framework lets you easily harness the power of OSGi. Liferay DXP uses the Declarative Services component framework and recommends that Liferay developers use it too. This tutorial assumes that you're using Declarative Services. You can, however, use any other OSGi component framework in Liferay DXP.

   Review your traditional application's XML files and migrate the configuration and metadata information to the portlet class as properties. You can do this by adding the `@Component` annotation to your portlet class and adding the necessary properties to that annotation. The end result should look similar to the following example:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.icon=/icon.png",
        "javax.portlet.name=1",
        "javax.portlet.display-name=Tasks Portlet",
        "javax.portlet.security-role-ref=administrator,guest,power-user",
        "javax.portlet.init-param.clear-request-parameters=true",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.expiration-cache=0",
        "javax.portlet.supports.mime-type=text/html",
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.info.title=Tasks Portlet",
        "javax.portlet.info.short-title=Tasks",
        "javax.portlet.info.keywords=Tasks",
    },
    service = Portlet.class
)
public class TasksPortlet extends MVCPortlet {
```

10. Convert all references of the `portletId` (e.g., `58_INSTANCE_4gtH`) to the class name of the portlet, replacing all periods with underscores (e.g., `com_liferay_web_proxy_portlet_WebProxyPortlet`).

11. If your traditional application has resource actions, you'll need to migrate those into your client module. Create the `/src/main/resources/resource-actions/default.xml` file, and copy your resource actions there. Make sure to create the `src/portlet.properties` file and add the following property:

    ```
    resource.actions.configs=resource-actions/default.xml
    ```

    As an example, you can view the Directory application's `default.xml` file.

12. Add any language keys that your application uses to the `src/main/resources/content/Language.properties` file. You should only include the language keys that are unique to your application. Your application will use the default language keys in Liferay when it is deployed.

Awesome! You've created your application's web client module and navigated through some of the most common tasks necessary to modularize your portlet classes and UI. There are certain parts of your application that may not be covered in this tutorial that you must account for. Liferay's Developer Network provides developer tutorials divided into popular areas so you can easily find the correct way to transform your legacy code to use Liferay DXP's updated best practices.

The table below serves as a quick reference guide. It summarizes the migration process for many of your application's directories, packages, and files. This is a sample table for a fictitious *tasks* applications.

Plugin Package	Module Package
`tasks-portlet/docroot/WEB-INF/src/com.liferay.tasks.asset`	`tasks-web/src/main/java/com.liferay.tasks.asset`
`tasks-portlet/docroot/WEB-INF/src/com.liferay.tasks.portlet`	`tasks-web/src/main/java/com.liferay.tasks.portlet`
`tasks-portlet/docroot/WEB-INF/src/content`	`tasks-web/src/main/resources/content`
`tasks-portlet/docroot/WEB-INF/src/resource-actions`	`tasks-web/src/main/resources/resource-actions`
`tasks-portlet/docroot/WEB-INF/src/portlet.properties`	`tasks-web/src/main/resources/portlet.properties`
`tasks-portlet/docroot/init.jsp`	`tasks-web/src/main/resources/META-INF/resources/init.jsp`
`tasks-portlet/docroot/tasks`	`tasks-web/src/main/resources/META-INF/resources/tasks`
`tasks-portlet/docroot/upcoming_tasks`	`tasks-web/src/main/resources/META-INF/resources/upcoming_tasks`

Many applications only have a web client module. Larger, more complex applications, such as Liferay Service Builder applications, require additional modules to hold their service API and service implementation logic. You'll learn how to create these modules next.

Converting Your Application's Service Builder API and Implementation

In this section, you'll learn about converting a Liferay Portal 6 Service Builder application to a 7.0 style application. In the previous section, you learned how to generate your implementation and API modules. If you haven't yet run the `service-builder` Blade CLI command outlined in step 2 of the previous section, run it now. The API module holds your application's Service Builder generated API and the implementation module holds your application's Service Builder implementation.

Before you begin editing the API and implementation modules, you'll need to configure your root project (e.g., `tasks`) to recognize the multiple modules residing there. A multi-module Gradle project must have a `settings.gradle` file in the root project for building purposes. Luckily, when you generated your Service Builder project's modules using Blade CLI, the `settings.gradle` file was inserted and pre-configured for the `api` and `service` modules. You should add your `web` module into the Service Builder project's generated parent folder and define it in the `settings.gradle` file too. You'll configure your `web` module via Gradle settings later, but for now, go ahead and copy the module into the project generated by the `service-builder` template. For example, an example tasks project's root folder would look like this:

- `tasks`
    - `gradle`
    - `tasks-api`
    - `tasks-service`
    - `tasks-web`
    - `build.gradle`
    - `gradlew`
    - `settings.gradle`

Your root project directory should now be in good shape. Next, you'll learn how to use Service Builder to generate your application's service API and service implementation code.

1. Copy your traditional application's `service.xml` file and paste it into the implementation module's root directory (e.g., `tasks/tasks-service`).

2. Blade CLI generated a `bnd.bnd` file for your service implementation module. Make sure to edit this `bnd.bnd` file to fit your application. For an example of a service implementation module's BND file, examine the `export-import-service` module's BND below:

```
Bundle-Name: Liferay Export Import Service
Bundle-SymbolicName: com.liferay.exportimport.service
Bundle-Version: 4.0.0
Export-Package:\
    com.liferay.exportimport.content.processor.base,\
    com.liferay.exportimport.controller,\
    com.liferay.exportimport.data.handler.base,\
    com.liferay.exportimport.lar,\
    com.liferay.exportimport.lifecycle,\
    com.liferay.exportimport.messaging,\
    com.liferay.exportimport.portlet.preferences.processor.base,\
    com.liferay.exportimport.portlet.preferences.processor.capability,\
    com.liferay.exportimport.search,\
    com.liferay.exportimport.staged.model.repository.base,\
    com.liferay.exportimport.staging,\
    com.liferay.exportimport.xstream
Liferay-Releng-Module-Group-Description:
Liferay-Releng-Module-Group-Title: Data Management
Liferay-Require-SchemaVersion: 1.0.0
-includeresource: content=../../staging/staging-lang/src/main/resources/content
```

3. Blade CLI also generated your service implementation module's `build.gradle` file. In this file, Service Builder is already configured to generate code both in this module and in your service API module. When you run Service Builder, Java classes, interfaces, and related files are generated in your `*api` and `*service` modules. Open your service implementation module's `build.gradle` file to view the default configuration.

   As you've learned already, you don't have to accept the generated build files' defaults. Blade CLI simply generated some standard OSGi and Liferay configurations.

   For example, Service Builder is already available for you by default. Blade CLI applies the Service Builder plugin automatically when a project contains the `service.xml` file. With the Service Builder plugin already available, you don't have to worry about configuring it in your project.

4. Another important part of your service implementation module's `build.gradle` file is the `buildService{...}` block. This block configures how Service Builder runs for your project. The current configuration will generate your API module successfully, but extra configuration might be necessary in certain cases.

5. Open a terminal and navigate to your root project folder. Then run `gradlew buildService`.

   Your `service.xml` file's configuration is used to generate your application's service API and service implementation classes in their respective modules. You've also generated other custom files (related to SQL, Hibernate, Spring, etc.), depending on your `buildService {...)` block's configuration. For more information on configuration options for the Service Builder plugin, see the Service Builder Gradle Plugin reference article.

6. Now that you've run Service Builder, continue copying custom classes into your implementation module. The table below highlights popular Liferay Portal 6 classes and packages and where they should be placed in your application. This table is intended to aid in the organization of your classes and configuration files; however, remember to follow the organizational methodologies that make the most sense for your application. One size does not fit all with your modules' directory schemes.

Plugin Package	Module Package
`tasks-portlet/docroot/WEB-INF/src/com.liferay.tasks.model.impl`	`tasks-service/src/main/java/com.liferay.tasks.model.impl`
`tasks-portlet/docroot/WEB-INF/src/com.liferay.tasks.service.impl`	`tasks-service/src/main/java/com.liferay.tasks.service.impl`
`tasks-portlet/docroot/WEB-INF/src/com.liferay.tasks.service.permission`	`tasks-service/src/main/java/com.liferay.tasks.service.permission`
`tasks-portlet/docroot/WEB-INF/src/com.liferay.tasks.service.persistence.impl`	`tasks-service/src/main/java/com.liferay.tasks.service.persistence.impl`
`tasks-portlet/docroot/WEB-INF/src/com.liferay.tasks.social`	`tasks-service/src/main/java/com.liferay.tasks.social`
`tasks-portlet/docroot/WEB-INF/src/com.liferay.tasks.util`	`tasks-service/src/main/java/com.liferay.tasks.util`
`tasks-portlet/docroot/WEB-INF/src/custom-sql`	`tasks-service/src/main/resources/META-INF/custom-sql`

---

7. Once you've copied all of your custom classes over, run `gradlew buildService` again to generate the remaining services.

Now that your services are generated, you'll need to wire up your modules so they can reference each other when deployed to Liferay's OSGi container. Blade CLI has already partially completed this task. For example, it assumes that the service implementation module depends on the service API module.

You still need to associate the client module with the `api` and `service` modules, since they were generated separately. To do this, follow the steps below:

1. In your project's `settings.gradle` file, you must add the `web` module with the `api` and `service` modules so it's included in the Gradle build lifecycle:

```
include "tasks-api", "tasks-service", "tasks-web"
```

2. Add the `api` and `service` modules as dependencies in you client module:

```
dependencies {
    compileOnly  project(':tasks-api')
    compileOnly  project(':tasks-service')
}
```

Excellent! You've successfully generated your application's services using Service Builder. They now reside in modules, and can be deployed to 7.0.

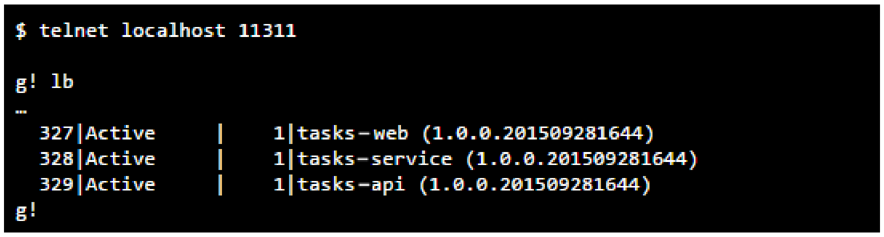
Building Your Module JARs for Deployment

Now it's time to build your modules and deploy them to your Liferay DXP instance. To build your project, run `gradlew build` from your project's root directory.

Once your project successfully builds, check all of your modules' `/build/libs` folders. There should be a newly generated JAR file in each, which is the file you'll need to deploy to Liferay DXP. You can deploy each JAR by running `blade deploy` from each module's root directory.

Note: If you deploy your modules out of order, you might receive error messages. For instance, if you try deploying your web client module first, you'll receive errors if it relies on the service implementation and service API modules. Once each module's dependencies are met, they will successfully be deployed for use in Liferay. For more information on checking each module's dependencies, see the Using the Felix Gogo Shell article.

Once you've successfully deployed your modules, you can list them from the Gogo shell as shown below.



```
$ telnet localhost 11311

g! lb
…
  327|Active     |    1|tasks-web (1.0.0.201509281644)
  328|Active     |    1|tasks-service (1.0.0.201509281644)
  329|Active     |    1|tasks-api (1.0.0.201509281644)
g!
```

Figure 18.1: Once you've connected to your Liferay instance in your Gogo shell prompt, run *lb* to list your new converted modules.

This tutorial explained how to convert your traditional application into the modular format of a 7.0 style applicaton. Specifically, you learned how to

- Create a web client (`*-web`) module that holds your application's portlet classes and UI.
- Create a service implementation module (`*-service`) and a service API module (`*-api`).
- Run Service Builder to generate code for your application's service and API modules.
- Wire your modules together by declaring their dependencies on each another.
- Build your modules and deploy them to your Liferay DXP installation.

Great job!

Related Topics

Portlets

Service Builder

18.2 Migrating Data Upgrade Processes to the New Framework for Modules

When you make changes to your plugin that affect the database, you can use a *data upgrade process* to upgrade data to the new database schema. 7.0 has a new data upgrade framework for modules. While the old framework required several classes, the new framework lets you orchestrate the upgrade steps from a single class. Managing the steps from one class facilitates developing upgrade processes. The data upgrade framework you use depends on your development framework.

- If your upgraded plugin is a traditional WAR, you don't need to do anything special; existing upgrade processes adapted to 7.0's API work as is. The new data upgrade framework is for modules only.

- If you converted your upgraded plugin to a module or you have an upgraded module, you must migrate any upgrade processes you want to continue using to the new data upgrade framework.

You can migrate any number of old upgrade processes (starting with the most recent ones) to the new framework. For example, if your module has versions 1.0, 1.1, 1.2, and 1.3, but you only expect customers on versions 1.2 and newer to upgrade, you might migrate upgrade processes for versions 1.2 and 1.3 only. This tutorial shows you how to migrate to the new framework.

Before beginning, make sure you know how to create an upgrade process that uses the new framework. Click here to read the tutorial on creating these upgrade processes.

Note: Liferay Portal 6 plugins may also include verify processes. Although you can migrate the verify processes to 7.0 without any changes, it's a best practice to perform verification in your upgrade processes instead.

First, you'll review how Liferay Portal 6 upgrade processes work.

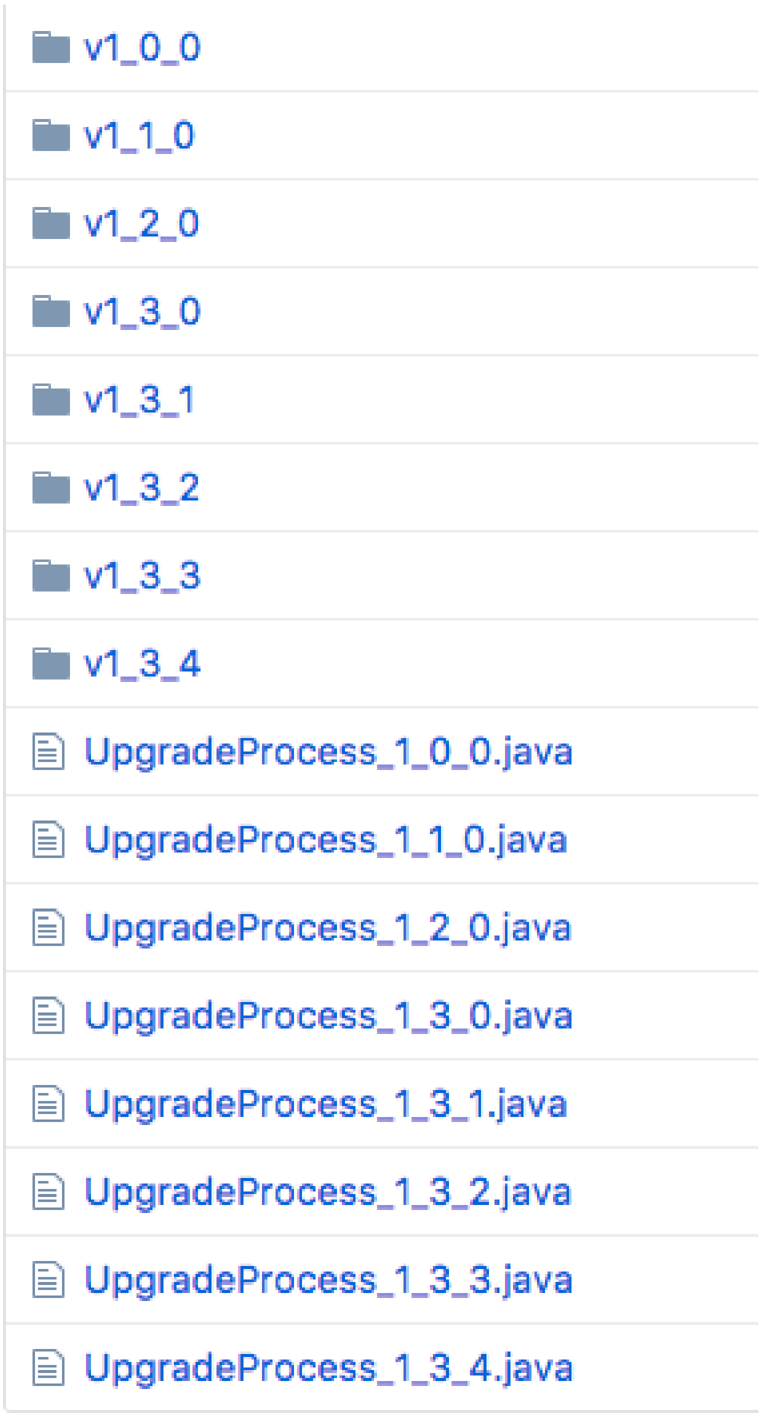
Understanding Liferay Portal 6 Upgrade Processes

Before getting started, it's important to understand how Liferay Portal 6 upgrade processes are structured. As an example, you'll use the Liferay Portal 6.2 upgrade process for the Knowledge Base Portlet. Click here to access it in GitHub.

In Liferay Portal 6 upgrade processes, the upgrade step classes for each schema version are in folders named after their schema version. For example, the Knowledge Base Portlet's upgrade step classes are in folders named `v1_0_0`, `v1_1_0`, `v1_2_0`, and so on. Each upgrade step class extends `UpgradeProcess` and overrides the `doUpgrade` method. The code in `doUpgrade` performs the upgrade. For example, the Knowledge Base Portlet's `v1_0_0/UpgradeRatingsEntry` upgrade step extends `UpgradeProcess` and performs the upgrade via the `updateRatingsEntries()` call in its `doUpgrade` method:

```
public class UpgradeRatingsEntry extends UpgradeProcess {

    @Override
    protected void doUpgrade() throws Exception {
        updateRatingsEntries();
    }

    ...

    protected void updateRatingsEntries() throws Exception {
        // Upgrade code
    }

    ...

}
```

The upgrade process classes are on the same level as the folders containing the upgrade steps and are also named after their schema version. For example, the Knowledge Base Portlet's upgrade process classes are named `UpgradeProcess_1_0_0`, `UpgradeProcess_1_1_0`, `UpgradeProcess_1_2_0`, and so on. Each upgrade process class also extends `UpgradeProcess` and runs the upgrade process in the `doUpgrade` method. It runs the upgrade process by passing the appropriate upgrade step to the `upgrade` method. For example,



v1_0_0
v1_1_0
v1_2_0
v1_3_0
v1_3_1
v1_3_2
v1_3_3
v1_3_4
UpgradeProcess_1_0_0.java
UpgradeProcess_1_1_0.java
UpgradeProcess_1_2_0.java
UpgradeProcess_1_3_0.java
UpgradeProcess_1_3_1.java
UpgradeProcess_1_3_2.java
UpgradeProcess_1_3_3.java
UpgradeProcess_1_3_4.java

Figure 18.2: The Knowledge Base Portlet's Liferay Portal 6.2 upgrade process.

the doUpgrade method in the Knowledge Base Portlet's UpgradeProcess_1_0_0 class runs the upgrade steps UpgradeRatingsEntry and UpgradeRatingsStats via the upgrade method:

```
@Override
protected void doUpgrade() throws Exception {
    upgrade(UpgradeRatingsEntry.class);
    upgrade(UpgradeRatingsStats.class);
}
```

Now that you know how Liferay Portal 6 upgrade processes are defined, you'll learn how to convert them to the new upgrade process framework in 7.0.

Converting your Liferay Portal 6 Upgrade Process to 7.0

So how do Liferay Portal 6 upgrade processes compare to those that use the new upgrade process framework in 7.0? First, the upgrade step classes are the same, so you can leave them unchanged. The big change in 7.0's new upgrade processes is that upgrade process classes no longer exist. Instead, you must combine your upgrade process classes' functionality into a single registrator class. Recall from the data upgrade process tutorial that registrators define an upgrade process that the upgrade process framework executes. Each registry.register call in the registrator registers the appropriate upgrade steps for each schema version. You must therefore transfer the functionality of your old upgrade process classes' doUpgrade methods into a registrator's registry.register calls.

For example, click here to see the Knowledge Base Portlet's new 7.0 upgrade process in GitHub.

Besides some additional upgrade step classes to handle changes made to the portlet for 7.0, the only difference in this upgrade process is that it contains a single registrator class, KnowledgeBaseServiceUpgrade, instead of multiple upgrade process classes. The KnowledgeBaseServiceUpgrade class, like all registrators, calls the appropriate upgrade steps for each schema version in its registry.register calls. For example, the first registry.register call registers the upgrade process for the 1.0.0 schema version:

```
registry.register(
        "com.liferay.knowledge.base.service", "0.0.1", "1.0.0",
        new com.liferay.knowledge.base.internal.upgrade.v1_0_0.
            UpgradeRatingsEntry(),
        new com.liferay.knowledge.base.internal.upgrade.v1_0_0.
            UpgradeRatingsStats());
```

Compare this to the above doUpgrade method from the corresponding Liferay Portal 6 upgrade process class UpgradeProcess_1_0_0. Both call the same upgrade steps. Click here to see the complete KnowledgeBaseServiceUpgrade registrator class and all its registry.register calls.

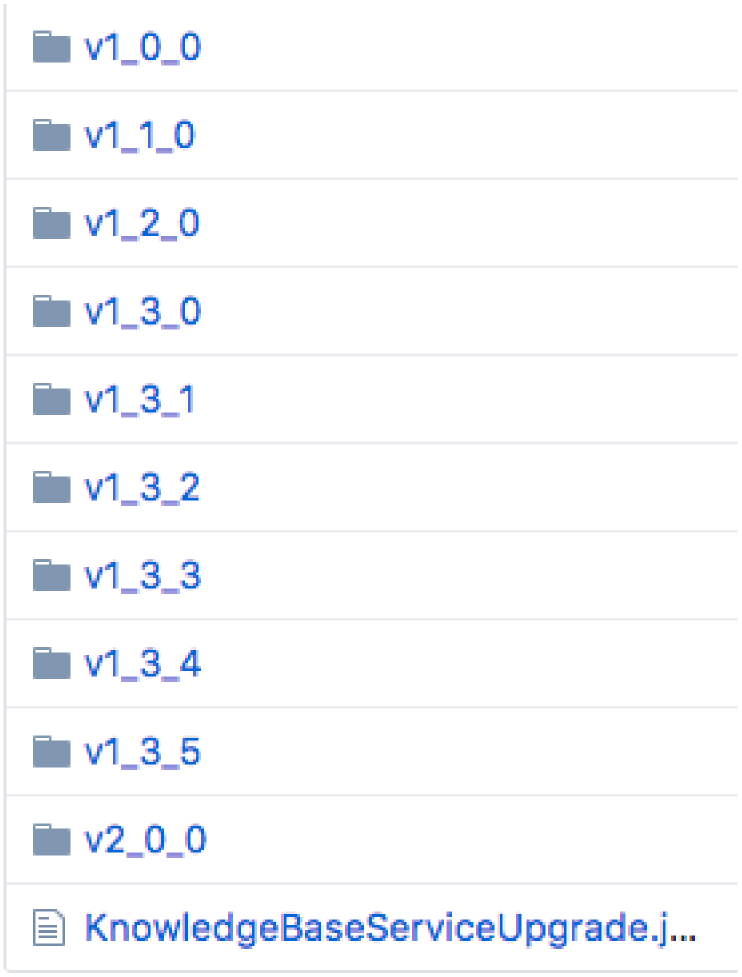
That's it! For instructions on creating new upgrade processes for 7.0, including complete steps on creating a registrator, click here.

Related Topics

Creating Data Upgrade Processes for Modules

Upgrading Plugins to 7.0

From Liferay Portal 6 to 7



v1_0_0
v1_1_0
v1_2_0
v1_3_0
v1_3_1
v1_3_2
v1_3_3
v1_3_4
v1_3_5
v2_0_0
KnowledgeBaseServiceUpgrade.j...

Figure 18.3: The Knowledge Base Portlet's new 7.0 upgrade process.

18.3 Migrating a Theme from the Plugins SDK to the Liferay Theme Generator

After you've upgraded your Liferay Portal 6 theme to 7.0, the Themes Generator offers enhanced development features and tools for optimizing your theme and streamlining theme management. To introduce one of its most powerful features, we'll pose some questions.

Questions:

- Do you want to make a temporary change to your theme's UI?
- Do you want to add the same UI modification to all of your themes without duplicating code?
- Do you want to share your new theme designs with a colleague?
- Do you want to test a new design concept in your theme without altering its code?

Answer: Themelets are the answer! Themelets are small, extendable, reusable modular pieces of code that let you make changes to your theme quickly. Because they are modular, you can use the same themelet for multiple themes!

Themelets are just one of the features you gain from migrating your existing Ant-based theme project to a Liferay Theme Generator project. The Liferay Theme Generator is a Node.js-based tool that gives you access to an array of theme Gulp tasks that facilitate developing and managing themes.

The *upgrade* Gulp task upgrades Liferay Portal 6 themes to 7.0. For details, refer to the Upgrading Themes tutorial.

In addition to the *upgrade* task, there are tasks for building and deploying themes and for interacting with deployed themes. For instance, you can automatically redeploy your theme as you make changes to it.

Do you periodically need to make changes to your theme's settings? No problem. You can configure your theme's settings through the command-line wizard that the Liferay Theme Generator provides. All you have to do is answer a few questions about the settings.

As you can see, the Liferay Theme Generator, Node.js, and Gulp development tools offer a lot to a Liferay theme developer.

This tutorial assumes that you have already installed the Liferay Theme Generator and that your upgraded theme was developed with the Plugins SDK. There are two ways you can migrate your Plugins SDK theme to the Theme Generator: importing your theme manually into the Theme Generator or converting it to a Theme Generator project from a Liferay Workspace. You'll learn how to import it manually first.

Importing Your Theme

The Liferay Theme Generator uses Yeoman to equip theme projects with the new development tools. Follow the steps below to set up your existing theme in such a project:

1. Navigate to the directory you want to import your theme into and run the following command:

   ```
   yo liferay-theme:import
   ```

   This runs the import sub-generator for the themes generator.

2. Enter the absolute path of the theme you want to import and press `Enter`.

```
**Note:** you must specify an absolute path, as the themes import
sub-generator does not support relative paths.
```

```
The theme's modified files (the files it modified from the base theme) are
copied and reorganized in a newly created  `src` directory. A `gulpfile.js`,
`liferay-theme.json`, `package.json` file  and a `node_modules` directory
are also added.

Next, the `gulp init` task runs and prompts you with a couple questions.
```

3. Enter the path to your app server.

4. Enter your site's URL (this can be your production site, development site, etc.), or press `Enter` to accept the default `http://localhost:8080`.

Your theme is now set up to use the Node.js build tools and theme Gulp tasks!

Migrating Themes to the Theme Generator Using Workspace

If you're a theme developer who wants to use Liferay Workspace to migrate your Plugins SDK theme to the Theme Generator, you can execute a single command to convert the theme project. Before beginning, make sure your Plugins SDK has been converted to a Liferay Workspace.

1. Using a command line tool, navigate to the root folder of your workspace.
2. Execute the following command to migrate your Plugins SDK theme to a Theme Generator theme:

   ```
   blade convert [PLUGINS_SDK_THEME_NAME]
   ```

   Blade CLI extracts the theme from the nested Plugins SDK folder and reorganizes it into a standard Theme Generator project. The converted theme is available in the workspace's `themes` folder.

That's it! Your Plugins SDK theme is now available as a Liferay Theme Generator project residing in Liferay Workspace.

Related Articles

Introduction to Themes
Liferay Theme Generator
Themelets
Upgrading Themes

18.4 Migrating a Theme from the Plugins SDK to Workspace

After you've adapted your Plugins SDK theme to Liferay DXP's API, you can continue maintaining it in the Plugins SDK. The Plugins SDK, however, is deprecated as of 7.0. Maintaining plugins in the SDK will become increasingly difficult. Liferay Workspace replaces the Plugins SDK, providing a comprehensive Gradle development environment and more. A simple command migrates Plugins SDK (Ant-based) themes to Workspace (Gradle-based) themes. From there you can build and deploy them to 7.0 as Web ARchives (WARs).

Running the Migration Command

Blade CLI's `convert` command migrates Plugins SDK themes to Workspace themes in Workspace's `wars` folder. Theme files are re-organized to follow the standard 7.0 theme folder structure.

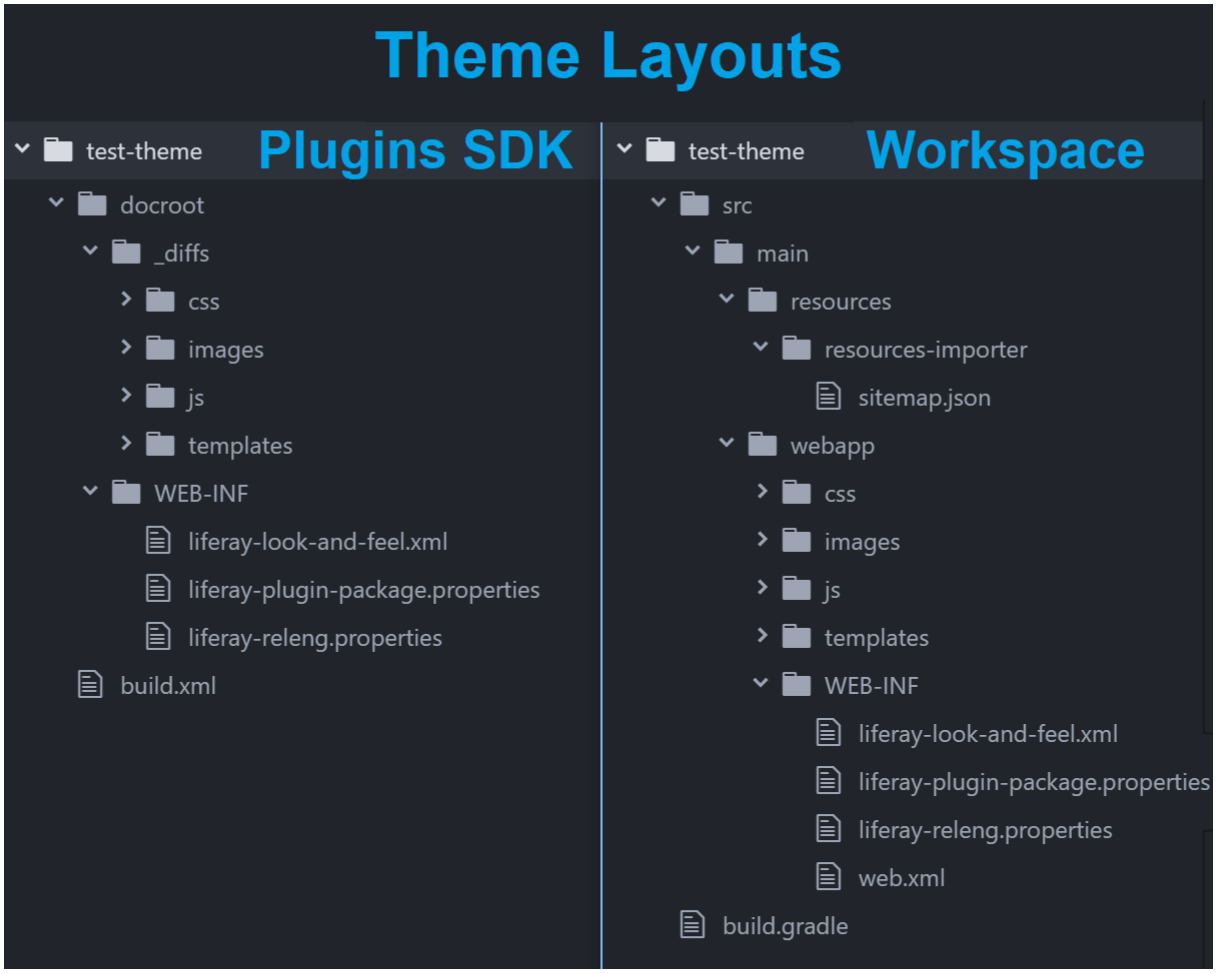
In a terminal, navigate to the Liferay Workspace root folder. Then pass your Plugins SDK theme's name to Blade CLI's `convert` command:

```
blade convert --themebuilder [THEME_PROJECT_NAME]
```

Blade CLI extracts the plugin from the Plugins SDK and reorganizes it in a standard web application project in Workspace's `wars` folder. Blade CLI uses the Theme Builder plugin to migrate your theme to a workspace. You can also migrate your Plugins SDK theme to a Liferay Theme Generator theme using Blade CLI. Follow the Migrating Themes to the Theme Generator Using Workspace tutorial for more information.

The image below shows the theme files before and after they're migrated to Workspace.

From your theme's new location, you can deploy it to 7.0 and maintain it using Workspace Gradle tasks. Welcome to your theme's new home in Workspace!



```
test-theme
  docroot
    _diffs
      css
      images
      js
      templates
    WEB-INF
      liferay-look-and-feel.xml
      liferay-plugin-package.properties
      liferay-releng.properties
  build.xml
```

```
test-theme
  src
    main
      resources
        resources-importer
          sitemap.json
      webapp
        css
        images
        js
        templates
        WEB-INF
          liferay-look-and-feel.xml
          liferay-plugin-package.properties
          liferay-releng.properties
          web.xml
  build.gradle
```

Figure 18.4: The `convert` command migrates a Plugins SDK theme project to a Workspace theme project.

Related Topics

Migrating a Theme from the Plugins SDK to the Theme Generator

Workspace Development lifecycle

Workspace Gradle Tasks

18.5 Customization with Ext Plugins

Ext plugins are deprecated for 7.0 and should only be used if absolutely necessary. They are deployable to Liferay Portal 7.0 CE GA4+.

The following app servers should be used for Ext plugin development in Liferay DXP:

- Tomcat 8.0

In most cases, Ext plugins are no longer necessary. There are, however, certain cases that require the use of an Ext plugin. Liferay only supports the following Ext plugin use cases:

- Providing custom implementations for any beans declared in Liferay DXP's Spring files (when possible, use service wrappers instead of an Ext plugin). 7.0 removed many beans, so make sure your overridden beans are still relevant if converting your legacy Ext plugin.
- Overwriting a class in a 7.0 core JAR. For a list of core JARs, see the Finding Core Liferay DXP Artifacts section.
- Modifying Liferay DXP's `web.xml` file.
- Adding to Liferay DXP's `web.xml` file.

Ext plugins are powerful tools used to extend Liferay DXP. They, however, increase the complexity of your Liferay DXP instance and are not recommended unless there is absolutely no other way to accomplish your task. 7.0 provides many extension points that let you customize almost every detail of Liferay DXP. If there's a way to customize what you want with an extension point, do it that way instead. See the More Extensible, Easier to Maintain section for more details on the advantages of using Liferay DXP's extension points.

Before deciding to use an Ext plugin, weigh the cost. Ext plugins let you use internal APIs and even let you overwrite Liferay DXP core files. This puts your deployment at risk of being incompatible with security, performance, or feature updates released by Liferay. When upgrading to a new version of Liferay DXP (even if it's a maintenance version or a service pack), you have to review all changes and manually modify your Ext plugin to merge your changes with Liferay DXP's. Additionally, Ext plugins aren't hot deployable. To deploy an Ext plugin, you must restart your server. Additional steps are also required to deploy or redeploy to production systems.

In this tutorial, you'll learn how to

- Create an Ext plugin
- Develop an Ext plugin
- Deploy an Ext plugin in Production

Before diving into creating an Ext plugin, however, first consider if an Ext plugin is even necessary at all.

Making the Decision to Use Ext Plugins

There are many parts of Liferay DXP that now provide an extension point via OSGi bundle. You should follow this three step process to decide whether an Ext plugin is necessary:

1. Find the OSGi extension point that you need. You can follow the Finding Extension Points tutorial as a guide.

2. If an OSGi extension point does not exist, use an Ext plugin.

3. Research new extension points after every release of Liferay DXP. When a new version of Liferay DXP provides the extension point you need, always use the extension point to replace the existing Ext plugin.

So how do you find an OSGi extension point?

Your first step is to examine the custom module projects that extend popular Liferay DXP extension points stored in the Liferay Blade Samples repository. For more information on these sample projects, see the Liferay Sample Projects tutorial. Usable extension points are also documented throughout Liferay's Developer Network categorized by the Liferay DXP section involved. For example, Overriding MVC Commands and Customizing the Product Menu are articles describing how to extend a Liferay DXP extension point. Want to learn how to override module JSPs or Liferay DXP core JSPs? Those processes are documented too!

There are a few corner cases where you may need an Ext plugin to customize a part of Liferay DXP that does not provide an extension point. Refer to the top of this tutorial for Ext plugin use cases supported by Liferay.

Note: In previous versions of Liferay Portal, you needed an Ext plugin to specify classes as portal property values (e.g., `global.starup.events.my.custom.MyStartupAction`), since the custom class had to be added to the portal class loader. This is no longer the case in 7.0 since all lifecycle events can use OSGi services with no need to edit these legacy properties.

Now that you know how to make an informed decision on using Ext plugins, you'll learn how to create one next.

Creating an Ext Plugin

An Ext plugin is a powerful tool for extending Liferay DXP. Because it increases the complexity of your Liferay DXP installation, you should only use an Ext plugin if you're sure you can't accomplish your goal in a different way. You can only create Ext plugins from a Plugins SDK. If you're using a Liferay Workspace to create your @project@ projects, you can merge a Plugins SDK instance into the workspace.

Creating an Ext Plugin Using Liferay @ide@

Follow the steps below, replacing the project name with your own, to create a custom Ext plugin:

1. Go to *File → New → Liferay Plugin Project*.
2. Enter your project's name and its display name. In the figure below, *example* is used for both. Note that the Display name field is automatically filled in with the capitalized version of the Project name.
3. Select the *Ant (liferay-plugins-sdk)* option for your build type.
4. Select the *Ext* plugin type. Then click *Next*.
5. If you have not yet configured a Plugins SDK, select *Next* and choose the Plugins SDK for which you want to generate the Ext plugin. Once you have your Plugins SDK configured, click *Finish*.

The Plugins SDK automatically appends `-ext` to the plugin project name when it creates your Ext plugin's project folder. Your Ext plugin project is added to your configured Plugins SDK instance and is also available via @ide@'s Package Explorer for further development.

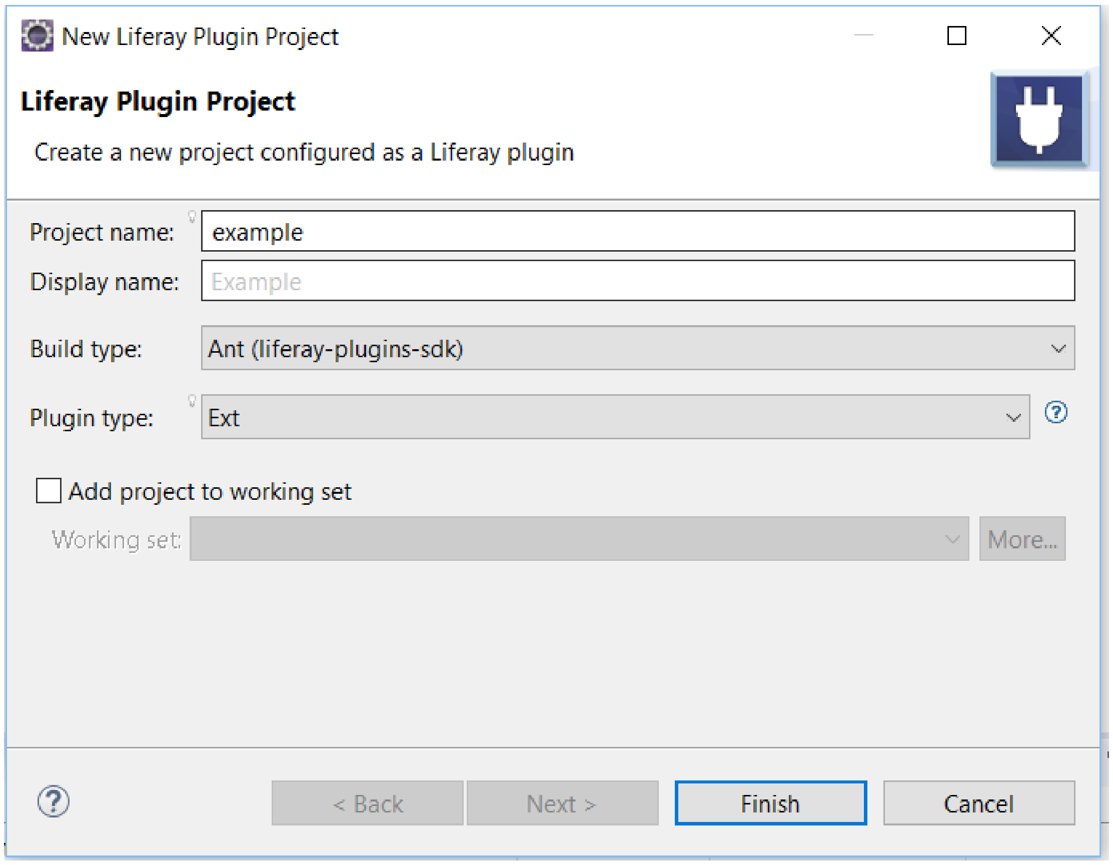
Creating an Ext Plugin Using the Command Line

To create a new Ext plugin from the command line, navigate to the `ext` folder in your Liferay Plugins SDK and enter the command below appropriate for your operating system. The two arguments after the `create` command are the project name and display name, respectively. The examples below use the project name *example* and the display name *Example*. Make sure to specify values you want to use for your Ext plugin's project name and display name.

In Linux or Mac OS, enter:

```
./create.sh example "Example"
```

In Windows, enter:


New Liferay Plugin Project

Liferay Plugin Project

Create a new project configured as a Liferay plugin

Project name: example

Display name: Example

Build type: Ant (liferay-plugins-sdk)

Plugin type: Ext

Add project to working set

Working set:

Figure 18.5: You can create an Ext plugin project with Liferay @ide@.

```
create.bat example "Example"
```

A BUILD SUCCESSFUL message from Ant tells you there's a new folder (e.g., folder example-ext for a project named *example*) inside your Plugins SDK's ext folder. The Plugins SDK automatically appends -ext to the project name.

Anatomy of an Ext Plugin

There are a few things to note about an Ext plugin's folder structure. Below is a listing of an Ext folder structure:

- [project name]-ext/
  - docroot/
    * WEB-INF/
      - ext-impl/
      - src/

      - ext-lib/
      - global/

· portal/

· ext-service/
· src/

· ext-util-bridges/
· src/

· ext-util-java/
· src/

· ext-util-taglib/
· src/

· ext-web/
· docroot/

Here are detailed explanations of the `/docroot/WEB-INF/` subfolders:

- `ext-impl/src`: Contains the custom implementation classes and classes that override core classes from `portal-impl.jar`.
- `ext-lib/global`: Contains libraries to be copied to the application server's global classloader upon deployment of the Ext plugin.
- `ext-lib/portal`: Contains libraries to be copied inside Liferay's main application. These libraries are usually necessary because they're invoked from classes you add in `ext-impl/src`.
- `ext-kernel/src`: Contains classes that should be available to other plugins. In advanced scenarios, this folder can be used to hold classes that overwrite classes from `portal-kernel.jar`.
- `ext-web/docroot`: Contains the web application's configuration files, including `WEB-INF/struts-config-ext.xml`, which lets you customize Liferay's core struts classes. Note that for 7.0, there are very few entities left to override in the `struts-config.xml` file. Any JSPs that you're customizing also belong here.
- `ext-util-bridges`, `ext-util-java` and `ext-util-taglib`: These folders are needed only in scenarios where you must customize these Liferay libraries: `util-bridges.jar`, `util-java.jar` and `util-taglib.jar`, respectively. If you're not customizing any of these libraries, you can ignore these folders.

By default, several files are also added to the plugin. Here are the most significant files:

- `build.xml`: The Ant build file for the Ext plugin project.
- `docroot/WEB-INF/liferay-plugin-package.properties`: Contains plugin-specific properties, including the plugin's display name, version, author, and license type.
- `docroot/WEB-INF/ext-web/docroot/WEB-INF` files:

- `portlet-ext.xml`: Used to overwrite the definition of a build-in portlet. The `portlet.xml` file contains very few portlet configurations in 7.0 because portlets were modularized and moved out of core. To override this file, copy the complete definition of the desired portlet from `portlet-custom.xml` in Liferay's source code, then apply the necessary changes.
- `liferay-portlet-ext.xml`: This file is similar to `portlet-ext.xml`, but is for additional definition elements specific to Liferay. The `liferay-portlet.xml` file contains very few definition elements in 7.0 because portlets were modularized and moved out of core. To override the remaining definition elements, copy the complete definition of the desired portlet from `liferay-portlet.xml` in Liferay's source code, then apply the necessary changes.
- `struts-config-ext.xml` and `tiles-defs-ext.xml`: These files are used to customize the struts actions used by core portlets. Since most portlets were modularized and moved out of core in 7.0, the `struts-config.xml` and `tiles-defs.xml` files are sparsely used.
- `web.xml`: Used to overwrite web application configurations and servlet information for 7.0.

Note: After creating an Ext plugin, remove the files from `docroot/WEB-INF/ext-web/docroot/WEB-INF` that you don't need to customize. Removing files you're not customizing prevents incompatibilities that could manifest from Liferay DXP updates.

Great! You've now created an Ext plugin and are familiar with its folder structure and its most significant files. Next, you'll use your Ext plugin to customize Liferay DXP.

Developing an Ext Plugin

An Ext plugin changes Liferay DXP itself when the plugin is deployed; it's not a separate component that you can easily remove at any time. For this reason, the Ext plugin development process is different from other plugin types. It's important to remember that once an Ext plugin is deployed, some of its files are copied *inside* the Liferay installation; the only way to remove the changes is by *redeploying* an unmodified Liferay application. You're also responsible for checking that patches and fix packs do not conflict with your Ext plugin.

The 7.0 compatible Plugins SDK is designed to only develop/deploy one Ext plugin. This means that all your customizations should live inside one Ext plugin. The Plugins SDK does not check for conflicts among multiple Ext plugins stored in the `/ext` folder, so do **not** develop/deploy multiple Ext plugins at once.

The Plugins SDK lets you deploy and redeploy your Ext plugin during your development phase. Redeployment involves *cleaning* (i.e., removing) your application server and unzipping your specified Liferay bundle to start from scratch. That way, any changes made to the Ext plugin during development are properly applied, and files removed from your plugin by previous changes aren't left behind in the Liferay DXP application. Because of this added complexity, you should use another plugin type to accomplish your goals whenever possible.

Before digging in to the details, here's an overview of the Ext plugin development processes described below:

- *Configure* your Plugins SDK environment to develop Ext plugins for Liferay DXP on your application server.
- *Deploy* and *publish* your Ext plugins for the first time.
- *Redeploy* normally or use a *clean redeployment* process after making changes to your Ext plugins.
- *Package* your Ext plugins for distribution.
- A list of advanced customization techniques.

It's time to learn each step of the development process.

Set Up the Build Environment

Before deploying an Ext plugin, you must edit the `build.[username].properties` file in the root folder of your Plugins SDK. If the file doesn't yet exist, create it now. Substitute `[username]` with your user ID on your computer. Once you've opened your build properties file, add the following properties–make sure the individual paths reflect the right locations on your system:

```
ext.work.dir=[work]

app.server.dir=[work]/liferay-ce-portal-[version]/[app server]

app.server.zip.name=[...]/liferay-ce-portal-[app server].zip
```

Your `app.server.zip.name` property must specify the path to your Liferay DXP `.zip` file. The `work` folder you specify for the `ext.work.dir` property is where you've unzipped your Liferay DXP runtime. The `app.server.dir` property must point to your application server's directory in your work folder. Look in your Liferay DXP bundle at the path to the application server to determine the value to use for your `app.server.dir` property.

For example, if your work folder is `C:/work`, specify it as your `ext.work.dir` property's value. If your Liferay DXP bundle .zip file is `C:/downloads/liferay-ce-portal-tomcat-7.0-ga4-[timestamp].zip`, set the `app.server.zip.name` property to that file's path. If the *relative path* to the application server *within* the Liferay DXP bundle `.zip` file is `liferay-ce-portal-7.0-ga4\tomcat-8.0.32`, specify an `app.server.dir` property value `C:/work/liferay-ce-portal-7.0-ga4/tomcat-8.0.32`.

Next you'll modify the Ext plugin you created and deploy it.

Initial Deployment

Your environment is set up and you're ready to start customizing. Once you're finished, you can deploy the plugin.

Deploy the Plugin Now you'll learn how to deploy your plugin using Liferay @ide@ or the command line.

Deploying in Liferay @ide@ Drag your Ext plugin project from your Package Explorer onto your server to deploy your plugin. Liferay @ide@ automatically restarts the server for the server to detect and publish your Ext plugin.

After deploying, if you don't see your customizations in the portal, your Ext plugin may not have successfully deployed. To confirm whether the Ext plugin deploys successfully, try deploying from the command line; the Ant deployment targets report success or failure.

Deploying via the Command Line

1. Open a command line window in your Ext plugin project folder and enter one of these commands:

   ```
   ant deploy

   ant direct-deploy
   ```

   The `direct-deploy` target deploys all changes directly to the appropriate folders in the Liferay application. The `deploy` target creates a `.war` file with your changes and then deploys it to your server. Either way, your server must be restarted after the deployment. Using `direct-deploy` is preferred for deploying an Ext plugin during development if your application server supports it. Direct deploy doesn't work in WebLogic or WebSphere application server environments.

A `BUILD SUCCESSFUL` message indicates your plugin is now being deployed. Your console window running Liferay DXP shows a message like this:

```
Extension environment for [your project]-ext has been applied. You must
reboot the server and redeploy all other plugins
```

If any changes applied through the Ext plugin affect the deployment process itself, you must redeploy all other plugins. Even if the Ext plugin doesn't affect the deployment process, it's a best practice to redeploy all your other plugins following initial deployment of the Ext plugin.

The `ant deploy` target builds a `.war` file with your changes and copies them to the auto-deploy folder inside the Liferay DXP installation. When the server starts, it detects the `.war` file, inspects it, and copies its contents to the appropriate destinations inside the deployed and running Liferay application.

2. Restart the server so that it detects and publishes your Ext plugin.

   Once your server restarts, open Liferay DXP to see the customizations you made with your Ext plugin.

Frequently, you'll want to add further customizations to your original Ext plugin. You can make unlimited customizations to an Ext plugin that has already been deployed, but the redeployment process is different from other project types. You'll learn more on redeploying your Ext plugin next.

Redeployment

So far, Ext plugin development has been similar to the development of other plugin types. You've now reached the point of divergence. When the plugin is first deployed, some of its files are copied into the Liferay DXP installation. After changing an Ext plugin, you'll either *clean redeploy* or *redeploy*, depending on the modifications you made to your plugin following the initial deployment. You'll learn about each redeployment method and when to use each one.

Clean Redeployment If you removed part(s) of your plugin, if there are changes to your plugin that can affect plugin deployment, or if you want to start with a clean Liferay DXP environment, *undeploy* your plugin and *clean* your application server before redeploying your Ext plugin. By cleaning the application server, the existing Liferay DXP installation is removed and the bundle specified in your Plugins SDK environment is unzipped in its place. See the instructions below to learn more about this process.

Using Liferay @ide@

1. Remove the plugin from the server. While selecting the Ext plugin in the *Servers* view, select the plugin's *Remove* option.

2. Clean the application server. Remember to shut down the application server before cleaning it. This is required on Windows because the server process locks the installed Liferay bundle's files.

   While selecting the Ext plugin project in the *Package Explorer* view, select the plugin's *Liferay → Clean App Server...* option.

3. Start the Liferay server.

4. Drag the Ext plugin and drop it onto the Liferay server.

5. While selecting the Liferay server in the *Servers* view, click the *Publish* option.

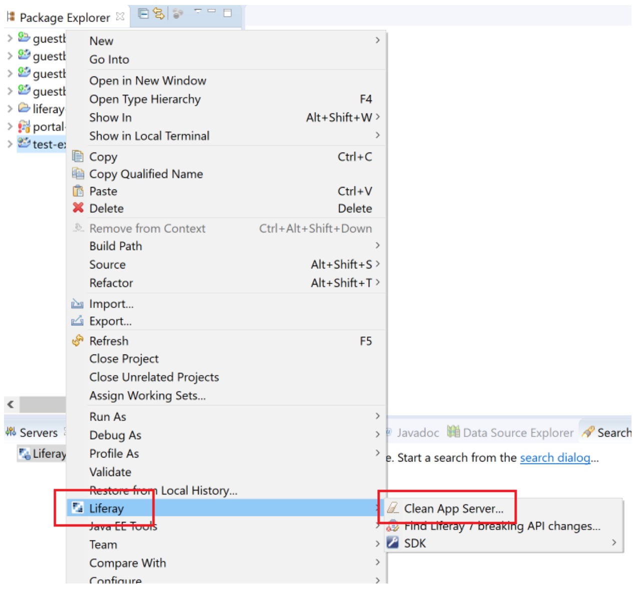


Figure 18.6: How to clean your app server

Using the Command Line

1. Stop the Liferay DXP server.

2. To deploy your Ext plugin, enter the following commands into your console:

```
cd [your-plugin-ext]
ant clean-app-server
ant direct-deploy
```

3. Start the Liferay DXP server.

Redeployment If you only added to your plugin or made modifications that don't affect the plugin deployment process, you can redeploy the Ext plugin. Follow the steps based on the tool you're using.

Using Liferay @ide@ Right-click your plugin located underneath your server and select *Redeploy*.

Using the Command Line Using the same procedure as for initial deployment. Open a command line window in your Ext plugin project's directory and execute either `ant deploy` or `ant direct-deploy`.

After your plugin is published to Liferay DXP, verify that your customizations are available.

Next you'll learn how to package an Ext plugin for distribution and production.

Distribution

Once you're finished developing an Ext plugin, you can package it in a `.war` file for distribution and production deployment.

Using Liferay @ide@ With your Ext plugin project selected in the *Package Explorer* view, select the project's *Liferay* → *SDK* → *war* option.

Using the Command Line From your Ext plugin's directory execute

```
ant war
```

The `.war` file is written to your `[liferay-plugins]/dist` directory.

Now that you've learned the basics of Ext plugin development, you'll look at some advanced customizations.

Using Advanced Configuration Files

Liferay DXP uses several internal configuration files for its own architecture; in addition, there are configuration files for the libraries and frameworks Liferay DXP depends on, like Struts and Spring. Configuration could be accomplished using fewer files with more properties in each, but maintenance and use is made easier by splitting up the configuration properties into several files. For advanced customization needs, it may be useful to override the configuration specified in multiple configuration files. Liferay DXP provides a clean way to do this from an Ext plugin without modifying the original files.

All the configuration files in Liferay DXP are listed below by their path in an Ext plugin folder. Here are descriptions of what each file is for and the path to the original file in Liferay DXP:

- `ext-impl/src/META-INF/ext-model-hints.xml`
  - **Description:** Overrides the default properties of the fields of the data models used by Liferay DXP's core portlets. These properties determine how the form fields for each model are rendered.
  - **Original file in Liferay DXP:** `portal-impl/src/META-INF/portal-model-hints.xml`
- `ext-impl/src/META-INF/ext-spring.xml`
  - **Description:** Overrides the Spring configuration used by Liferay DXP and any of its core portlets. It's most commonly used to configure specific data sources or swap the implementation of a default service with a custom one.
  - **Original file in Liferay DXP:** `portal-impl/src/META-INF/*-spring.xml`
- `ext-impl/src/META-INF/portal-log4j-ext.xml`
  - **Description:** Allows overriding the Log4j configuration. It can be used to configure appenders for log file location, naming, and rotation. See the Log4j XML Configuration Primer. Increasing or decreasing the log level of a class or class hierarchy is best done outside of an Ext plugin, in Liferay DXP's' UI or a Log4j XML file in a module or the `osgi/log4j` folder.
  - **Original file in Liferay:** `portal-impl/src/META-INF/portal-log4j.xml`
- `ext-web/docroot/WEB-INF/portlet-ext.xml`
  - **Description:** Overrides the core portlets' declarations. It's most commonly used to change the init parameters or the roles specified.
  - **Original file in Liferay DXP:** `portal-web/docroot/WEB-INF/portlet-custom.xml`
- `ext-web/docroot/WEB-INF/liferay-portlet-ext.xml`
  - **Description:** Overrides the Liferay-specific core portlets' declarations. It core portlets included in Liferay DXP. Refer to the liferay-portlet-app_7_0_0.dtd file for details on all the available options. Use this file with care; the code of the portlets may be assuming some of these options to be set to certain values.

    - **Original file in Liferay DXP:** `portal-web/docroot/WEB-INF/liferay-portlet.xml`

- `ext-web/docroot/WEB-INF/liferay-display.xml`
    - **Description:** Overrides the portlets that are shown in the interface for adding applications and the categories in which they're organized.
    - **Original file in Liferay DXP:** `portal-web/docroot/WEB-INF/liferay-display.xml`

You'll learn how to deploy your Ext plugin in production next.

Deploying in Production

Often, you can't use Ant to deploy web applications in production or pre-production environments. Additionally, some application servers such as WebSphere or WebLogic have their own deployment tools and don't work with Liferay DXP's auto-deployment process. Here are two methods to consider for deploying and redeploying an Ext plugin in production.

Method 1: Redeploying Liferay's Web Application

You can use this method in any application server that supports auto-deployment; What's the benefit? The only artifact that needs to be transferred to the production system is your Ext plugin's `.war` file, produced using the `ant war` target. This `.war` file is usually small and easy to transport. Execute these steps on the server:

1. Redeploy Liferay DXP:

   If this is your first time deploying your Ext plugin to this server, skip this step. Otherwise, start by executing the same steps you first used to deploy Liferay DXP on your app server. If you're using a bundle, unzip it again. If you installed Liferay DXP manually on an existing application server, you must redeploy the Liferay DXP `.war` file and copy both the libraries required globally by Liferay DXP and your Ext plugin to the appropriate folder within the application server.

2. Copy the Ext plugin `.war` into the auto-deploy folder. For a bundled Liferay DXP distribution, the `deploy` folder is in Liferay's *root* folder of your bundle (e.g., `liferay-ce-portal-[version]/`).

3. Once the Ext plugin is detected and deployed by Liferay DXP, restart your Liferay DXP server.

Method 2: Generate an Aggregated WAR File

Some application servers don't support auto-deploy; WebSphere and WebLogic are two examples. With an aggregated `.war` file, the Ext plugin is merged before deployment to production. A single `.war` file then contains Liferay DXP plus the changes from your Ext plugin. Before you deploy the aggregated Liferay DXP `.war` file, copy the dependency `.jar` files for Liferay DXP and your Ext plugin to the global application server class loader in the production server. The precise location varies from server to server; refer to Deployment to get the details for your application server.

To create the aggregated `.war` file, deploy the Ext plugin first to the Liferay DXP bundle you are using in your development environment. Once it's deployed, restart the server so that the plugin is fully deployed and shut it down again. Now the aggregated file is ready.

Create a `.war` file by zipping the Liferay web application folder from within the app server. Copy into your application server's global classpath all of the libraries on which your Ext plugin depends.

Now, perform these actions on your server:

1. Redeploy Liferay DXP using the aggregated `.war` file.

2. Stop the server and copy the new version of the global libraries to the appropriate directory in the application server.

Next, you'll learn about Liferay's licensing and contributing standards.

Licensing and Contributing

Liferay DXP is Open Source software licensed under the LGPL 2.1 license. If you reuse any code snippet and redistribute it, whether publicly or to a specific customer, make sure your modifications are compliant with the license. One common way is to make the source code of your modifications are available to the community under the same license. Make sure you read the license text yourself to find the option that best fits your needs.

If your goal in making changes is fixing a bug or improving Liferay DXP, it could be of interest to a broader audience. Consider contributing it back to the project. That benefits all users of the product including you, since you won't have to maintain the changes with each newly released version of Liferay DXP. You can notify Liferay of bugs or improvements at issues.liferay.com. Check out the Participate section of portal.liferay.dev, to learn all the ways that you can contribute to the Liferay Portal project.

In summary, an Ext plugin is a powerful way to extend Liferay DXP. There are no limits to what you can customize, so use it carefully. Before using an Ext plugin, see if you can implement all or part of the desired functionality through Application Display Templates or a different plugin type. OSGi modules offer you a lot of extension capabilities themselves, without introducing the complexity that's inherent with Ext plugins. If you need to use an Ext plugin, make your customization as small as possible and follow the instructions in this tutorial carefully to avoid issues.

Chapter 19

Developing a Web Application

In this Learning Path, you'll create the Liferay Guestbook Web Application from scratch using tools like Liferay @ide@ and BLADE tools. As you create this application, you'll learn how to create a back-end database, web services, a security model, UI, and more using all the best practices and standards for Liferay DXP. Completing this Learning Path will prepare you to write your own application and further explore Liferay's APIs.

To develop a web application with Liferay, start at the beginning: setting up a Liferay development environment. Though you can use anything from a text editor and the command line, to your Java IDE of choice, Liferay provides Liferay @ide@ to optimize development on Liferay's platform. It integrates Liferay's BLADE tools for modular development.

Once your development environment is set up, you'll begin creating the application. From modeling data to Service Builder, you'll learn everything you need to know to create and run your application.

From there you'll see everything from UI standards to providing remote services. Once everything is completed and wrapped up with a bow, you can distribute the application on Marketplace.

Let's Go!

19.1 Development Setup Overview

```
<p>Development Setup Overview<br>Step 1 of 1</p>
```

Liferay's development tools aim to help you get started fast. The basic steps for installing Liferay @ide@ are

- Download a Liferay @ide@ bundle.
- Unzip the downloaded package to a location on your system.
- Start @ide@.

You'll follow these steps, and then generate an environment for developing your first Liferay DXP application.

Installing a Liferay @ide@ Bundle

To install Liferay @ide@, follow these steps:

1. Download and install the Java Development Kit (JDK). Liferay DXP runs on Java. The JDK is required because you'll be developing Liferay DXP apps in Liferay @ide@. The JDK is an enhanced version of the Java Environment used for developing new Java technology.

2. Download Liferay @ide@. Installing it is easy: unzip it to a convenient location on your system.

3. To run Liferay @ide@, execute the `eclipse` executable.

The first time you start Liferay @ide@, it prompts you to select an Eclipse workspace. If you specify a folder where no workspace currently exists, Liferay @ide@ creates a new workspace in that folder. Follow these steps to create a new workspace:

1. When prompted, indicate your workspace's path. Name your new workspace `guestbook-workspace` and click *OK*.

2. When Liferay @ide@ first launches, it presents a welcome page. Click the *Workbench* icon to continue.

Nice job! Your development environment is installed and your workspace is set up.

Creating a Liferay Workspace

Now you'll create another kind of workspace–a Liferay Workspace. By holding and managing your Liferay DXP projects, a Liferay Workspace provides a simplified, straightforward way to develop Liferay DXP applications. In the background, a Liferay Workspace uses Blade CLI and Gradle to manage dependencies and organize your build environment. Note that to avoid configuration issues, you can only create one Liferay Workspace for each Eclipse Workspace.

Follow these steps to create a Liferay Workspace in Liferay @ide@:

1. Select *File* → *New* → *Liferay Workspace Project*. Note: you may have to select *File* → *New* → *Other*, then choose *Liferay Workspace Project* in the *Liferay* category.

   A *New Liferay Workspace* dialog appears, which presents several configuration options.

2. Give your workspace the name `com-liferay-docs-guestbook`.

3. Next, choose your workspace's location. Leave the default setting checked. This places your Liferay Workspace inside your Eclipse workspace.

4. Check the *Download Liferay bundle* checkbox to automatically download and unzip a Liferay DXP instance in your workspace. When prompted, name the server `liferay-tomcat-bundle`.

5. Click *Finish* to create your Liferay Workspace. This may take a while because Liferay Liferay DXP downloads the Liferay DXP bundle in the background.

A dialog appears prompting you to open the Liferay Workspace perspective. Click *Yes*, and your perspective switches to Liferay Workspace.

Congratulations! Your development environment is ready! Next, you'll get started developing your first Liferay DXP application.

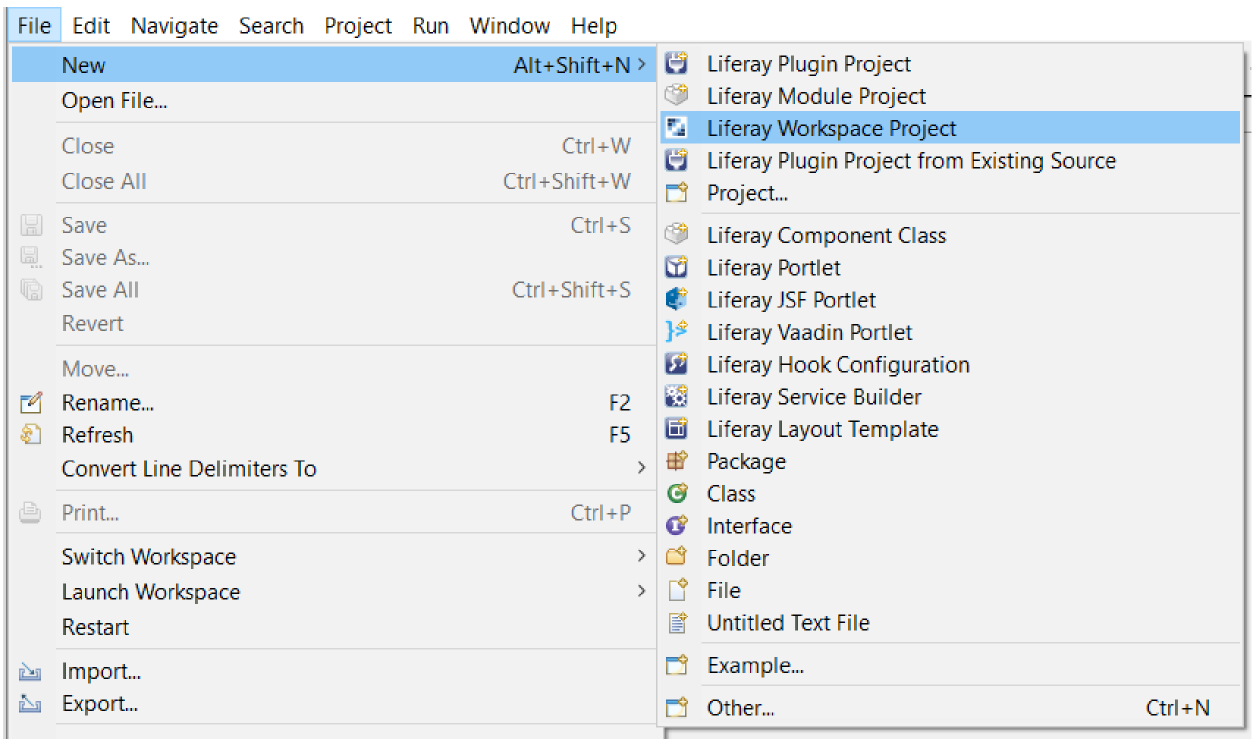
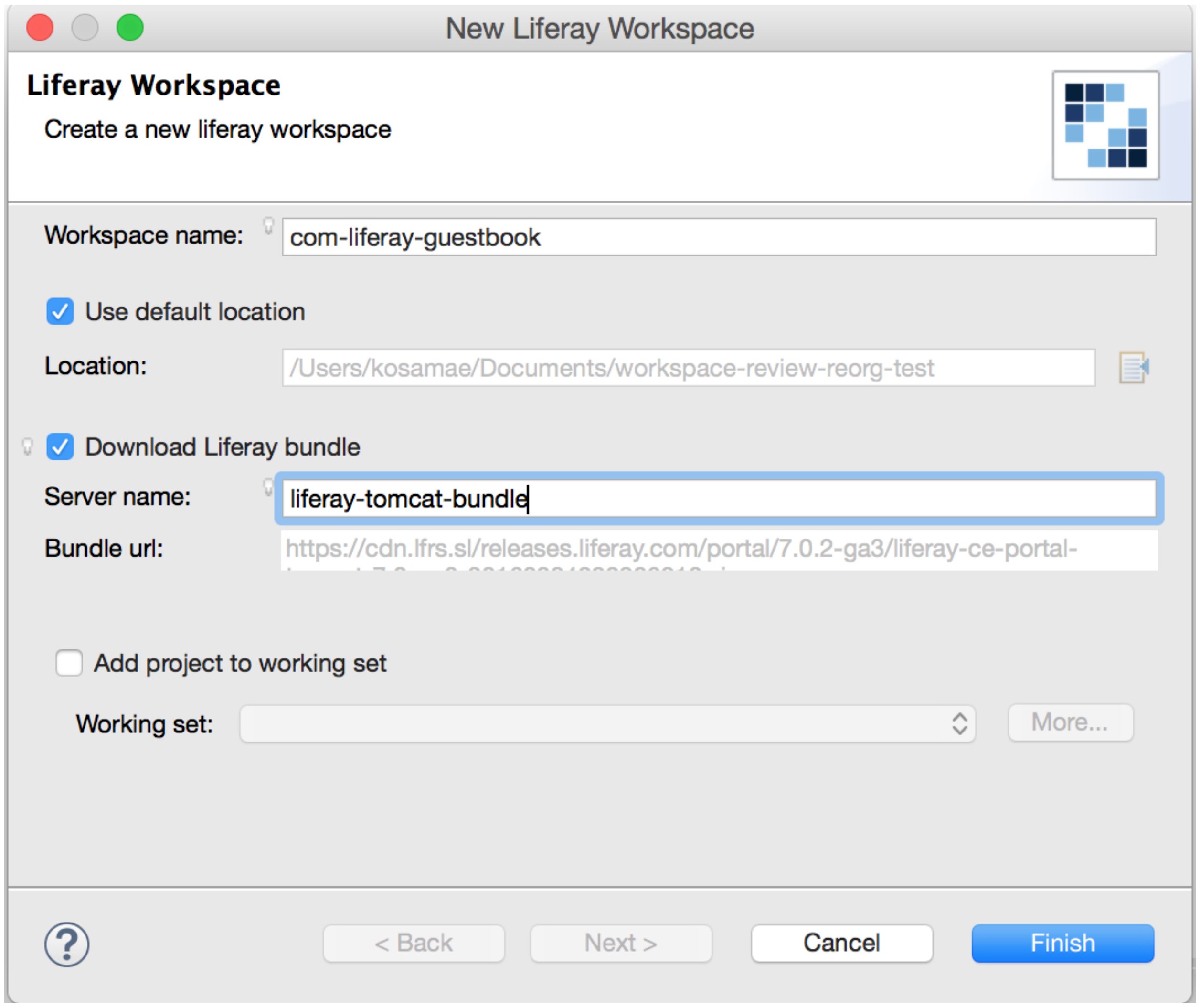


Figure 19.1: By selecting *Liferay Workspace*, you begin the process of creating a new workspace for your Liferay DXP projects.



Workspace name: com-liferay-guestbook

✓ Use default location

Location: /Users/kosamae/Documents/workspace-review-reorg-test

✓ Download Liferay bundle

Server name: liferay-tomcat-bundle

Bundle url: https://cdn.lfrs.sl/releases.liferay.com/portal/7.0.2-ga3/liferay-ce-portal-

Add project to working set

Working set:

Figure 19.2: Liferay @ide@ provides an easy-to-follow menu to create your Liferay Workspace.

Chapter 20

Creating a Working Prototype

So far, you've installed and set up Liferay @ide@, and created a Liferay Workspace. Next, you'll create your application and start adding basic features to it. Here's what you'll do:

- Create your application and deploy it to your Liferay DXP instance.
- Create a functional button for adding and removing guestbook entries.
- Create a form for users to create and edit guestbook entries.
- Create a UI for displaying guestbook entries.
- Implement a prototype storage system (to be replaced later) for storing guestbook entries.

At the end, you'll have a fully functional prototype application ready to be enhanced later! There's no time like now to get started.

Let's Go!

20.1 Writing Your First Liferay DXP Application

`<p>Developing Your First Portlet<br>Step 1 of 8</p>`

It's easy to get started developing your first Liferay DXP application. Here, you'll learn step-by-step how to create your project and deploy your application to Liferay DXP. Before you know it, you'll have your application deployed alongside those that come with Liferay DXP.

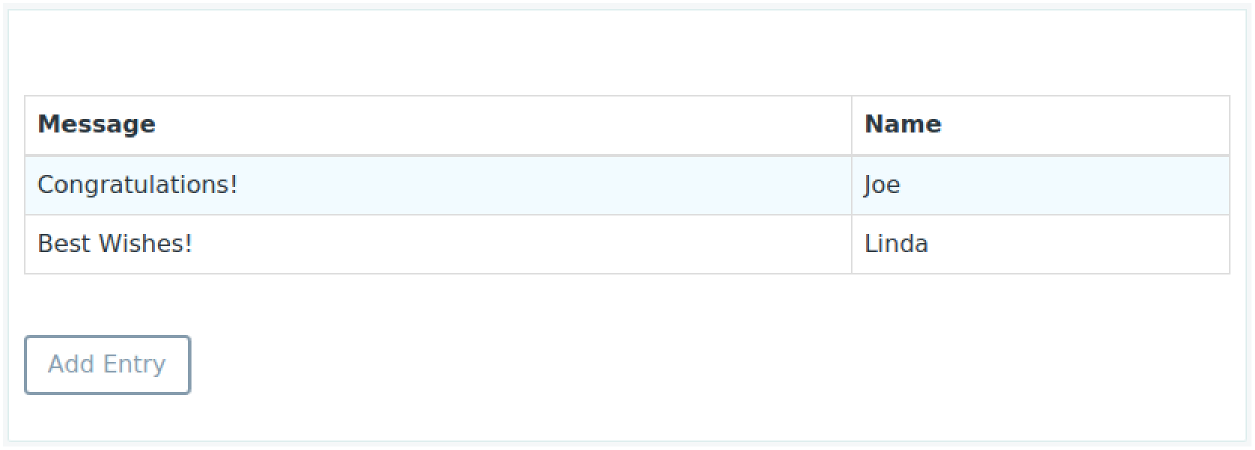
Your first application is simple: you'll build a guestbook application that looks like this:

By default, it shows guestbook messages that various users leave on your website. To add a message, you click the *Add Entry* button to show a form that lets you enter and save a message.

Ready to write your first Liferay DXP application?

Creating Your First Liferay DXP Application

Your first step is to create a *Liferay Module Project*. Modules are the core building blocks of Liferay DXP applications. Every application is made from one or more modules. Each module encapsulates a functional piece of an application, and then multiple modules form a complete application. There's good reason for this: modules let you swap out code implementations more or less at will. This makes your applications easy to maintain and upgrade.



Message	Name
Congratulations!	Joe
Best Wishes!	Linda

Figure 20.1: You'll create this simple application.

These modules are OSGi modules. The OSGi container in Liferay DXP can run any OSGi module. Each module is packaged as a JAR file that contains a manifest file. The manifest is needed for the container to recognize the module. Technically, a module that contains only a manifest is still valid. Of course, such a module wouldn't be very interesting.

Now you'll create your first module. For the purpose of this Learning Path, you'll create your modules inside your Liferay Workspace. Follow these instructions to create your first Liferay Module Project:

1. In the Project Explorer in Liferay @ide@, right click on your Liferay Workspace and select *New → Liferay Module Project*.

2. Complete the first screen of the wizard with the following information:

    - Enter `guestbook-web` for the Project name.
    - Use the *Gradle* Build type.
    - Select `mvc-portlet` for the Project Template.

    Click *Next*.

3. On the second screen of the wizard, enter `Guestbook` for the component class name, and `com.liferay.docs.guestbook.portlet` for the package name. Click *Finish*.

Note that it may take a while for @ide@ to create your project, because Gradle downloads your project's dependencies for you during project creation. Once this is done, you have a module project named `guestbook-web`. The `mvc-portlet` template configured the project with the proper dependencies and generated all the files you need to get started:

- The portlet class (in the package you specified)
- JSP files (in `/src/main/resources`)
- Language properties (also in `/src/main/resources`)

Your new module project is a *portlet* application. Next, you'll learn exactly what a portlet is.

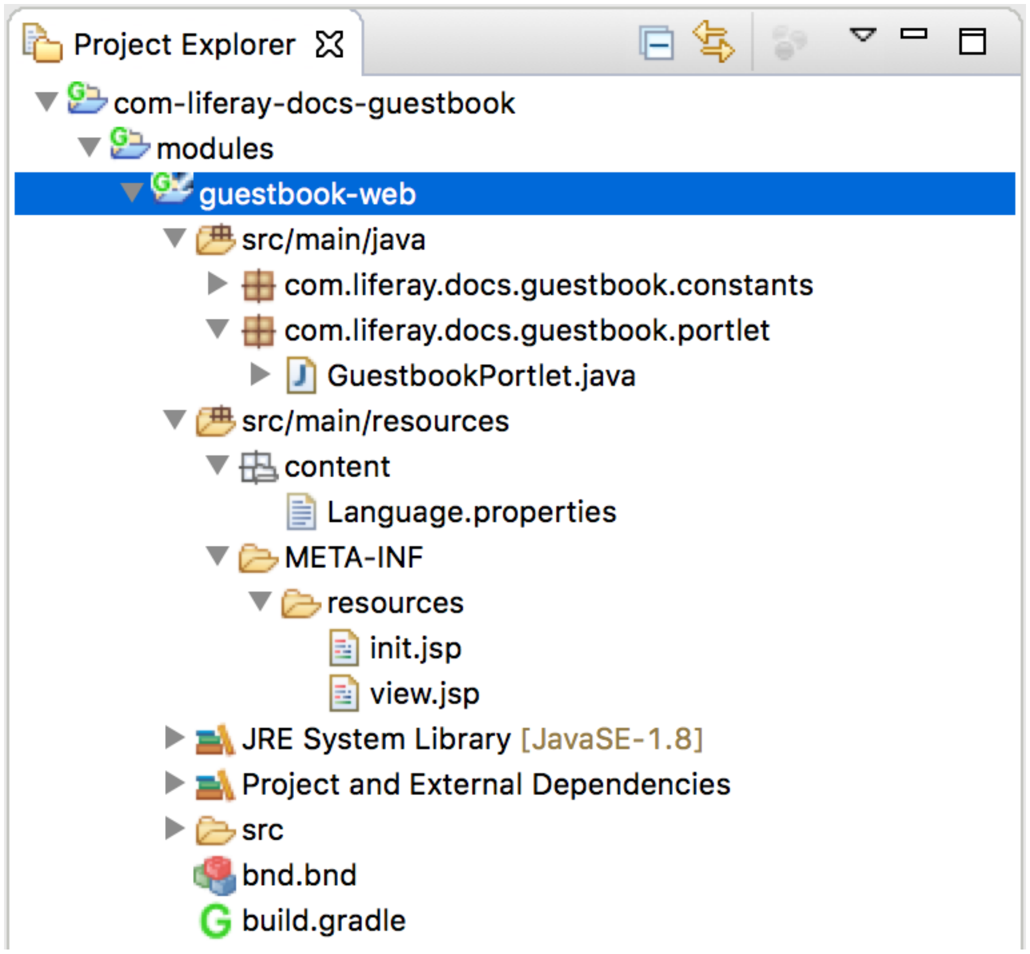


Figure 20.2: Your new module project appears in your Liferay Workspace's `modules` folder.

What is a Portlet?

When you access a web site, you interact with an application. That application could be simple: it might only show you one piece of information, such as an article. The application might be complex: you could be doing your taxes, entering lots of data into an application that calculates whether you owe or are due a refund. These applications run on a *platform* that provides application developers the building blocks they need to make applications.

Liferay DXP provides a platform that contains common features needed by today's applications, including user management, security, user interfaces, services, and more. Portlets are one of those basic building blocks. Often a web application takes up the entire page. If you want, you can do this with applications in Liferay DXP as well. Portlets, however, allow Liferay DXP to serve many applications on the same page at the same time. Liferay DXP's framework takes this into account at every step. For example, features like platform-generated URLs exist to support Liferay's ability to serve multiple applications on the same page.

What is a Component?

Portlets created in Liferay Module Projects are generated as *Components*. If the module (sometimes also called a *bundle*) encapsulates pieces of your application, the component is the object that contains the core functionality. A Component is an object that is managed by a component framework or container.

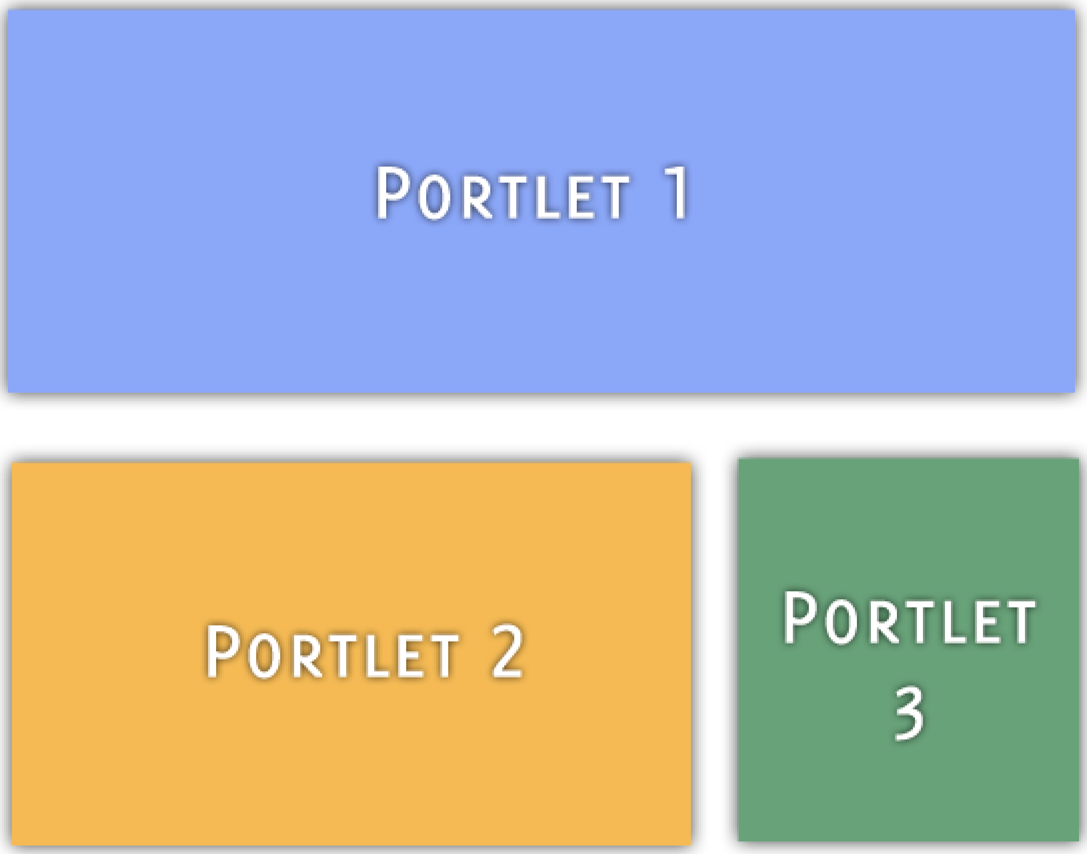


Figure 20.3: Many Liferay applications can run at the same time on the same page.

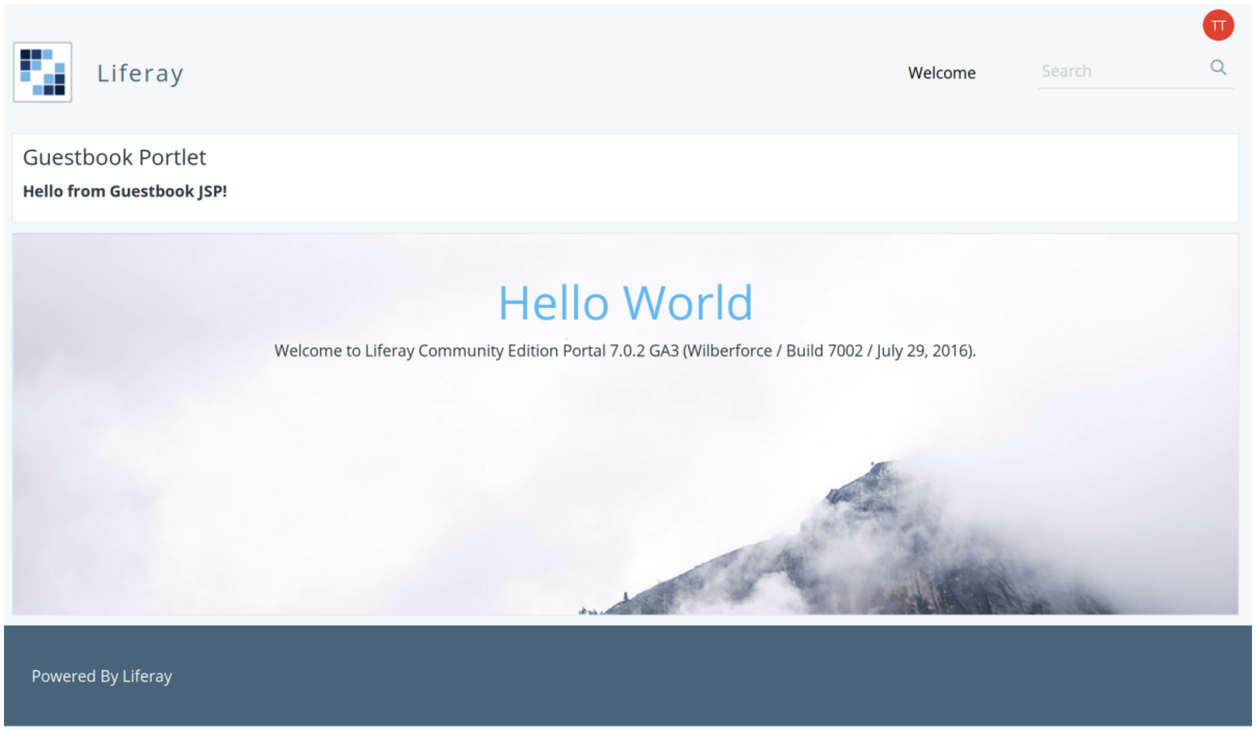
Components are deployed inside modules, and they're created, started, stopped, and destroyed as needed by the container. What a perfect model for a web application! It can be made available only when needed, and when it's not, the container can make sure it doesn't use any resources needed by other components.

In this case, you created a Declarative Services (DS) component. With Declarative Services, you declare that an object is a component, and you define some data about the component so the container knows how to manage it. A default configuration was created for you; you'll examine it later.

Deploying the Application

Even though all you've done is generate it, the `guestbook-web` project is ready to be built and deployed to Liferay DXP. Make sure that your server is running, and if it isn't, select it in @ide@'s Servers pane and click the start button. After it starts, drag and drop the `guestbook-web` project from the Project Explorer to the server. If this is your first time starting Liferay DXP, you'll go through a short wizard to set up your server. In this wizard, make sure you use the default database (Hypersonic). Although this database isn't intended for production use, it works fine for development and testing.

Next, check that your application is available in Liferay DXP. Open a browser, navigate to your portal (http://localhost:8080 by default), and add your application to a page. To add an application to a page, click the *Add* button in the upper right hand corner (it looks like a plus sign), and then select *Applications*. In the Applications list, your application should appear in the Sample category. Its name should be `guestbook-web-module Portlet`.



Liferay
Welcome
Search
Guestbook Portlet
Hello from Guestbook JSP!
Hello World
Welcome to Liferay Community Edition Portal 7.0.2 GA3 (Wilberforce / Build 7002 / July 29, 2016).
Powered By Liferay

Figure 20.4: This is the default Liferay homepage. It contains several portlets, including the initial version of the Guestbook application that you created.

Now you're ready to jump in and start developing your Guestbook portlet.

20.2 Creating an Add Entry Button

```
<p>Developing Your First Portlet<br>Step 2 of 8</p>
```

A guestbook application is pretty simple, right? People come to your site and post their names and brief messages. Other users can read these entries and post their own.

When you created your project, it generated a file named `view.jsp` in your project's `src/main/resources/META-INF/resources` folder. This file creates the default view for users when the portlet is added to the page. Right now it only contains some sample content:

```
<%@ include file="/init.jsp" %>

<p>
    <b><liferay-ui:message key="guestbook-web.caption"/></b>
</p>
```

First, `view.jsp` imports `init.jsp`. By convention, imports and tag library declarations in Liferay DXP portlet applications should be in an `init.jsp` file. The other JSP files in the application import `init.jsp`. This lets you handle JSP dependency management in a single file.

Besides importing `init.jsp`, `view.jsp` displays a message defined by a language key. This key and its value are declared in your project's `src/main/resources/content/Language.properties` file.

It's time to start developing the Guestbook application. First, users need a way to add a guestbook entry. In `view.jsp`, follow these steps to add this button:

1. Remove everything under the include for `init.jsp`.

2. Below the include, add the following AlloyUI tags to display an Add Entry button inside of a button row:

   ```
   <aui:button-row>
       <aui:button value="Add Entry"></aui:button>
   </aui:button-row>
   ```

You can use aui tags in `view.jsp` since `init.jsp` declares the AlloyUI tag library by default (as well as other important imports and tags):

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>

<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
<%@ taglib uri="http://liferay.com/tld/portlet" prefix="liferay-portlet" %>
<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>

<portlet:defineObjects />

<liferay-theme:defineObjects />
```

Your application now displays a button instead of a message, but the button doesn't do anything. Next, you'll create a URL for your button.

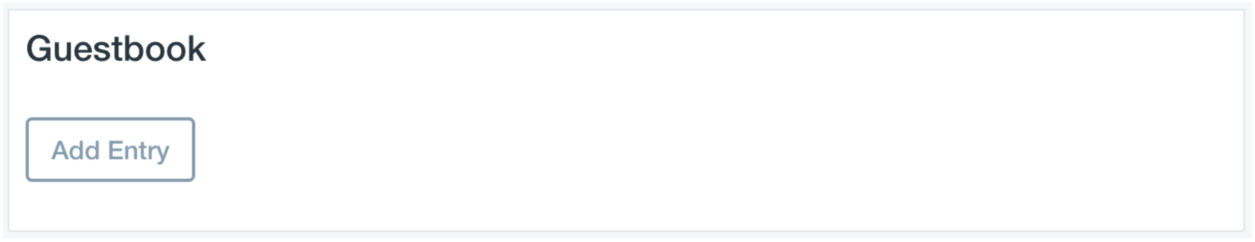


Figure 20.5: Your new button is awesome, but it doesn't work yet.

20.3 Generating Portlet URLs

```
<p>Developing Your First Portlet<br>Step 3 of 8</p>
```

Recall that users can place multiple portlets on a single page. As a developer, you have no idea what other portlets will share a page with yours. This means that you can't define URLs for various functions in your application like you may be used to.

For example, consider a Calendar application that a user puts on the same page as a Blog application. To implement the functionality for deleting calendar events and blog entries in the respective application, both application developers append the del parameter to the URL, and give it a primary key value so the application can look up and delete the calendar event or blog entry. Since both applications read this parameter, their delete functionality clashes.

System-generated URLs are Liferay DXP's solution to this. By generating a unique URL parameter for each piece of functionality, Liferay DXP lets multiple applications with the same or similar functionality coexist in perfect harmony. Unfortunately, for this to work in your portlet you must manually add support for it. Fortunately, doing so is very straightforward.

In view.jsp, follow these steps to enable system-generated URLs in your portlet:

1. Add these tags below <%@ include file="/init.jsp" %>, but above the <aui:button-row> tag:

```
<portlet:renderURL var="addEntryURL">
    <portlet:param name="mvcPath" value="/edit_entry.jsp"></portlet:param>
</portlet:renderURL>
```

2. Add this attribute to the <aui:button> tag, before value="Add Entry":

```
onClick="<%= addEntryURL.toString() %>"
```

   Your view.jsp page should now look like this:

```
<%@ include file="/init.jsp" %>

<portlet:renderURL var="addEntryURL">
    <portlet:param name="mvcPath" value="/edit_entry.jsp"></portlet:param>
</portlet:renderURL>

<aui:button-row>
    <aui:button onClick="<%= addEntryURL.toString() %>" value="Add Entry"></aui:button>
</aui:button-row>
```

The <portlet:renderURL> tag's var attribute creates the addEntryURL variable to hold the system-generated URL. The <portlet:param> tag defines a URL parameter to append to the URL. In this example, a URL parameter named mvcPath with a value of /edit_entry is appended to the URL.

Note that your GuestbookPortlet class (located in your guestbook-web module's com.liferay.docs.guestbook.portlet package) extends Liferay's MVCPortlet class. In a Liferay MVC portlet, the mvcPath URL parameter indicates a page within your portlet. To navigate to another page in your portlet, use a portal URL with the parameter mvcPath to link to the specific page.

In the example above, you created a renderURL that points to your application's edit_entry.jsp page, which you haven't yet created. Note that using an AlloyUI button to follow the generated URL isn't required. You can use any HTML construct that contains a link. Users can click your button to access your application's edit_entry.jsp page. This currently produces an error since no edit_entry.jsp exists yet. Creating edit_entry.jsp is your next step.

20.4 Linking to Another Page

```
<p>Developing Your First Portlet<br>Step 4 of 8</p>
```

In the same folder your view.jsp is in, create the edit_entry.jsp file:

1. Right-click your project's src/main/resources/META-INF/resources folder and choose *New → File*.

2. Name the file `edit_entry.jsp` and click *Finish*.

3. Add this line to the top of the file:

   ```
   <%@ include file="init.jsp" %>
   ```

   Remember, it's a best practice to add all JSP imports and tag library declarations to a single file that's imported by your application's other JSP files. For `edit_entry.jsp`, you need these imports to access the portlet tags that create URLs and the Alloy tags that create the form.

4. You'll create two URLs: one to submit the form and one to go back to the `view.jsp`. To create the URL to go back to `view.jsp`, add the following tag below the first line you added:

   ```
   <portlet:renderURL var="viewURL">
       <portlet:param name="mvcPath" value="/view.jsp"></portlet:param>
   </portlet:renderURL>
   ```

Next, you must create a new URL for submitting the form. Before you do, some explanation is in order.

20.5 Triggering Portlet Actions

```
<p>Developing Your First Portlet<br>Step 5 of 8</p>
```

Recall that portlets run in a portion of a page, and a page can contain multiple portlets. Because of this, portlets have *phases* of operation. Here, you'll learn about the most important two. The first phase is the one you've already used: the *render* phase. All this means is that the portlet draws itself, using the JSPs you write for it.

The other phase is called the *action* phase. This phase runs once, when a user triggers a portlet action. The portlet performs whatever action the user triggered, such as performing a search or adding a record to a database. Then, based on what happened in the action, the portlet goes back to the render phase and re-renders itself according to its new state.

To save a guestbook entry, you must trigger a portlet action. For this, you'll create an action URL.

Add the following tag in `edit_entry.jsp` after the closing `</portlet:renderURL>` tag:

```
<portlet:actionURL name="addEntry" var="addEntryURL"></portlet:actionURL>
```

You now have the two required URLs for your form.

20.6 Creating a Form

```
<p>Developing Your First Portlet<br>Step 6 of 8</p>
```

The form for creating guestbook entries is pretty simple. All you need are two fields: one for the name of the person submitting the entry, and one for the entry itself.

Add the following tags to the end of your `edit_entry.jsp` file:

```
<aui:form action="<%= addEntryURL %>" name="<portlet:namespace />fm">
        <aui:fieldset>
            <aui:input name="name"></aui:input>
            <aui:input name="message"></aui:input>
        </aui:fieldset>

        <aui:button-row>
            <aui:button type="submit"></aui:button>
            <aui:button type="cancel" onClick="<%= viewURL.toString() %>"></aui:button>
        </aui:button-row>
</aui:form>
```

Save `edit_entry.jsp` and redeploy your application. If you refresh the page and click the *Add Entry* button, your form appears. If you click the *Cancel* button, it works! However, don't try the *Save* button yet. You haven't yet created the action that saves a guestbook entry, so clicking *Save* produces an error.

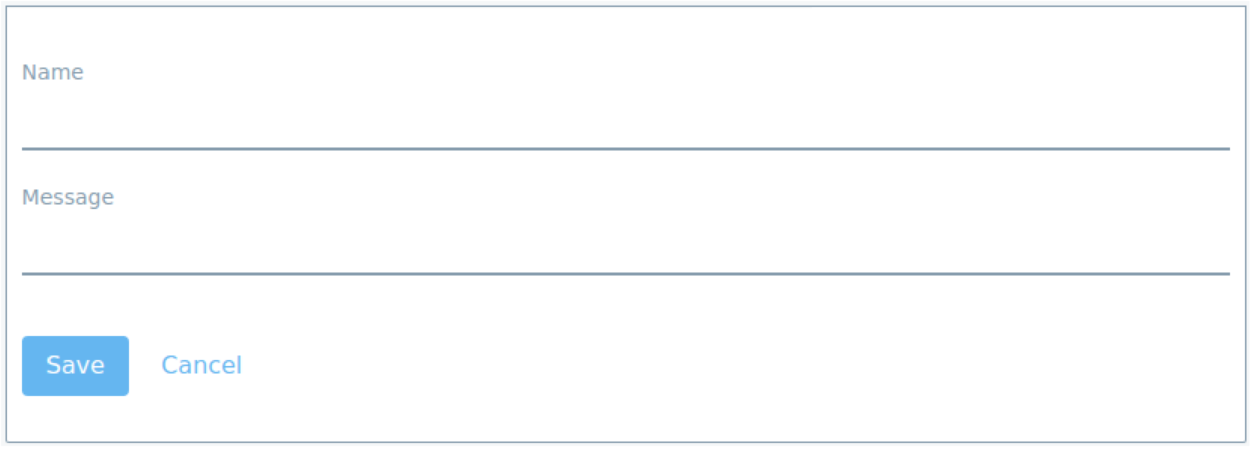


Figure 20.6: This is the Guestbook application's form for adding entries.

Implementing the portlet action (what happens when the user clicks *Save*) is your next task.

20.7 Implementing Portlet Actions

```
<p>Developing Your First Portlet<br>Step 7 of 8</p>
```

When users submit the form, your application stores the form data for display in the guestbook. To keep this first application simple, you'll implement this using a part of the Portlet API called Portlet Preferences. Normally, of course, you'd use a database. Liferay DXP's Service Builder tool eliminates a great deal of complexity when working with databases. For now, however, you can create the first iteration of your guestbook application using portlet preferences.

To make your portlet do anything other than re-render itself, you must implement portlet actions. An action defines some processing, usually based on user input, that the portlet must perform before it renders itself. In the case of the guestbook portlet, the action you'll implement next saves a guestbook entry that a user typed into the form. Saved guestbook entries can be retrieved and displayed later.

Since you're using Liferay's MVC Portlet framework, you have an easy way to implement actions. Portlet actions are implemented in the portlet class, which acts as the controller. In the form you just created, you made an action URL, and you called it `addEntry`. To create a portlet action, you create a method in the portlet class with the same name. `MVCPortlet` calls that method when a user triggers its matching URL.

1. Open GuestbookPortlet. The project template generated this class when you created the portlet project.

2. Create a method with the following signature:

   ```
   public void addEntry(ActionRequest request, ActionResponse response) {

   }
   ```

3. Press [CTRL]+[SHIFT]+O to organize imports and import the required javax.portlet.ActionRequest and javax.portlet.ActionResponse classes.

You've now created a portlet action. It doesn't do anything, but at least you won't get an error now if you submit your form. Next, you should make the action save the form data.

Because of the limitations of the portlet preferences API, you must store each guestbook entry as a String in a string array. Since your form has two fields, you must use a delimiter to determine where the user name ends and the guestbook entry begins. The caret symbol (^) makes a good delimiter because users are highly unlikely to use that symbol in a guestbook entry.

Note: The portlet preferences API is used here for prototyping purposes only. In most cases, you'll need a more robust solution for storing data. You'll learn how to implement such a solution later in the *Service Builder* section.

The following method implements adding a guestbook entry to a portlet preference called guestbook-entries:

```
public void addEntry(ActionRequest request, ActionResponse response) {
    try {
        PortletPreferences prefs = request.getPreferences();

        String[] guestbookEntries = prefs.getValues("guestbook-entries",
                new String[1]);

        ArrayList<String> entries = new ArrayList<String>();

        if (guestbookEntries[0] != null) {
            entries = new ArrayList<String>(Arrays.asList(prefs.getValues(
                    "guestbook-entries", new String[1])));
        }

        String userName = ParamUtil.getString(request, "name");
        String message = ParamUtil.getString(request, "message");
        String entry = userName + "^" + message;

        entries.add(entry);

        String[] array = entries.toArray(new String[entries.size()]);

        prefs.setValues("guestbook-entries", array);

        try {
            prefs.store();
        }
        catch (IOException ex) {
            Logger.getLogger(GuestbookPortlet.class.getName()).log(
                    Level.SEVERE, null, ex);
        }
        catch (ValidatorException ex) {
            Logger.getLogger(GuestbookPortlet.class.getName()).log(
                    Level.SEVERE, null, ex);
```

```
        }

    }
    catch (ReadOnlyException ex) {
        Logger.getLogger(GuestbookPortlet.class.getName()).log(
                Level.SEVERE, null, ex);
    }
}
```

1. Replace your existing addEntry method with the above method.

2. Press [CTRL]+[SHIFT]+O to organize imports and select the javax.portlet.PortletPreferences and java.util.logging.Logger when prompted (not their Liferay equivalents).

First, the preferences are retrieved. Then the guestbook-entries preference is retrieved and converted to an ArrayList so that you can add an entry without worrying about exceeding the size of the array. Next, the name and message fields from your form are retrieved. Note that Liferay's ParamUtil class makes it very easy to retrieve URL parameters.

Finally, the fields are combined into a String delimited by a caret, and the new entry is added to the ArrayList, which is then converted back to an array so it can be stored as a preference. The try/catch blocks are required by the portlet preferences API.

This isn't the normal way to use portlet preferences, but it provides a quick and easy way for you to store guestbook entries in this first version of your application. In a later step, you'll implement a robust way to store guestbook entries in a database.

The next and final feature to implement is a mechanism for viewing guestbook entries.

20.8 Displaying Guestbook Entries

```
<p>Developing Your First Portlet<br>Step 8 of 8</p>
```

To display guestbook entries, you must do the reverse of what you did to store them: retrieve them from portlet preferences, loop through them, and present them on the page. The best way to do this with MVC Portlet is to use the Model-View-Controller paradigm. You already have the view (your JSP files) and your controller (your portlet class). Now you need your model.

Creating Your Model

1. Create a new package called com.liferay.docs.guestbook.model. To do this, right-click your src/main/java folder and select *New → Package*. Then enter the package name in the dialog box that appears.

2. Next, create your model class. This is a simple class that models a guestbook entry. To do this, right-click your new package and select *New → Class*. Name the class Entry, and click *Finish*.

   You now have a Java class for your guestbook entries. Next, you'll give it the fields you need to store entries.

3. Create two private String variables: name and message.

```
private String name;
private String message;
```

4. Right-click a blank area of the editor and select *Source → Generate Getters and Setters*. Click *Select All* in the dialog that pops up, and then click *OK*.

5. Next, provide two constructors: one that initializes the class with no values for the two fields, and one that takes the two fields as parameters and sets their values:

   ```
   public Entry() {
      this.name = null;
      this.message = null;
   }

   public Entry(String name, String message) {
      setName(name);
      setMessage(message);
   }
   ```

Your completed model class looks like this:

```
package com.liferay.docs.guestbook.model;

public class Entry {

    private String name;
    private String message;

    public Entry() {
        this.name = null;
        this.message = null;
    }

    public Entry(String name, String message) {
        setName(name);
        setMessage(message);
    }

    public String getName() {
        return name;
    }

    public String getMessage() {
        return message;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setMessage(String message) {
        this.message = message;
    }

}
```

Now that you have your model, you have an easy way of encapsulating guestbook entries so they can be processed by the controller layer and displayed by the view layer. Your next step is to enhance the controller (your portlet class) so that guestbook entries are processed and ready to display when users see the guestbook application.

Customizing How Your Application is Rendered

As mentioned earlier, your application is using two portlet phases: render and action. To make the guestbook show the saved guestbook entries when users view the application, you need to customize your portlet's render functionality, which it's currently inheriting from its parent class, `MVCPortlet`.

1. Open GuestbookPortlet and add the following method below your addEntry method:

   ```
   @Override
   public void render(RenderRequest renderRequest, RenderResponse renderResponse)
       throws PortletException, IOException {

       PortletPreferences prefs = renderRequest.getPreferences();
       String[] guestbookEntries = prefs.getValues("guestbook-entries", new String[1]);

       if (guestbookEntries[0] ≠ null) {
           List<Entry> entries = parseEntries(guestbookEntries);
           renderRequest.setAttribute("entries", entries);
       }

       super.render(renderRequest, renderResponse);
   }
   ```

   This method retrieves the guestbook entries from the configuration, converts it to a List of Entry objects, and places that List into the request object. It then calls the parent class's render method.

2. Beneath the render method, add the following method that converts the array to a List of your model objects:

   ```
   private List<Entry> parseEntries(String[] guestbookEntries) {
       List<Entry> entries = new ArrayList<Entry>();

       for (String entry : guestbookEntries) {
           String[] parts = entry.split("\\^", 2);
           Entry gbEntry = new Entry(parts[0], parts[1]);
           entries.add(gbEntry);
       }

       return entries;
   }
   ```

3. Press [CTRL]+[SHIFT]+O to organize imports.

Note: When you are prompted to choose imports, here are some guidelines:

- Always use org.osgi... packages instead of aQute.bnd...
- Generally use java.util... or javax.portlet... packages.
- You never use java.awt... in this project.
- Only use com.liferay... when it is for a Liferay specific implementation or your custom implementation of a concept.

For example:

- If you are given the choice between javax.portlet.Portlet and com.liferay.portlet.Portlet choose javax.portlet.Portlet.
- If you are given the choice between org.osgi.component and aQute.bnd.annotation.component choose org.osgi.component

- However, if you are given the choice between `java.util.Map.Entry` and `com.liferay.docs.guestbook.model.Entry` (the custom class you created) choose `com.liferay.docs.guestbook.model.Entry`

If at some point you think you chose an incorrect import, but you're not sure what it might be, you can erase all of the imports from the file and press [CTRL]+[SHIFT]+O again and see if you can identify where you went wrong.

As you can see, this method splits the entries in the `String` array into two parts based on the caret (^) character.

Now that you have your controller preparing your data for display, your next step is to implement the view so users can see guestbook entries.

Displaying Guestbook Entries

Liferay's development framework makes it easy to loop through data and display it nicely to the end user. You'll use a Liferay UI construct called *Search Container* to make this happen.

1. Add these tags to your `view.jsp` in between the `</portlet:renderURL>` and `<aui:button-row>` tags:

```
<jsp:useBean id="entries" class="java.util.ArrayList" scope="request"/>

<liferay-ui:search-container>
    <liferay-ui:search-container-results results="<%= entries %>" />

    <liferay-ui:search-container-row
        className="com.liferay.docs.guestbook.model.Entry"
        modelVar="entry"
    >
        <liferay-ui:search-container-column-text property="message" />

        <liferay-ui:search-container-column-text property="name" />
    </liferay-ui:search-container-row>

    <liferay-ui:search-iterator />
</liferay-ui:search-container>
```

Save your work, deploy your application, and try adding some guestbook entries.

Awesome! You've finished your working prototype! You have a working application that adds and saves guestbook entries.

The way you're saving the entries isn't the best way to persist data in your application. Next, you'll use Service Builder to generate your persistence classes and the methods you need to store your application data in the database.

Chapter 21

Generating the Back-end

So far, you have a prototype application that uses Liferay's Model-View-Controller (MVC) portlet framework. MVC is a great design pattern for web applications because it splits your application into three parts (the model, the view, and the controller). This lets you swap out those parts if necessary.

A *persistence* layer and a *service* layer are added to these three parts of your application. To get the prototype working, you used Portlet Properties to create a rudimentary persistence layer. Since this isn't a long-term solution, you'll now replace that layer by persisting your guestbooks and their entries to a database.

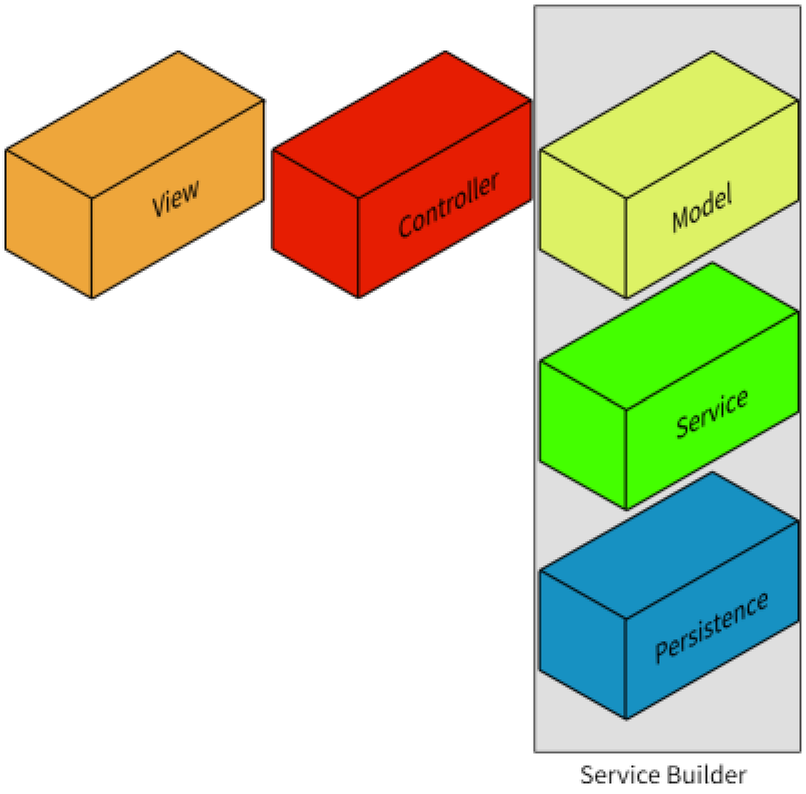


Figure 21.1: Service Builder generates the shaded layers of your application.

Service Builder is Liferay's code generation tool for defining object models and mapping those models to

SQL databases. By defining your model in a single XML file, you can generate your object model (the M in MVC), your service layer, and your persistence layer all in one shot. At the same time, you can generate web services (more on that later) and support every database Liferay DXP supports.

Ready to begin?

Let's Go!

21.1 What is Service Builder?

`<p>Generating the Back-end<br>Step 1 of 3</p>`

Now you'll use Service Builder to generate create, read, update, delete, and find operations for your application. You'll also use Service Builder to generate the necessary model, persistence, and service layers for your application. Then you can add your application's necessary business logic.

Guestbook Application Design

In the prototype application, you defined a single guestbook's entries and displayed them in a list. The full application will handle multiple Guestbooks and their entries. To make this work, you'll create two tables in the database: one for guestbooks, and one for guestbook entries.

Service Layer

This application is data-driven. It uses services for storing and retrieving data. The application asks for data, and the service fetches it. The application can then display this data to the user, who reads or modifies it. If the data is modified, the application passes it back to the service, and the service stores it. The application doesn't need to know anything about how the service does what it does.

To get started, you'll create a Service Builder project and populate its `service.xml` file with all the necessary entities to generate this code:

1. In Liferay @ide@, click *File → New → Liferay Module Project*.
2. Name the project `guestbook`.
3. Select `service-builder` for the Project Template Name.
4. Click *Next*.
5. Enter `com.liferay.docs.guestbook` for the *Package Name*.
6. Click *Finish*.

This creates two modules: an API module (`guestbook-api`) and a service module (`guestbook-service`). Next, you'll learn how to use them.

21.2 Generating Model, Service, and Persistence Layers

`<p>Generating the Back-end<br>Step 2 of 3</p>`

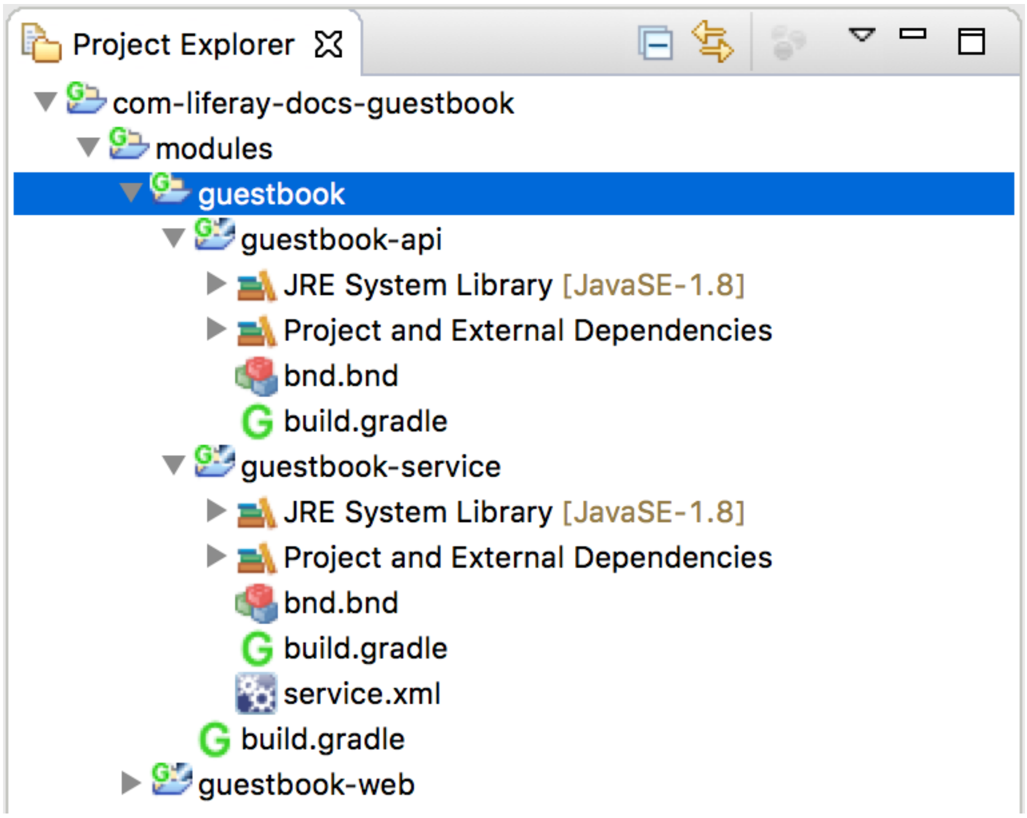

Project Explorer
com-liferay-docs-guestbook
modules
guestbook
guestbook-api
JRE System Library [JavaSE-1.8]
Project and External Dependencies
bnd.bnd
build.gradle
guestbook-service
JRE System Library [JavaSE-1.8]
Project and External Dependencies
bnd.bnd
build.gradle
service.xml
build.gradle
guestbook-web

Figure 21.2: Your current project structure.

The persistence layer saves and retrieves your model data. The service layer is a buffer between your application and persistence layers: having it lets you swap out your persistence layer for a different implementation without modifying anything but the calls in the service layer.

To model the guestbooks and entries, you'll create guestbook and entry model classes. But you won't do this directly in Java. Instead, you'll define them in Service Builder, which generates your object model and maps it to all the SQL databases Liferay DXP supports.

This application's design lets you create multiple guestbooks, each containing different sets of entries. All users with permission to access the application can add entries, but only administrative users can add guestbooks.

It's time to get started. You'll create the `Guestbook` entity first:

1. In your `guestbook-service` project, open `service.xml`.

2. When Liferay @ide@ generated your project, it filled this file with dummy entities, which you'll replace. First replace the file's opening contents (below the `DOCTYPE`) with the following code:

```
<service-builder auto-namespace-tables="true" package-path="com.liferay.docs.guestbook">
    <author>liferay</author>
    <namespace>GB</namespace>
    <entity name="Guestbook" local-service="true" uuid="true">
```

   This defines the author, namespace, and the entity name. The namespace keeps the database field names from conflicting. The last tag is the opening tag for the `Guestbook` entity definition. In this tag, you enable local services for the entity, define its name, and specify that it should have a universally unique identifier (UUID).

3. Next, replace the PK fields section:

   ```
   <column name="guestbookId" primary="true" type="long" />
   ```

   This defines `guestbookId` as the entity's primary key, of the type `long`.

4. The group instance can be left alone.

   ```
   <column name="groupId" type="long" />
   ```

   This defines the ID of the site in Liferay DXP that the entity instance belongs to (more on this in a moment).

5. Leave the Audit Fields section alone. Add status fields:

   ```
   <!-- Status fields -->

   <column name="status" type="int" />
   <column name="statusByUserId" type="long" />
   <column name="statusByUserName" type="String" />
   <column name="statusDate" type="Date" />
   ```

   The Audit section defines Liferay DXP metadata. The `companyId` is the primary key of a portal instance. The `userId` is the primary key of a user. The `createDate` and `modifiedDate` store the respective dates on which the entity instance is created and modified. The Status section is used later to implement workflow.

6. In the Other fields section, remove all the generated fields and put this one in their place:

   ```
    <column name="name" type="String" />
   ```

7. Next, remove everything else from the Guestbook entity. Before the closing `</entity>` tag, add this finder definition:

   ```
       <finder name="GroupId" return-type="Collection">
           <finder-column name="groupId" />
       </finder>
   ```

   This defines a finder that generates a `get` method you'll use to retrieve Guestbook entities. The fields used by the finder define the scope of the data retrieved. This finder gets all Guestbooks by their `groupId`, which corresponds to the site the application is on. This lets administrators put Guestbooks on multiple sites, and each `Guestbook` has its own data scoped to its site.

The `Guestbook` entity is finished for now. Next, you'll create the `Entry` entity:

1. Add the opening entity tag:

   ```
   <entity name="Entry" local-service="true" uuid="true">
   ```

As with the `Guestbook` entity, you enable local services, define the entity's name, and specify that it should have a UUID.

2. Add the tag to define the primary key and the `groupId`:

```
<column name="entryId" primary="true" type="long" />

<column name="groupId" type="long" />
```

3. Add the audit fields as you did with the `Guestbook` entity:

```
<column name="companyId" type="long" />
<column name="userId" type="long" />
<column name="userName" type="String" />
<column name="createDate" type="Date" />
<column name="modifiedDate" type="Date" />
```

4. Add status fields like you did for the guestbook:

```
 <!-- Status fields -->

 <column name="status" type="int" />
 <column name="statusByUserId" type="long" />
 <column name="statusByUserName" type="String" />
 <column name="statusDate" type="Date" />
```

5. Add the fields that define an `Entry`:

```
<column name="name" type="String" />
<column name="email" type="String" />
<column name="message" type="String" />
<column name="guestbookId" type="long" />
```

The `name`, `email`, and `message` fields comprise an `Entry`. These fields define the name of the person creating the entry, their email address, and the Guestbook message, respectively. The `guestbookId` is assigned automatically by code you'll write, and is a `Guestbook` foreign key. This ties the `Entry` to a specific `Guestbook`.

6. Add your finder and closing entity tag:

```
    <finder name="G_G" return-type="Collection">
        <finder-column name="groupId" />
        <finder-column name="guestbookId" />
    </finder>
</entity>
```

Here, you define a finder that gets guestbook entries by `groupId` and `guestbookId`. As before, the `groupId` corresponds to the site the application is on. The `guestbookId` defines the guestbook the entries come from. This finder returns a `Collection` of entries.

7. Define your exception types outside the `<entity>` tags, just before the closing `</service-builder>` tag:

```
<exceptions>
    <exception>EntryEmail</exception>
    <exception>EntryMessage</exception>
    <exception>EntryName</exception>
    <exception>GuestbookName</exception>
</exceptions>
```

These generate exception classes you'll use later in try/catch statements.

8. Save your `service.xml` file.

Now you're ready to run Service Builder to generate your model, service, and persistence layers!

1. In the Gradle Tasks pane on the right side of @ide@, open `guestbook-service` → `build`.
2. Run `buildService` by right-clicking it and selecting *Run Gradle Tasks*. Make sure you're connected to the Internet, as Gradle downloads dependencies the first time you run it.
3. In the Project Explorer, right-click the `guestbook-service` module and select *Refresh*. Repeat this step for the `guestbook-api` module. This ensures that the new classes and interfaces generated by Service Builder show up in @ide@.
4. In the Project Explorer, right-click the `guestbook-service` module and select *Gradle → Refresh Gradle Project*. Repeat this step for the `guestbook-api` module. This ensures that your modules' Gradle dependencies are up to date.

Service Builder is based on a design philosophy called loose coupling. It generates three layers of your application: the model, the service, and the persistence layers. Loose coupling means you can swap out the persistence layer with little to no change in the model and service layers. The model is in the `-api` module, and the service and persistence layers are in the `-service` module.

Each layer is implemented using Java Interfaces and implementations of those interfaces. Rather than have one `Entry` class that represents your model, Service Builder generates a system of classes that include a `Guestbook` interface, a `GuestbookBaseImpl` abstract class that Service Builder manages, and a `GuestbookImpl` class that you can customize. This design lets you customize your model, while Service Builder generates code that's tedious to write. That's why Service Builder is a code generator for code generator haters.

Next, you'll create the service implementations.

21.3 Implementing Service Methods

```
<p>Generating the Back-end<br>Step 3 of 3</p>
```

When you use Service Builder, you implement the services in the service module. Because your application's projects are components, you can reference your service layer from your web module.

You'll implement services for guestbooks and entries in the `guestbook-service` module's `GuestbookLocalServiceImpl` and `EntryLocalServiceImpl`, respectively.

Follow these steps to implement services for guestbooks in `GuestbookLocalServiceImpl`:

1. In the `com.liferay.docs.guestbook.service.impl` package, open `GuestbookLocalServiceImpl`. Then add this `addGuestbook` method:

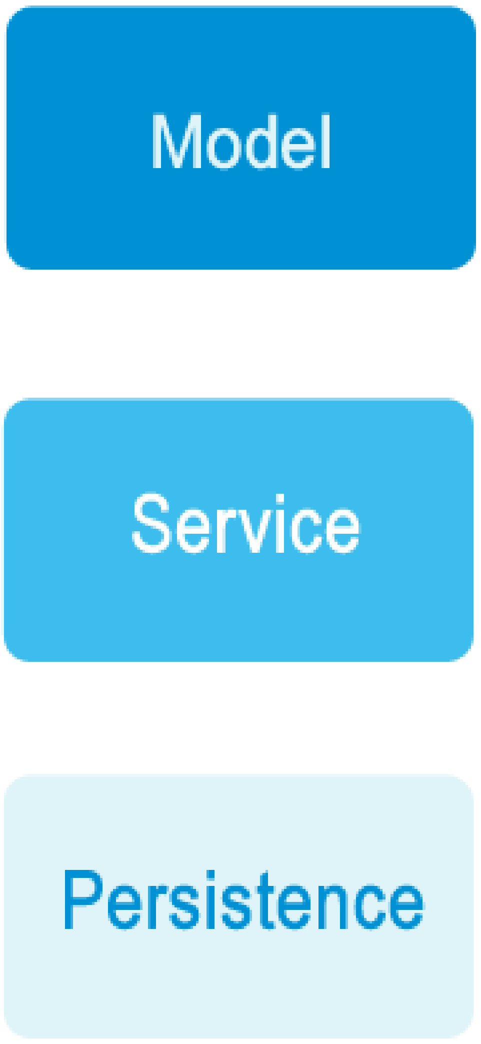


Figure 21.3: The Model, Service, and Persistence Layer.

```
public Guestbook addGuestbook(
    long userId, String name, ServiceContext serviceContext)
    throws PortalException {

    long groupId = serviceContext.getScopeGroupId();

    User user = userLocalService.getUserById(userId);

    Date now = new Date();

    validate(name);

    long guestbookId = counterLocalService.increment();

    Guestbook guestbook = guestbookPersistence.create(guestbookId);

    guestbook.setUuid(serviceContext.getUuid());
    guestbook.setUserId(userId);
    guestbook.setGroupId(groupId);
    guestbook.setCompanyId(user.getCompanyId());
    guestbook.setUserName(user.getFullName());
    guestbook.setCreateDate(serviceContext.getCreateDate(now));
    guestbook.setModifiedDate(serviceContext.getModifiedDate(now));
    guestbook.setName(name);
    guestbook.setExpandoBridgeAttributes(serviceContext);

    guestbookPersistence.update(guestbook);

    return guestbook;

}
```

This method adds a guestbook to the database. It retrieves metadata from the environment (such as the current user's ID, the group ID, etc.), along with data passed from the user. It validates this data and uses it to construct a `Guestbook` object. The method then persists this object to the database and returns the object. You only implement the business logic here because Service Builder already generated the model and all the code that maps that model to the database.

2. Add the methods for getting `Guestbook` objects:

```
public List<Guestbook> getGuestbooks(long groupId) {

    return guestbookPersistence.findByGroupId(groupId);
}

public List<Guestbook> getGuestbooks(long groupId, int start, int end,
    OrderByComparator<Guestbook> obc) {

    return guestbookPersistence.findByGroupId(groupId, start, end, obc);
}

public List<Guestbook> getGuestbooks(long groupId, int start, int end) {

    return guestbookPersistence.findByGroupId(groupId, start, end);
}

public int getGuestbooksCount(long groupId) {

    return guestbookPersistence.countByGroupId(groupId);
}
```

These call the finders you generated with Service Builder. The first method retrieves a list of guestbooks from the site specified by `groupId`. The next two methods get paginated lists, optionally in a particular order. The final method gives you the total number of guestbooks for a given site.

3. Finally, add the guestbook validator method:

```
protected void validate(String name) throws PortalException {
    if (Validator.isNull(name)) {
        throw new GuestbookNameException();
    }
}
```

   This method uses Liferay DXP's `Validator` to make sure the user entered text for the guestbook name.

4. Press [CTRL]+[SHIFT]+O to organize imports and select the following when prompted:

   - `java.util.Date`
   - `com.liferay.portal.kernel.service.ServiceContext`
   - `com.liferay.docs.guestbook.model.Entry`
   - `com.liferay.portal.kernel.util.Validator`

Now you're ready to implement services for entries in `EntryLocalServiceImpl`. Do so now by following these steps:

1. In the `com.liferay.docs.guestbook.service.impl` package, open `EntryLocalServiceImpl`. Add this `addEntry` method:

```
public Entry addEntry(
    long userId, long guestbookId, String name, String email,
    String message, ServiceContext serviceContext)
    throws PortalException {

    long groupId = serviceContext.getScopeGroupId();

    User user = userLocalService.getUserById(userId);

    Date now = new Date();

    validate(name, email, message);

    long entryId = counterLocalService.increment();

    Entry entry = entryPersistence.create(entryId);

    entry.setUuid(serviceContext.getUuid());
    entry.setUserId(userId);
    entry.setGroupId(groupId);
    entry.setCompanyId(user.getCompanyId());
    entry.setUserName(user.getFullName());
    entry.setCreateDate(serviceContext.getCreateDate(now));
    entry.setModifiedDate(serviceContext.getModifiedDate(now));
    entry.setExpandoBridgeAttributes(serviceContext);
    entry.setGuestbookId(guestbookId);
    entry.setName(name);
    entry.setEmail(email);
    entry.setMessage(message);

    entryPersistence.update(entry);

    return entry;
}
```

Like the addGuestbook method, addEntry takes data from the current context along with data the user entered, validates it, and creates a model object. That object is then persisted to the database and returned.

2. Add this updateEntry method:

```
public Entry updateEntry (
    long userId, long guestbookId, long entryId, String name, String email,
    String message, ServiceContext serviceContext)
    throws PortalException, SystemException {

    Date now = new Date();

    validate(name, email, message);

    Entry entry = getEntry(entryId);

    User user = userLocalService.getUserById(userId);

    entry.setUserId(userId);
    entry.setUserName(user.getFullName());
    entry.setModifiedDate(serviceContext.getModifiedDate(now));
    entry.setName(name);
    entry.setEmail(email);
    entry.setMessage(message);
    entry.setExpandoBridgeAttributes(serviceContext);

    entryPersistence.update(entry);

    return entry;
}
```

This method first retrieves the entry and updates its data to reflect what the user submitted, including its date modified.

3. Add this deleteEntry method:

```
public Entry deleteEntry (long entryId, ServiceContext serviceContext)
    throws PortalException {

    Entry entry = getEntry(entryId);

    entry = deleteEntry(entryId);

    return entry;
}
```

This method retrieves the entry object defined by entryId, deletes it from the database, and then returns the deleted object.

4. Add the methods for getting Entry objects:

```
public List<Entry> getEntries(long groupId, long guestbookId) {
    return entryPersistence.findByG_G(groupId, guestbookId);
}

public List<Entry> getEntries(long groupId, long guestbookId, int start, int end)
    throws SystemException {

    return entryPersistence.findByG_G(groupId, guestbookId, start, end);
}
```

```
public List<Entry> getEntries(
    long groupId, long guestbookId, int start, int end, OrderByComparator<Entry> obc) {

    return entryPersistence.findByG_G(groupId, guestbookId, start, end, obc);
}

public int getEntriesCount(long groupId, long guestbookId) {
    return entryPersistence.countByG_G(groupId, guestbookId);
}
```

These methods, like the getters in `GuestbookLocalServiceImpl`, call the finders you generated with Service Builder. These `getEntries*` methods, however, retrieve entries from a specified guestbook and site. The first method gets a list of entries. The next method gets a paginated list. The third method sorts the paginated list, and the last method gets the total number of entries as an integer.

5. Add the `validate` method:

```
protected void validate(String name, String email, String entry)
    throws PortalException {

    if (Validator.isNull(name)) {
        throw new EntryNameException();
    }

    if (!Validator.isEmailAddress(email)) {
        throw new EntryEmailException();
    }

    if (Validator.isNull(entry)) {
        throw new EntryMessageException();
    }
}
```

   This method makes sure the user entered relevant data when creating an entry.

6. Press [CTRL]+[SHIFT]+O to organize imports and select the following when prompted:

   - `java.util.Date`
   - `com.liferay.portal.kernel.service.ServiceContext`
   - `com.liferay.docs.guestbook.model.Entry`
   - `com.liferay.portal.kernel.util.Validator`

Nice work! These local service methods implement the services that are referenced in the portlet class.

Updating Generated Classes

Now that you've implemented the service methods, you must make them available to the rest of your application. To do this, run `buildService` again:

1. In *Gradle Tasks → guestbook-service → build*, right-click `buildService` and select *Run Gradle Tasks*. In the utility classes, Service Builder populates calls to your newly created service methods.

2. In the Project Explorer, right-click the `guestbook-service` module and select *Refresh*. Repeat this step for the `guestbook-api` module. This ensures that the changes made by Service Builder show up in Liferay @ide@.

3. In the Project Explorer, right-click the `guestbook-service` module and select *Gradle → Refresh Gradle Project*. Repeat this step for the `guestbook-api` module. This ensures that your modules' Gradle dependencies are up to date.

Tip: If something goes awry when working with Service Builder, repeat these steps to run Service Builder again and refresh your api and service modules.

Excellent! Your new back-end has been generated. Now it's time to refactor your prototype to use it.

Chapter 22

Refactoring the Prototype

In an earlier section of this Learning Path, you created a Guestbook portlet prototype. Then you wrote a `service.xml` file to define your application's data model, and used Service Builder to generate the back-end code (the model, service, and persistence layers). You also added service methods using the appropriate extension points: your entities' `*LocalServiceImpl` classes. Now you need to integrate the original prototype with the new back-end to create a fully functional application.

There are many differences between the prototype and the application you'll create. In the back-end, you've already accounted for one big difference: users can create multiple Guestbooks that each have their own entries. In the front-end, however, only site administrators should be able to create guestbooks. Therefore, you'll create another portlet called Guestbook Admin and place it in the Content menu for sites.

To turn this application from a prototype into a full-fledged Liferay web application, you'll make these changes:

- Modify your view layer's folder structure to account for the administrative portlet
- Set the Display Category so users can find the application more easily
- Create a file to store the application's text keys
- Change the controller to call your new Service Builder-based back-end
- Update the view so it can display multiple Guestbooks in tabs

Ready to begin?
Let's Go!

22.1 Organizing Folders for Larger Applications

`<p>Refactoring the Prototype<br>Step 1 of 6</p>`

Currently, all your JSPs sit in your web module's `src/main/resources/META-INF/resources` folder, which serves as the context root folder. To make a clear separation between the Guestbook portlet and the Guestbook Admin portlet, you must place the files that make up their view layers in separate folders:

1. In the `guestbook-web` project, right click the `src/main/resources/META-INF/resources` folder and select *New → Folder*. Name the new folder `guestbookwebportlet` and click *Finish*.

2. Copy `view.jsp` and `edit_entry.jsp` into the new folder by dragging and dropping them there.

3. Open both files and change the `init.jsp` location at the top of the file:

   ```
   <%@include file="../init.jsp"%>
   ```

4. Check the other references to JSPs within the files to make sure that they point to the new locations.

As you update your view layer to take full advantage of the new back-end, you'll update any references to the old paths. In addition, you must update the resource location in your component properties. In the next step, you'll update all of those properties, including the one that defines the resource location.

22.2 Defining the Component Metadata Properties

<p>Refactoring the Prototype<br>Step 2 of 6</p>

When users add applications to a page, they pick them from a list of *display categories*.

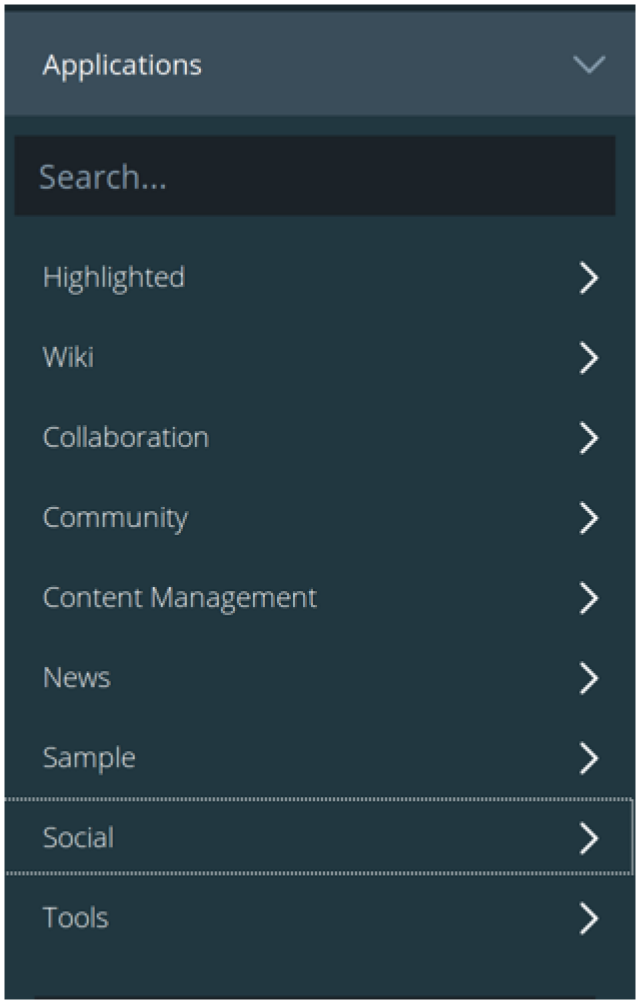


Figure 22.1: Users choose applications from a list of display categories.

A portlet's display category is defined in its component class as a metadata property. Since the Guestbook portlet lets users communicate with each other, you'll add it to the Social category. Only one Guestbook portlet should be added to a page, so you'll also define it as a *non-instanceable* portlet. Such a portlet can appear only once on a page or site, depending on its scope.

Open the `GuestbookPortlet` class and update the component class metadata properties to match this configuration:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.social",
        "com.liferay.portlet.instanceable=false",
        "com.liferay.portlet.scopeable=true",
        "javax.portlet.display-name=Guestbook",
        "javax.portlet.expiration-cache=0",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/guestbookwebportlet/view.jsp",
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user",
        "javax.portlet.supports.mime-type=text/html"
    },
    service = Portlet.class
)
```

The `com.liferay.portlet.display-category=category.social` property sets the Guestbook portlet's display category to *Social*. The `com.liferay.portlet.instanceable=false` property specifies that the Guestbook portlet is non-instanceable, so only one instance of the portlet can be added to a page. In the property `javax.portlet.init-param.view-template`, you also update the location of the main `view.jsp` to its new location in `/guestbookwebportlet`.

Since you edited the portlet's metadata, you must remove and re-add the portlet to the page before continuing:

1. Go to `localhost:8080` in your web browser.

2. Sign in to your administrative account.

3. The Guestbook portlet now shows an error on the page. Click its portlet menu (at the top-right of the portlet), then select *Remove* and click *OK* to confirm.

4. Open the *Add* menu and select *Applications*.

5. Open the *Social* category and drag and drop the *Guestbook* application onto the page.

Great! Now the Guestbook portlet appears in an appropriate category. Though you were able to add it to the page before, the user experience is better.

22.3 Creating Portlet Keys

```
<p>Refactoring the Prototype<br>Step 3 of 6</p>
```

`PortletKeys` let you manage important things like the portlet name or other repeatable, commonly used variables in one place. This way, if you need to change the portlet's name, you can do it in one place, and then reference it in every class that needs it. Keys must be referenced first as a component property, and then as a class.

Follow these steps to create your application's `PortletKeys`:

1. In your guestbook-web module, open the GuestbookPortlet class and update the component class metadata properties by adding one new property:

   ```
   "javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK,
   ```

   Note that you need the trailing comma if you've added the property to the middle of the list. If you've added it to the end of the last, leave it off.

2. Save GuestbookPortlet. It now shows an error because you haven't added the key to the class.

3. Open the com.liferay.docs.guestbook.constants package.

4. Open GuestbookPortletKeys and create a public, static, final String called GUESTBOOK with a value of com_liferay_docs_guestbook_portlet_GuestbookPortlet:

   ```
   public static final String GUESTBOOK =
           "com_liferay_docs_guestbook_portlet_GuestbookPortlet";
   ```

5. Save the file.

Now GuestbookPortlet's error has disappeared, and your application can be deployed again. Nice job! Next, you'll integrate your application with the new back-end you generated with Service Builder.

22.4 Integrating the New Back-end

```
<p>Refactoring the Prototype<br>Step 4 of 6</p>
```

It's a good practice to start with a working prototype as a proof of concept, but eventually that prototype must transform into a real application. Up to this point, you've made all the preparations to do that, and now it's time to replace the prototype back-end with the real, database-driven back-end you created with Service Builder.

For the prototype, you manually created the application's model. The first thing you want to do is remove it, because Service Builder generated a new one:

1. Find the com.liferay.docs.guestbook.model package in the guestbook-web module.

2. Delete it. You'll see errors in your project, but that's because you haven't replaced the model yet.

Now you get to do some dependency management. For the web module to access the generated services, you must make it aware of the API and service modules. Then you can update the addEntry method in GuestbookPortlet to use the new services:

1. First, open guestbook-web's build.gradle file and add these dependencies:

   ```
   compileOnly project(":modules:guestbook:guestbook-api")
   compileOnly project(":modules:guestbook:guestbook-service")
   ```

2. Right-click on the guestbook-web project and select *Gradle → Refresh Gradle Project*.

3. Now you must add *references* to the Service Builder services you need. To do this, add them as class variables with `@Reference` annotations on their setter methods. Open `GuestbookPortlet` and add these references to the bottom of the file:

```
    @Reference(unbind = "-")
    protected void setEntryService(EntryLocalService entryLocalService) {
        _entryLocalService = entryLocalService;
    }

    @Reference(unbind = "-")
    protected void setGuestbookService(GuestbookLocalService guestbookLocalService) {
        _guestbookLocalService = guestbookLocalService;
    }

    private EntryLocalService _entryLocalService;
    private GuestbookLocalService _guestbookLocalService;
```

   Note that it's Liferay's code style to add class variables this way. The `@Reference` annotation on the setters allows Liferay's OSGi container to inject references to your generated services so you can use them. The `unbind` parameter tells the container there's no method for unbinding these services: the references can die with the class during garbage collection when they're no longer needed.

4. Now you can modify the `addEntry` method to use these service references:

```
public void addEntry(ActionRequest request, ActionResponse response)
            throws PortalException {

        ServiceContext serviceContext = ServiceContextFactory.getInstance(
            Entry.class.getName(), request);

        String userName = ParamUtil.getString(request, "name");
        String email = ParamUtil.getString(request, "email");
        String message = ParamUtil.getString(request, "message");
        long guestbookId = ParamUtil.getLong(request, "guestbookId");
        long entryId = ParamUtil.getLong(request, "entryId");

    if (entryId > 0) {

        try {

            _entryLocalService.updateEntry(
                serviceContext.getUserId(), guestbookId, entryId, userName,
                email, message, serviceContext);

            SessionMessages.add(request, "entryAdded");

            response.setRenderParameter(
                "guestbookId", Long.toString(guestbookId));

        }
        catch (Exception e) {
            System.out.println(e);

            SessionErrors.add(request, e.getClass().getName());

            PortalUtil.copyRequestParameters(request, response);

            response.setRenderParameter(
                "mvcPath", "/guestbookwebportlet/edit_entry.jsp");
        }

    }
    else {
        try {
            _entryLocalService.addEntry(
                serviceContext.getUserId(), guestbookId, userName, email,
                message, serviceContext);

            SessionMessages.add(request, "entryAdded");

            response.setRenderParameter(
                "guestbookId", Long.toString(guestbookId));

        }
        catch (Exception e) {
            SessionErrors.add(request, e.getClass().getName());

            PortalUtil.copyRequestParameters(request, response);

            response.setRenderParameter(
                "mvcPath", "/guestbookwebportlet/edit_entry.jsp");
        }
    }
}
```

This `addEntry` method gets the name, message, and email fields that the user submits in the JSP and passes them to the service to be stored as entry data. The `if-else` logic checks whether there's an existing `entryId`. If there is, the update service method is called, and if not, the add service method is called. In both cases, it sets a render parameter with the Guestbook ID so the application can display the guestbook's entries after this one has been added. This is all done in `try`...`catch` statements.

5. Now add `deleteEntry`, which you didn't have before:

```
public void deleteEntry(ActionRequest request, ActionResponse response) throws PortalException {
        long entryId = ParamUtil.getLong(request, "entryId");
        long guestbookId = ParamUtil.getLong(request, "guestbookId");

        ServiceContext serviceContext = ServiceContextFactory.getInstance(
            Entry.class.getName(), request);

        try {

            response.setRenderParameter(
                "guestbookId", Long.toString(guestbookId));

            _entryLocalService.deleteEntry(entryId, serviceContext);
        }

        catch (Exception e) {
            Logger.getLogger(GuestbookPortlet.class.getName()).log(
                Level.SEVERE, null, e);
        }
}
```

This method retrieves the entry object (using its ID from the request) and calls the service to delete it.

6. Next you must replace the `render` method:

```
@Override
public void render(RenderRequest renderRequest, RenderResponse renderResponse)
        throws IOException, PortletException {

        try {
            ServiceContext serviceContext = ServiceContextFactory.getInstance(
                Guestbook.class.getName(), renderRequest);

            long groupId = serviceContext.getScopeGroupId();

            long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");

            List<Guestbook> guestbooks = _guestbookLocalService.getGuestbooks(
                groupId);

            if (guestbooks.isEmpty()) {
                Guestbook guestbook = _guestbookLocalService.addGuestbook(
                    serviceContext.getUserId(), "Main", serviceContext);

                guestbookId = guestbook.getGuestbookId();
            }

            if (guestbookId == 0) {
                guestbookId = guestbooks.get(0).getGuestbookId();
            }

            renderRequest.setAttribute("guestbookId", guestbookId);
        }
        catch (Exception e) {
            throw new PortletException(e);
        }

        super.render(renderRequest, renderResponse);
}
```

This new render method checks for any guestbooks in the current site. If there aren't any, it creates one. Either way, it grabs the first guestbook so its entries can be displayed by your view layer.

7. Remove the parseEntries method. It's a remnant of the prototype application.

8. Hit Ctrl-Shift-O to organize your imports.

Awesome! You've updated your controller to use services. Next, you'll tackle the view.

22.5 Updating the View

```
<p>Refactoring the Prototype<br>Step 5 of 6</p>
```

You updated more than just the basic mechanism behind creating the entry–you completely changed its method and structure. You must, therefore, update the UI as well. To do that, you must create a new JSP for managing guestbooks, and update the existing JSPs.

1. First, you must update your dependencies. In your `guestbook-web` module, open `init.jsp` from `/src/main/resources/META-INF/resources/`. In this file, add the following additional dependencies:

```
<%@ taglib uri="http://liferay.com/tld/frontend" prefix="liferay-frontend" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://liferay.com/tld/security" prefix="liferay-security" %>
<%@ page import="java.util.List" %>
<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>
<%@ page import="com.liferay.portal.kernel.util.HtmlUtil" %>
<%@ page import="com.liferay.portal.kernel.util.StringPool" %>
<%@ page import="com.liferay.portal.kernel.model.PersistedModel" %>
<%@ page import="com.liferay.portal.kernel.dao.search.SearchEntry" %>
<%@ page import="com.liferay.portal.kernel.dao.search.ResultRow" %>
<%@ page import="com.liferay.docs.guestbook.model.Guestbook" %>
<%@ page import="com.liferay.docs.guestbook.service.EntryLocalServiceUtil" %>
<%@ page import="com.liferay.docs.guestbook.service.GuestbookLocalServiceUtil" %>
<%@ page import="com.liferay.docs.guestbook.model.Entry" %>
```

2. Open the `view.jsp` file found in `/resources/META-INF/resources/guestbookwebportlet`. Replace this file's contents with the following code:

```
<%@include file="../init.jsp"%>

<%
long guestbookId = Long.valueOf((Long) renderRequest
        .getAttribute("guestbookId"));
%>

<aui:button-row cssClass="guestbook-buttons">

    <portlet:renderURL var="addEntryURL">
        <portlet:param name="mvcPath" value="/guestbookwebportlet/edit_entry.jsp" />
        <portlet:param name="guestbookId"
            value="<%=String.valueOf(guestbookId)%>" />
    </portlet:renderURL>

    <aui:button onClick="<%=addEntryURL.toString()%>" value="Add Entry"></aui:button>

</aui:button-row>

<liferay-ui:search-container total="<%=EntryLocalServiceUtil.getEntriesCount()%>">
<liferay-ui:search-container-results
    results="<%=EntryLocalServiceUtil.getEntries(scopeGroupId.longValue(),
                    guestbookId, searchContainer.getStart(),
                    searchContainer.getEnd())%>" />

<liferay-ui:search-container-row
    className="com.liferay.docs.guestbook.model.Entry" modelVar="entry">

    <liferay-ui:search-container-column-text property="message" />

    <liferay-ui:search-container-column-text property="name" />

</liferay-ui:search-container-row>

<liferay-ui:search-iterator />

</liferay-ui:search-container>
```

This `view.jsp` now retrieves the entries from the guestbook it gets from the `render` method. It does this inside a Liferay DXP construct called a *Search Container*. This is a front-end component that makes it easy to display data in rows and columns. The `EntryLocalServiceUtil` call retrieves the data from your new Service Builder-based back-end. Otherwise, this JSP is much the same: you still have an *Add Entry* button with its corresponding URL.

Next, you need to edit the `edit_entry.jsp`:

1. Open `edit_entry.jsp` and replace the existing code with this:

```
<%@include file="../init.jsp" %>

<%

long entryId = ParamUtil.getLong(renderRequest, "entryId");

Entry entry = null;
```

```
if (entryId > 0) {
  entry = EntryLocalServiceUtil.getEntry(entryId);
}

long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");

%>

<portlet:renderURL var="viewURL">

<portlet:param name="mvcPath" value="/guestbookwebportlet/view.jsp"></portlet:param>

</portlet:renderURL>

<portlet:actionURL name="addEntry" var="addEntryURL"></portlet:actionURL>

<aui:form action="<%= addEntryURL %>" name="<portlet:namespace />fm">

<aui:model-context bean="<%= entry %>" model="<%= Entry.class %>" />

    <aui:fieldset>

        <aui:input name="name" />
        <aui:input name="email" />
        <aui:input name="message" />
        <aui:input name="entryId" type="hidden" />
        <aui:input name="guestbookId" type="hidden" value='<%= entry == null ? guestbookId : entry.getGuestbookId() %>'/>

    </aui:fieldset>

    <aui:button-row>

        <aui:button type="submit"></aui:button>
        <aui:button type="cancel" onClick="<%= viewURL.toString() %>"></aui:button>

    </aui:button-row>
</aui:form>
```

This is much the same form, though there are more fields now. Using some AlloyUI tags, the form is linked to your `Entry` entity. The two hidden fields contain the new `entryId` and the `guestbookId` for the guestbook the new entry belongs to. The submit button is an `ActionURL` that executes the `addEntry` method in the controller (your portlet class).

Congratulations! You've now successfully replaced your prototype back-end with a real, database-driven back-end. Next, you'll do a quick review and deploy your application.

22.6 Fitting it All Together

```
<p>Refactoring the Prototype<br>Step 6 of 6</p>
```

You've created a complete data-driven application from the back-end to the display. It's a great time to review how everything connects together.

The Entry

First, you defined your model in Service Builder's configuration file, `service.xml`. The main part of this is your `Entry` object:

```
<entity local-service="true" name="Entry" uuid="true">

    <!-- PK fields -->

    <column name="entryId" primary="true" type="long" />

    <!-- Group instance -->

    <column name="groupId" type="long" />

    <!-- Audit fields -->

    <column name="companyId" type="long" />
    <column name="userId" type="long" />
    <column name="userName" type="String" />
    <column name="createDate" type="Date" />
    <column name="modifiedDate" type="Date" />
    <column name="name" type="String" />
    <column name="email" type="String" />
    <column name="message" type="String" />
    <column name="guestbookId" type="long" />

    <finder name="G_G" return-type="Collection">
        <finder-column name="groupId" />
        <finder-column name="guestbookId" />
    </finder>
</entity>
```

Next, you created a service implementation in `EntryLocalServiceImpl` that defined how to get and store the entry. Every field you defined was accounted for in the `addEntry` method.

```
public Entry addEntry(long userId, long guestbookId, String name, String email,
        String message, ServiceContext serviceContext)
        throws PortalException {

        long groupId = serviceContext.getScopeGroupId();

        User user = userLocalService.getUserById(userId);

        Date now = new Date();

        validate(name, email, message);

        long entryId = counterLocalService.increment();

        Entry entry = entryPersistence.create(entryId);

        entry.setUuid(serviceContext.getUuid());
        entry.setUserId(userId);
        entry.setGroupId(groupId);
        entry.setCompanyId(user.getCompanyId());
        entry.setUserName(user.getFullName());
        entry.setCreateDate(serviceContext.getCreateDate(now));
        entry.setModifiedDate(serviceContext.getModifiedDate(now));
        entry.setExpandoBridgeAttributes(serviceContext);
        entry.setGuestbookId(guestbookId);
        entry.setName(name);
        entry.setEmail(email);
        entry.setMessage(message);

        entryPersistence.update(entry);

        return entry;
}
```

Notice that all the fields you described in Service Builder (including things like the `uuid`) are present here.

You also added ways to get entries:

```
public List<Entry> getEntries(long groupId, long guestbookId) {
    return entryPersistence.findByG_G(groupId, guestbookId);
}

public List<Entry> getEntries(
    long groupId, long guestbookId, int start, int end, OrderByComparator<Entry> obc) {

    return entryPersistence.findByG_G(groupId, guestbookId, start, end, obc);
}

public List<Entry> getEntries(long groupId, long guestbookId, int start, int end)
    throws SystemException {

    return entryPersistence.findByG_G(groupId, guestbookId, start, end);
}
```

In `service.xml` you defined `groupId` and `guestbookId` as the two finder fields, and in these methods you called methods generated to the persistence layer.

After you implemented all that, Service Builder propagated your implementation to the interfaces, so they could be called. Then, in the portlet class, you created references to the service classes that Service Builder generated, and used those references to access the service to add an entry:

```
_entryLocalService.addEntry( serviceContext.getUserId(), guestbookId,
    userName, email,message, serviceContext);
```

Finally, you wrapped all this up in a user interface that lets users enter the information they want, and displays the data they've entered.

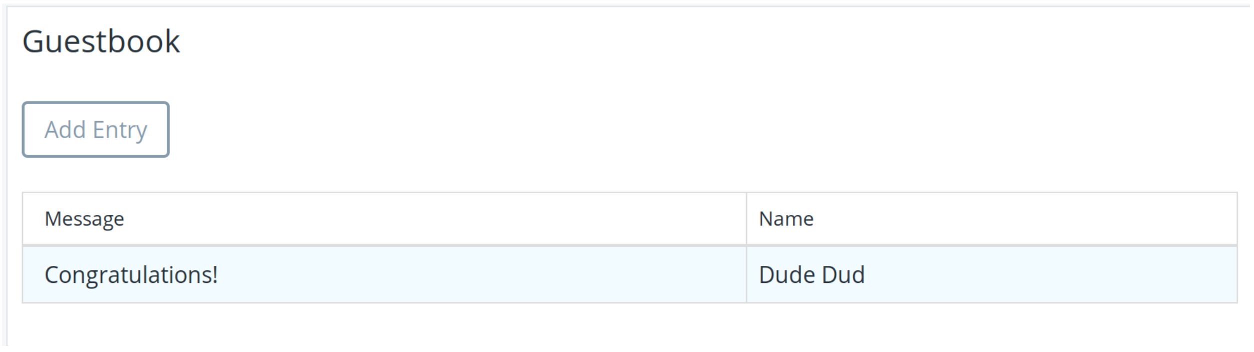
Now that you've built the application, and you can see a clear picture of how it all works, it's time to test it.

Deploying and Testing the Application

1. Drag and drop the `guestbook-api` module onto the server.
2. Drag and drop the `guestbook-service` module onto the server.
3. Look for the STARTED messages from the console.
4. Go to your Liferay DXP instance at `localhost:8080` in your browser to test your updated application.
5. Click *Add Entry*.
6. Enter a *Name*, *Message*, and *Email Address*.
7. Click *Submit*.
8. Verify that your entry appears.

What's Next?

You've created a working web application and deployed it on Liferay DXP. If you've created web applications before, though, you know that it's missing some important features: security, front-end validation, and an interface for administrators to create multiple guestbooks per portlet. In the next section, you'll begin adding these (and more) features.



Guestbook

Message	Name
Congratulations!	Dude Dud

Figure 22.2: A new guestbook and entry.

Chapter 23

Writing an Administrative Portlet

Like the prototype, the real application lets users add and view guestbook entries. The application's back-end, however, is much more powerful. It can support many guestbooks and their associated entries. Despite this, there's no UI to support these added features. When you create this UI, you must also make sure that only administrators can add guestbooks.

To accomplish this, you must create a Guestbook Admin portlet and place it in Liferay DXP's administrative interface–specifically, within the Content menu. This way, the Guestbook Admin portlet is accessible only to Site Administrators, and users can use the Guestbook portlet to create entries.

In short, this is a simple application with a simple interface:

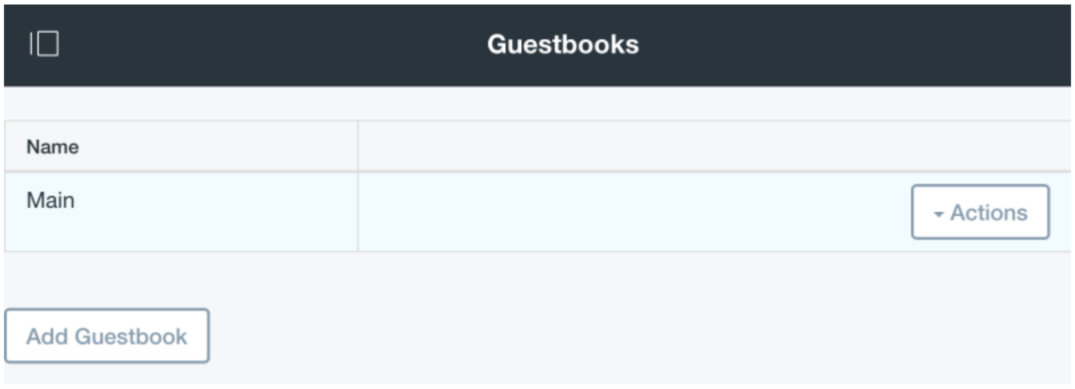


Figure 23.1: The Guestbook Admin portlet lets administrators manage Guestbooks.

Are you ready to begin?

Let's Go!

23.1 Creating the Classes

```
<p>Writing the Guestbook Admin App<br>Step 1 of 5</p>
```

Because the Guestbook and Guestbook Admin applications should be bundled together, you'll create the new application manually inside the `guestbook-web` project, rather than by using a wizard. If you disagree with this design decision, you can create a separate project for Guestbook Admin; the project template you'd use is *panel-app*. For now, however, it's better to go through the process manually to learn how it all works:

1. Right-click the `com.liferay.docs.guestbook.portlet` package in the `guestbook-web` project and select *New → Class*.

2. Name the class `GuestbookAdminPortlet`.

3. Click *Browse* next to the Superclass and search for `MVCPortlet`. Click it and select *OK*.

4. Click *Finish*.

You now have your Guestbook Admin application's portlet class. For an administrative application, however, you need at least one more component.

Panels and Categories

As described in the product menu tutorial, there are three sections of the product menu as illustrated below.

Each section is called a *panel category*. A panel category can hold various menu items called *panel apps*. In the illustration above, the Sites menu is open to reveal its panel apps and categories (yes, you can nest them).

The most natural place for the Guestbook Admin portlet is in the *Content* panel category with Liferay DXP's other content-based apps. This integrates it nicely in the spot where site administrators expect it to be. This also means you don't have to create a new category for it: you can just create the panel entry, which is what you'll do next. If you'd like to learn more about panel categories and apps after this, see the product menu tutorial and the control menu tutorial.

Follow these steps to create the panel entry for the Guestbook Admin portlet:

1. Add the dependency you need to extend Liferay DXP's panel categories and apps. To do this, open `guestbook-web`'s `build.gradle` file and add this dependency:

   ```
   compileOnly group: "com.liferay", name: "com.liferay.application.list.api", version: "2.0.0"
   ```

2. Right-click `guestbook-web` and select *Gradle → Refresh Gradle Project*.

3. Right-click `src/main/java` in the `guestbook-web` project and select *New → Package*. Name the package `com.liferay.docs.guestbook.application.list` and click *Finish*.

4. Right-click your new package and select *New → Class*. Name the class `GuestbookAdminPanelApp`. Click *Browse* next to Superclass, search for `BasePanelApp`, select it, and click *OK*. Then click *Finish*.

Great! You've created the classes you need, and you're ready to begin working on them.

23.2 Adding Metadata

```
<p>Writing the Guestbook Admin App<br>Step 2 of 5</p>
```

Now that you've generated the classes, you must turn them into OSGi components. Remember that because components are container-managed objects, you must provide metadata that tells Liferay DXP's OSGi container how to manage their lifecycles.

Follow these steps:

1. Add the following portlet key to the `GuestbookPortletKeys` class:

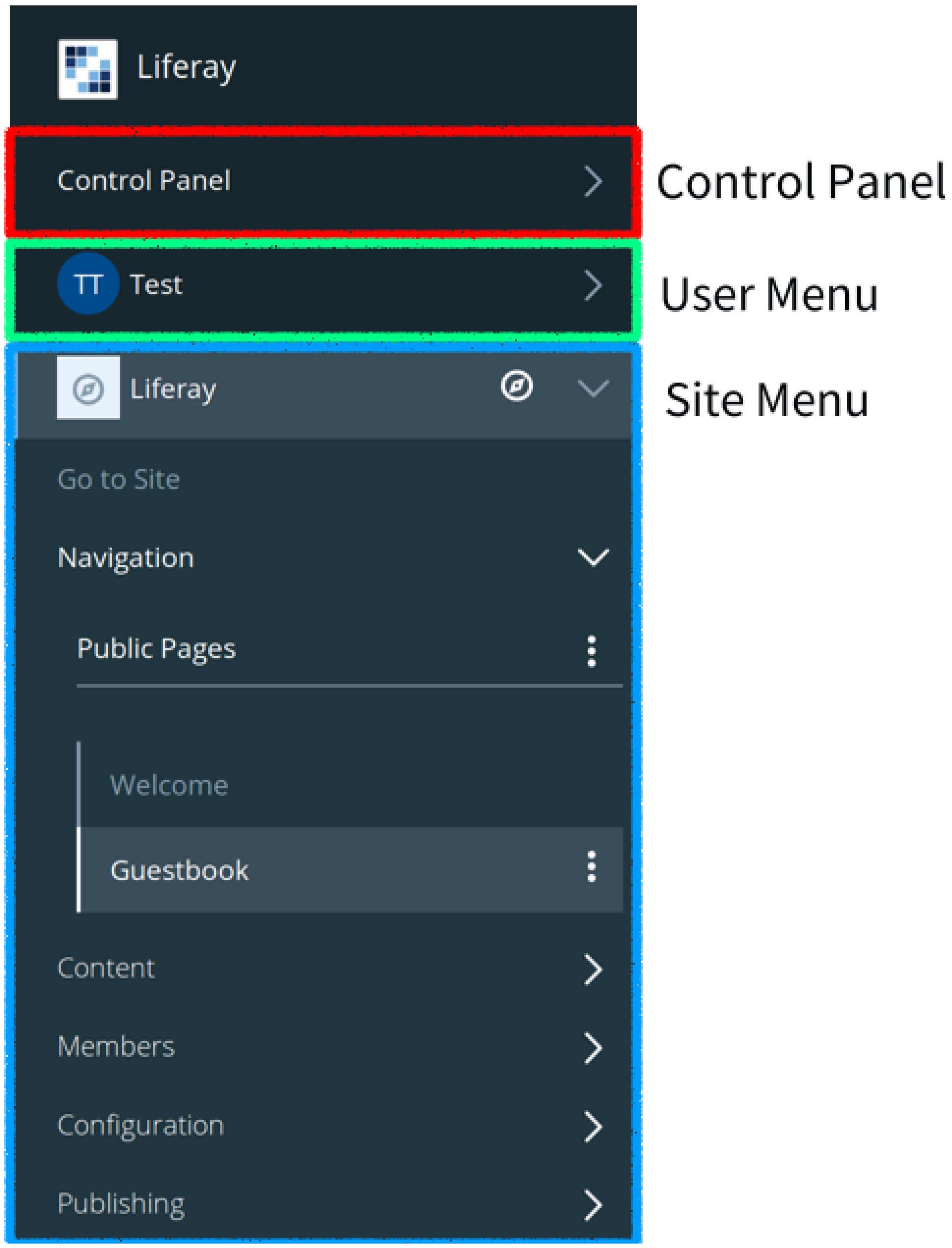


Figure 23.2: The product menu is split into three sections: the Control Panel, the User menu, and the Sites menu.

```
public static final String GUESTBOOK_ADMIN =
  "com_liferay_docs_guestbook_portlet_GuestbookAdminPortlet";
```

2. Open the `GuestbookAdminPortlet` class and add the `@Component` annotation immediately above the class declaration:

   ```
   @Component(
       immediate = true,
       property = {
               "com.liferay.portlet.display-category=category.hidden",
               "com.liferay.portlet.scopeable=true",
               "javax.portlet.display-name=Guestbooks",
               "javax.portlet.expiration-cache=0",
               "javax.portlet.init-param.portlet-title-based-navigation=true",
               "javax.portlet.init-param.template-path=/",
               "javax.portlet.init-param.view-template=/guestbookadminportlet/view.jsp",
               "javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK_ADMIN,
               "javax.portlet.resource-bundle=content.Language",
               "javax.portlet.security-role-ref=administrator",
               "javax.portlet.supports.mime-type=text/html",
               "com.liferay.portlet.add-default-resource=true"
       },
       service = Portlet.class
   )
   ```

3. Hit [CTRL]+[SHIFT]+O to add the `javax.portlet.Portlet` and other imports.

There are only a few new things here. Note the value of the `javax.portlet.display-name` property: `Guestbooks`. This is the name that appears in the Site menu. Also note the value of the `javax.portlet.name` property: `+ GuestbookPortletKeys.GUESTBOOK_ADMIN`. This specifies the portlet's title via the `GUESTBOOK_ADMIN` portlet key that you just created.

Pay special attention to the following metadata property:

```
com.liferay.portlet.display-category=category.hidden
```

This is the same property you used before with the Guestbook portlet. You placed that portlet in the Social category. The value `category.hidden` specifies a special category that doesn't appear anywhere. You're putting the Guestbook Admin portlet here because it'll be part of the Site menu, and you don't want users adding it to a page. This prevents them from doing that.

Next, you can configure the Panel app class. Follow these steps:

1. Open the `GuestbookAdminPanelApp` class and add the `@Component` annotation immediately above the class declaration:

   ```
   @Component(
       immediate = true,
       property = {
           "panel.app.order:Integer=300",
           "panel.category.key=" + PanelCategoryKeys.SITE_ADMINISTRATION_CONTENT
       },
       service = PanelApp.class
   )
   ```

   The `panel.category.key` metadata property determines where to place the Guestbook Admin portlet in the Product Menu. Remember that the Product Menu is divided into three main sections: the Control Panel, the User Menu, and the Site Administration area. The value of the `panel.category.key` property is PanelCategoryKeys.SITE_ADMINISTRATION_CONTENT, which means Guestbook Admin is in *Site Administration → Content*. The key is provided by the PanelCategoryKeys class. The panel.app.order value determines the rank for the Guestbook Admin portlet in the list.

2. Finally, update the class to use the proper name and portlet keys:

```
public class GuestbookAdminPanelApp extends BasePanelApp {

    @Override
    public String getPortletId() {
        return GuestbookPortletKeys.GUESTBOOK_ADMIN;
    }

    @Override
    @Reference(
        target = "(javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK_ADMIN + ")",
        unbind = "-"
    )
    public void setPortlet(Portlet portlet) {
        super.setPortlet(portlet);
    }

}
```

3. Hit [CTRL]+[SHIFT]+O to organize imports. This time, import com.liferay.portal.kernel.model.Portlet instead of javax.portlet.Portlet.

Now that the configuration is out of the way, you're free to implement the app's functionality: adding, editing, and deleting guestbooks. That's the next step.

23.3 Updating Your Service Layer

<p>Writing the Guestbook Admin App<br>Step 3 of 5</p>

In an earlier section, you wrote an addGuestbook service method in GuestbookLocalServiceImpl, but you never used it. To have full functionality over guestbooks, you must also add methods for updating and deleting guestbooks, as well as for returning the number of guestbooks in a site.

Adding Guestbook Service Methods

Remember that when working with Service Builder, you define your service in the *Impl classes. After you add or remove a method from an *Impl class, or change the signature of a method in an *Impl class, you must run Service Builder. Service Builder updates the affected interfaces and any other generated code.

Follow these steps to add the required guestbook service methods:

1. Go to the guestbook-service project and open GuestbookLocalServiceImpl.java in the com.liferay.docs.guestbook.serv package. Add the following method for updating a guestbook:

```
public Guestbook updateGuestbook(long userId, long guestbookId,
    String name, ServiceContext serviceContext) throws PortalException,
                SystemException {

        Date now = new Date();

        validate(name);
        Guestbook guestbook = getGuestbook(guestbookId);

        User user = userLocalService.getUser(userId);

        guestbook.setUserId(userId);
        guestbook.setUserName(user.getFullName());
        guestbook.setModifiedDate(serviceContext.getModifiedDate(now));
        guestbook.setName(name);
        guestbook.setExpandoBridgeAttributes(serviceContext);

        guestbookPersistence.update(guestbook);

        return guestbook;
}
```

The `updateGuestbook` method retrieves the `Guestbook` by its ID, replaces its data with what the user entered, and then calls the persistence layer to save it back to the database.

2. Next, add the following method for deleting a guestbook:

```
public Guestbook deleteGuestbook(long guestbookId,
                ServiceContext serviceContext) throws PortalException,
                SystemException {

        Guestbook guestbook = getGuestbook(guestbookId);

        List<Entry> entries = entryLocalService.getEntries(
                        serviceContext.getScopeGroupId(), guestbookId);

        for (Entry entry : entries) {
                entryLocalService.deleteEntry(entry.getEntryId());
        }

        guestbook = deleteGuestbook(guestbook);

        return guestbook;
}
```

It's important to consider what should happen if you delete a guestbook that has existing entries. If you just deleted the guestbook, the guestbook's entries would still exist in the database, but they'd be orphaned. Your `deleteGuestbook` service method makes a service call to delete a guestbook's entries before deleting that guestbook. This way, guestbook entries are never orphaned.

3. Use [CTRL]+[SHIFT]+O to update your imports, then save `GuestbookLocalServiceImpl.java`.

4. In the Gradle Tasks pane on the right side in Liferay @ide@, run Service Builder by opening the `guestbook-service` module and double-clicking `buildService`.

Now that you've finished updating the service layer, it's time to work on the Guestbook Admin portlet itself.

23.4 Defining Portlet Actions

```
<p>Writing the Guestbook Admin App<br>Step 4 of 5</p>
```

The Guestbook Admin portlet now needs action methods for adding, updating, and deleting guestbooks. As with the Guestbook portlet, action methods call the corresponding service methods. Note that since your services and applications are all running in the same container, any application can call the Guestbook services. This is an advantage of Liferay DXP's OSGi-based architecture: different applications or modules can call services published by other modules. If a service is published, it can be used via `@Reference`. You'll take advantage of this here in the Guestbook Admin portlet to consume one of the same services consumed by the Guestbook portlet (the `addGuestbook` service).

Adding Three Portlet Actions

The Guestbook Admin portlet must let administrators add, update, and delete `Guestbook` objects. You'll create portlet actions to meet these requirements. Open `GuestbookAdminPortlet.java` and follow these steps:

1. Add the following action method and instance variables needed for adding a new guestbook:

```
public void addGuestbook(ActionRequest request, ActionResponse response)
    throws PortalException {

    ServiceContext serviceContext = ServiceContextFactory.getInstance(
        Guestbook.class.getName(), request);

    String name = ParamUtil.getString(request, "name");

    try {
        _guestbookLocalService.addGuestbook(
            serviceContext.getUserId(), name, serviceContext);
    }
    catch (PortalException pe) {

        Logger.getLogger(GuestbookAdminPortlet.class.getName()).log(
            Level.SEVERE, null, pe);

        response.setRenderParameter(
            "mvcPath", "/guestbookadminportlet/edit_guestbook.jsp");
    }
}

private GuestbookLocalService _guestbookLocalService;

@Reference(unbind = "-")
protected void setGuestbookService(GuestbookLocalService guestbookLocalService) {
    _guestbookLocalService = guestbookLocalService;
}
```

   Since `addGuestbook` is a portlet action method, it takes `ActionRequest` and `ActionResponse` parameters. To make the service call to add a new guestbook, the guestbook's name must be retrieved from the request. The `serviceContext` must also be retrieved from the request and passed as an argument in the service call. If an exception is thrown, you should display the Add Guestbook form and not the default view. That's why you add this line in the `catch` block:

```
response.setRenderParameter("mvcPath",
        "/guestbookadminportlet/edit_guestbook.jsp");
```

   Later, you'll use this for field validation and to show error messages to the user. Note that `/guestbookadminportlet/edit_guestbook.jsp` doesn't exist yet; you'll create it in the next section when you're designing the Guestbook Admin portlet's user interface.

2. Add the following action method for updating an existing guestbook:

```
public void updateGuestbook(ActionRequest request, ActionResponse response)
    throws PortalException {

    ServiceContext serviceContext = ServiceContextFactory.getInstance(
        Guestbook.class.getName(), request);

    String name = ParamUtil.getString(request, "name");
    long guestbookId = ParamUtil.getLong(request, "guestbookId");

    try {
        _guestbookLocalService.updateGuestbook(
            serviceContext.getUserId(), guestbookId, name, serviceContext);

    } catch (PortalException pe) {

        Logger.getLogger(GuestbookAdminPortlet.class.getName()).log(
            Level.SEVERE, null, pe);

        response.setRenderParameter(
            "mvcPath", "/guestbookadminportlet/edit_guestbook.jsp");
    }
}
```

This method retrieves the guestbook name, ID, and the `serviceContext` from the request. The `updateGuestbook` service call uses the guestbook's ID to identify the guestbook to update. If there's a problem with the service call, the Guestbook Admin portlet displays the Edit Guestbook form again so that the user can edit the form and resubmit:

```
response.setRenderParameter("mvcPath",
        "/guestbookadminportlet/edit_guestbook.jsp");
```

Note that the Edit Guestbook form uses the same JSP as the Add Guestbook form to avoid duplication of code.

3. Add the following action method for deleting a guestbook:

```
public void deleteGuestbook(ActionRequest request, ActionResponse response)
    throws PortalException {

    ServiceContext serviceContext = ServiceContextFactory.getInstance(
        Guestbook.class.getName(), request);

    long guestbookId = ParamUtil.getLong(request, "guestbookId");

    try {
        _guestbookLocalService.deleteGuestbook(guestbookId, serviceContext);
    }
    catch (PortalException pe) {

        Logger.getLogger(GuestbookAdminPortlet.class.getName()).log(
            Level.SEVERE, null, pe);
    }
}
```

This method uses the service layer to delete the guestbook by its ID. Since the `deleteGuestbook` action is invoked from the Guestbook Admin portlet's default view, there's no need to set the `mvcPath` render parameter to point to a particular JSP if there was a problem with the `deleteGuestbook` service call.

4. Hit [CTRL]+[SHIFT]+O to organize imports. Save the file.

You now have your service methods and portlet action methods in place. Your last task is to implement the Guestbook Admin portlet's user interface.

23.5 Creating a User Interface

<p>Writing the Guestbook Admin App<br>Step 5 of 5</p>

It's time to create the Guestbook Admin portlet's user interface. The portlet's default view has a button for adding new guestbooks. It must also display the guestbooks that already exist.

Each guestbook's name is displayed along with an Actions button. The Actions button reveals options for editing the guestbook, configuring its permissions, or deleting it.

Creating JSPs for the Guestbook Admin Portlet's User Interface

The Guestbook Admin portlet's user interface is made up of three JSPs: the default view, the Actions button, and the form for adding or editing a guestbook.

Create the default view first:

1. Create a folder for the Guestbook Admin portlet's JSPs. In `src/main/resources/META-INF/resources`, create a folder called `guestbookadminportlet`.

2. Create a file in this folder called `view.jsp` and fill it with this code:

```
<%@include file="../init.jsp"%>

<liferay-ui:search-container
    total="<%= GuestbookLocalServiceUtil.getGuestbooksCount(scopeGroupId) %>">
    <liferay-ui:search-container-results
        results="<%= GuestbookLocalServiceUtil.getGuestbooks(scopeGroupId,
            searchContainer.getStart(), searchContainer.getEnd()) %>" />

    <liferay-ui:search-container-row
        className="com.liferay.docs.guestbook.model.Guestbook" modelVar="guestbook">

        <liferay-ui:search-container-column-text property="name" />

        <liferay-ui:search-container-column-jsp
            align="right"
            path="/guestbookadminportlet/guestbook_actions.jsp" />

    </liferay-ui:search-container-row>

    <liferay-ui:search-iterator />
</liferay-ui:search-container>

<aui:button-row cssClass="guestbook-admin-buttons">
    <portlet:renderURL var="addGuestbookURL">
        <portlet:param name="mvcPath"
            value="/guestbookadminportlet/edit_guestbook.jsp" />
        <portlet:param name="redirect" value="<%= "currentURL" %>" />
    </portlet:renderURL>

    <aui:button onClick="<%= addGuestbookURL.toString() %>"
        value="Add Guestbook" />
</aui:button-row>
```

First is the standard `init.jsp` include to gain access to the imports.

Next is a button row with a single button for adding new guestbooks: `<aui:button-row cssClass="guestbook-admin-buttons">`. The `cssClass` attribute lets you specify a custom CSS class for additional styling. The `<portlet:renderURL>` tag constructs a URL that points to the edit_guestbook.jsp. You haven't created this JSP yet, but you'll use it for adding a new guestbook and editing an existing one.

Finally, a Liferay search container is used to display the list of guestbooks. Three sub-tags define the search container:

- <liferay-ui:search-container-results>
- <liferay-ui:search-container-row>
- <liferay-ui:search-iterator>

The <liferay-ui:search-container-results> tag's results attribute uses a service call to retrieve the guestbooks in the scope. The total attribute uses another service call to get a count of guestbooks.

The <liferay-ui:search-container-row> tag defines what rows contain. In this case, the className attribute defines com.liferay.docs.guestbook.model.Guestbook". The modelVar attribute defines guestbook as the variable for the currently iterated guestbook. In the search container row, two columns are defined. The <liferay-ui:search-container-column-text property="name" /> tag specifies the first column. This tag displays text. Its property="name" attribute specifies that the text to be displayed is the current guestbook object's name attribute. The tag <liferay-ui:search-container-column-jsp path="/guestbookadminportlet/guestbook_actions.jsp" align="right" /> specifies the second (and last) column. This tag includes another JSP file within a search container column. Its path attribute specifies the path to the JSP file that should be displayed: guestbook_actions.jsp.

Finally, the <liferay-ui:search-iterator /> tag iterates through and displays the list of guestbooks. Using Liferay's search container makes the Guestbook Admin portlet look like a native Liferay DXP portlet. It also provides built-in pagination so that your portlet can automatically display large numbers of guestbooks on one site.

Your next step is to add the guestbook_actions.jsp file that's responsible for displaying the list of possible actions for each guestbook.

3. Create a new file called guestbook_actions.jsp in your project's /guestbookadminportlet folder. Paste in this code:

```
<%@include file="../init.jsp"%>

<%
    String mvcPath = ParamUtil.getString(request, "mvcPath");

    ResultRow row = (ResultRow) request
                    .getAttribute("SEARCH_CONTAINER_RESULT_ROW");

    Guestbook guestbook = (Guestbook) row.getObject();
%>

<liferay-ui:icon-menu>

    <portlet:renderURL var="editURL">
        <portlet:param name="guestbookId"
            value="<%=String.valueOf(guestbook.getGuestbookId()) %>" />
        <portlet:param name="mvcPath"
            value="/guestbookadminportlet/edit_guestbook.jsp" />
    </portlet:renderURL>

    <liferay-ui:icon image="edit" message="Edit"
            url="<%=editURL.toString() %>" />
    <portlet:actionURL name="deleteGuestbook" var="deleteURL">
            <portlet:param name="guestbookId"
                value="<%= String.valueOf(guestbook.getGuestbookId()) %>" />
    </portlet:actionURL>

    <liferay-ui:icon-delete url="<%=deleteURL.toString() %>" />

</liferay-ui:icon-menu>
```

This JSP comprises the pop-up actions menu that shows the possible actions users can perform on a guestbook: editing it or deleting it. First, `init.jsp` is included because it contains all the JSP imports. Because `guestbook_actions.jsp` is included for every Search Container row, it retrieves the guestbook in the current iteration. The scriptlet grabs that guestbook so its ID can be supplied to the menu tags.

The `<liferay-ui:icon-menu` tag dominates `guestbook_actions.jsp`. It's a container for menu items, of which there are currently only two (you'll add more later). The Edit menu item displays the Edit icon and the message *Edit*:

```
<liferay-ui:icon image="edit" message="Edit"
        url="<%=editURL.toString() %>" />
```

The `editURL` variable comes from the `<portlet:renderURL var="editURL">` tag with two parameters: `guestbookId` and `mvcPath`. The `guestbookId` parameter specifies the guestbook to edit (it's the one from the selected search container result row), and the `mvcPath` parameter specifies the Edit Guestbook form's path.

The Delete menu item displays a delete icon and the default message *Delete*:

```
<liferay-ui:icon-delete url="<%=deleteURL.toString() %>" />
```

Unlike the `editURL`, which is a render URL that links to the `edit_guestbook.jsp`, the `deleteURL` is an action URL that invokes the portlet's `deleteGuestbook` action. The tag `<portlet:actionURL name="deleteGuestbook" var="deleteURL">` creates this action URL, which only takes one parameter: the `guestbookId` of the guestbook to be deleted.

Now there's just one more JSP file left to create: the `edit_guestbook.jsp` that contains the form for adding a new guestbook and editing an existing one.

4. Create a new file called `edit_guestbook.jsp` in your project's `/guestbookadminportlet` directory. Then add the following code to it:

```
<%@include file = "../init.jsp" %>

<%
        long guestbookId = ParamUtil.getLong(request, "guestbookId");

        Guestbook guestbook = null;

        if (guestbookId > 0) {
                guestbook = GuestbookLocalServiceUtil.getGuestbook(guestbookId);
        }
%>

<portlet:renderURL var="viewURL">
        <portlet:param name="mvcPath" value="/guestbookadminportlet/view.jsp" />
</portlet:renderURL>
```

```
<portlet:actionURL name='<%= guestbook == null ? "addGuestbook" : "updateGuestbook" %>' var="editGuestbookURL" />

<aui:form action="<%= editGuestbookURL %>" name="fm">

        <aui:model-context bean="<%= guestbook %>" model="<%= Guestbook.class %>" />

        <aui:input type="hidden" name="guestbookId"
            value='<%= guestbook == null ? "" : guestbook.getGuestbookId() %>' />

        <aui:fieldset>
             <aui:input name="name" />
        </aui:fieldset>

        <aui:button-row>
             <aui:button type="submit" />
             <aui:button onClick="<%= viewURL %>" type="cancel"  />
        </aui:button-row>
</aui:form>
```

After the `init.jsp` import, you declare a `null` guestbook variable. If there's a `guestbookId` parameter in the request, then you know that you're editing an existing guestbook, and you use the `guestbookId` to retrieve the corresponding guestbook via a service call. Otherwise, you know that you're adding a new guestbook.

Next is a view URL that points to the Guestbook Admin portlet's default view. This URL is invoked if the user clicks *Cancel* on the Add Guestbook or Edit Guestbook form. After that, you create an action URL that invokes either the Guestbook Admin portlet's `addGuestbook` method or its `updateGuestbook` method, depending on whether the `guestbook` variable is null.

If a guestbook is being edited, the current guestbook's name should appear in the form's name field. You use the following tag to define a model of the guestbook that can be used in the AlloyUI form:

```
<aui:model-context bean="<%= guestbook %>" model="<%= Guestbook.class %>" />
```

The form itself is created with the following tag:

```
<aui:form action="<%= editGuestbookURL %>" name="<portlet:namespace />fm">
```

When the form is submitted, the `editGuestbookURL` is invoked, which calls the Guestbook Admin portlet's `addGuestbook` or `updateGuestbook` method, as discussed above.

The `guestbookId` must appear on the form so that it can be submitted. The user, however, doesn't need to see it. Thus, you specify `type="hidden"`:

```
<aui:input type="hidden" name="guestbookId"
        value='<%= guestbook == null ? "" : guestbook.getGuestbookId() %>' />
```

The name, of course, should be editable by the user so it's not hidden.

The last item on the form is a button row with two buttons. The *Submit* button submits the form, invoking the `editGuestbookURL` which, in turn, invokes either the `addGuestbook` or `updateGuestbook` method. The *Cancel* button invokes the `viewURL` which displays the default view.

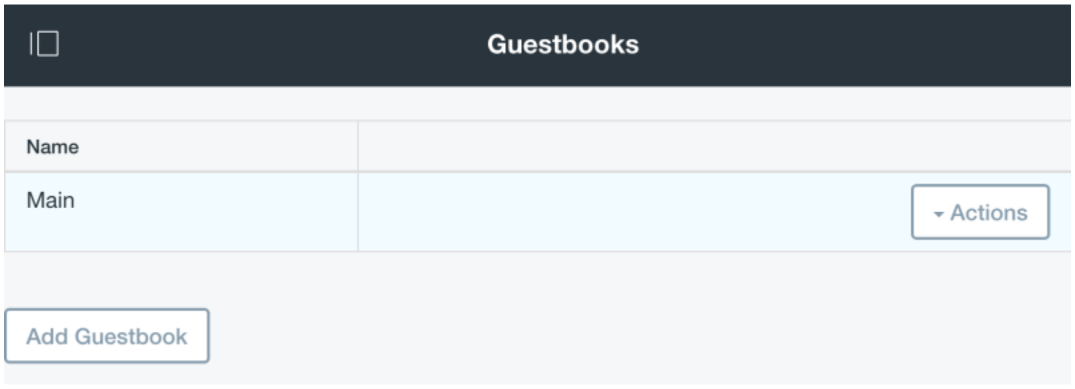


Figure 23.3: The Guestbook Admin portlet lets administrators add or edit guestbooks, configure their permissions, or delete them.

Excellent! You've now finished creating the UI for the Guestbook Admin portlet. It should now match the figure below:

Test out the Guestbook Admin portlet! Try adding, editing, and deleting guestbooks.

Now all the Guestbook application's primary functions work. There are still many missing features, however. For example, if there's ever an error, users never see it: all the code written so far just prints messages in the logs. Next, you'll learn how to display those errors to the user.

Chapter 24

Using Resources and Permissions

You now have an application that uses the database for data storage. This is a great foundation to build on. What comes next? What if users want a Guestbook that's limited to certain trusted people? To do that, you have to implement permissions.

Thankfully, with Liferay DXP you don't have to write an entire permissions system from scratch: the framework provides a robust and well-tested permissions system that you can implement quickly.

Ready to start?

Let's Go!

24.1 Configuring Your Permissions Scheme

<p>Implementing Permissions<br>Step 1 of 4</p>

Liferay DXP's permissions framework is configured declaratively, like Service Builder. You define all your permissions in an XML file that by convention is called `default.xml` (but you could really call it whatever you want). Then you implement permissions checks in the following places in your code:

- In the view layer, when showing links or buttons to protected functionality
- In the actions, before performing a protected action
- Later, in your service, before calling the local service

You should first define the permissions you want. To get started, think of your application's use cases and how access to that functionality should be controlled:

- The Add Guestbook button should be available only to administrators.
- The Guestbook tabs should be filtered by permissions so administrators can control who can see them.
- To prevent anonymous users from spamming the guestbook, the Add Entry button should be available only to site members.
- Users should be able to set permissions on their own entries.

Now you're ready to create the permissions configuration. Objects in your application (such as Guestbook and Entry) are defined as *resources*, and *resource actions* manage how users can interact with those resources. There are therefore two kinds of permissions: portlet permissions and resource (or model) permissions. Portlet permissions protect access to global functions, such as *Add Entry*. If users don't have permission to access that global function, they're missing a portlet permission. Resource permissions protect access to objects, such as Guestbook and Entry. A user may have permission to view one Entry, view and edit another Entry, and may not be able to access another Entry at all. This is due to a resource permission.

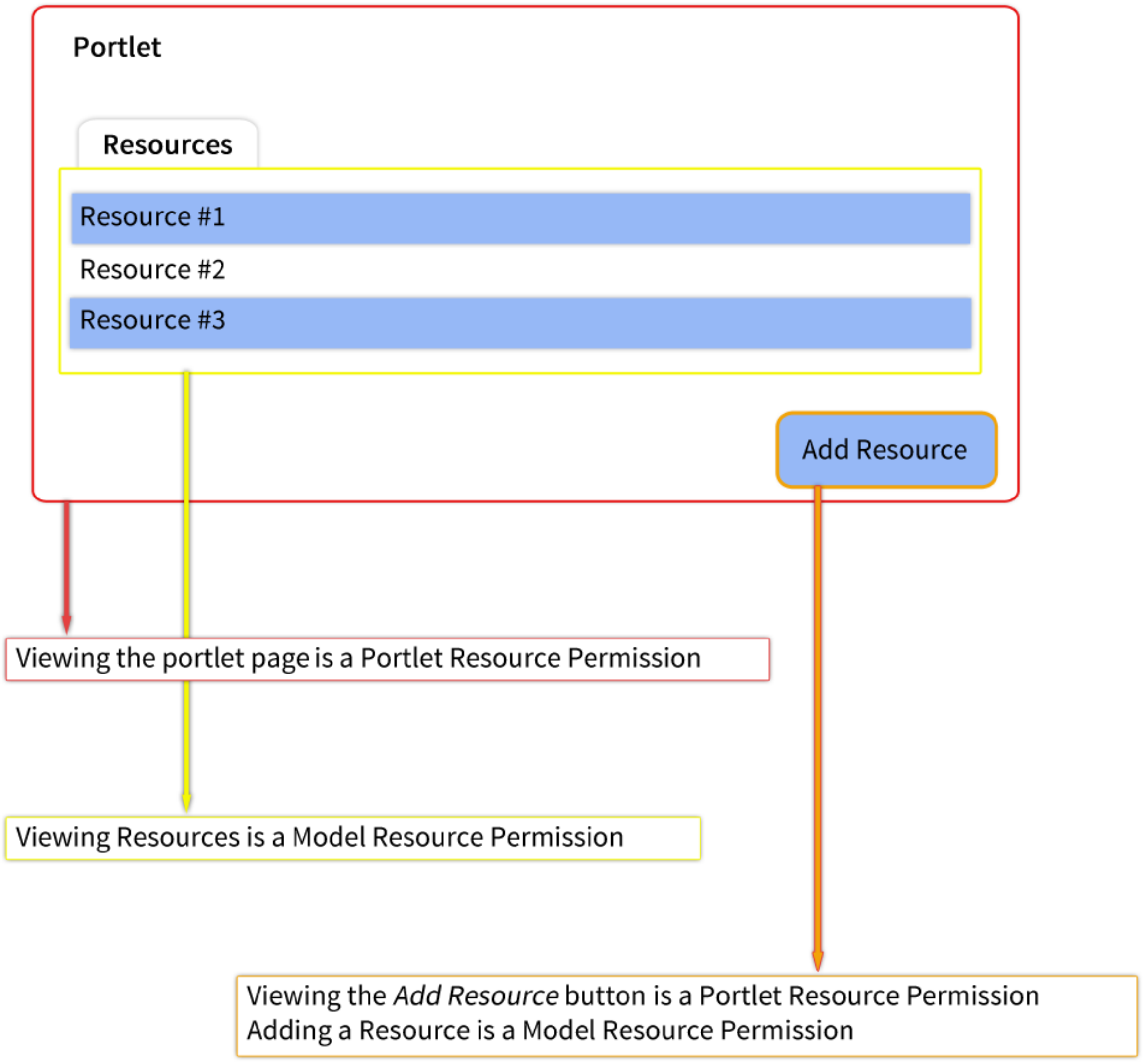

Portlet
Resources
Resource #1
Resource #2
Resource #3
Add Resource
Viewing the portlet page is a Portlet Resource Permission
Viewing Resources is a Model Resource Permission
Viewing the *Add Resource* button is a Portlet Resource Permission
Adding a Resource is a Model Resource Permission

Figure 24.1: Portlet permissions and resource permissions cover different parts of the application.

The first thing you must do is tell the framework where your permissions are defined. You'll define resource and model permissions in the module where your model is defined:

1. In guestbook-service's src/main/resources folder, create a file called portlet.properties.

2. In this file, place the following property:

```
resource.actions.configs=META-INF/resource-actions/default.xml
```

This property defines the name and location of your permissions definition file.
Next, create the permissions file:

1. In the META-INF folder, create a subfolder called resource-actions.

2. Create a new file in this folder called default.xml.

3. Click the *Source* tab. Add the following DOCTYPE declaration to the top of the file:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action
Mapping 7.0.0//EN" "http://www.liferay.com/dtd/liferay-resource-action-mapping_7_0_0.dtd">
```

4. Place the following wrapper tags into your default.xml file, below the DOCTYPE declaration:

```
<resource-action-mapping>

</resource-action-mapping>
```

   You'll define your resource and model permissions inside these tags.

5. Next, place the permissions for your com.liferay.docs.guestbook package between the <resource-action-mapping> tags:

```
<model-resource>
    <model-name>com.liferay.docs.guestbook</model-name>
    <portlet-ref>
        <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookPortlet</portlet-name>
        <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookAdminPortlet</portlet-name>
    </portlet-ref>
    <root>true</root>
    <permissions>
        <supports>
            <action-key>ADD_GUESTBOOK</action-key>
            <action-key>ADD_ENTRY</action-key>
            <action-key>VIEW</action-key>
        </supports>
        <site-member-defaults>
            <action-key>ADD_ENTRY</action-key>
        </site-member-defaults>
        <guest-defaults>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
            <action-key>ADD_GUESTBOOK</action-key>
            <action-key>ADD_ENTRY</action-key>
        </guest-unsupported>
    </permissions>
</model-resource>
```

   This defines the baseline configuration for the Guestbook and Entry entities. The supported actions are ADD_GUESTBOOK, ADD_ENTRY, and VIEW. Site members can ADD_ENTRY by default, while guests can't perform either add action (but they can view).

6. Below that, but above the closing `</resource-action-mapping>`, place the `Guestbook` model permissions:

```
<model-resource>
    <model-name>com.liferay.docs.guestbook.model.Guestbook</model-name>
    <portlet-ref>
        <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookPortlet</portlet-name>
        <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookAdminPortlet</portlet-name>
    </portlet-ref>
    <permissions>
        <supports>
            <action-key>ADD_ENTRY</action-key>
            <action-key>DELETE</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>UPDATE</action-key>
            <action-key>VIEW</action-key>
        </supports>
        <site-member-defaults>
            <action-key>ADD_ENTRY</action-key>
            <action-key>VIEW</action-key>
        </site-member-defaults>
        <guest-defaults>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
            <action-key>UPDATE</action-key>
        </guest-unsupported>
    </permissions>
</model-resource>
```

   This defines the `Guestbook` specific actions, including adding, deleting, updating, and viewing. By default, site members and guests can view guestbooks, but guests can't update them.

7. Below the `Guestbook` model permissions, but still above the closing `</resource-action-mapping>`, place the `Entry` model permissions:

```
<model-resource>
    <model-name>com.liferay.docs.guestbook.model.Entry</model-name>
    <portlet-ref>
        <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookPortlet</portlet-name>
    </portlet-ref>
    <permissions>
        <supports>
            <action-key>DELETE</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>UPDATE</action-key>
            <action-key>VIEW</action-key>
        </supports>
        <site-member-defaults>
            <action-key>VIEW</action-key>
        </site-member-defaults>
        <guest-defaults>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
            <action-key>UPDATE</action-key>
        </guest-unsupported>
    </permissions>
</model-resource>
```

   This defines `Entry` specific actions. By default, a site member can add or view an entry, and a guest can only view an entry.

8. Save the file.

This defines permissions at the model level, but you must also define portlet permissions. These are managed in the `guestbook-web` module, which contains the portlet class. Follow these steps to add the portlet permissions in the `guestbook-web` module:

1. In `guestbook-web`'s `src/main/resources` folder, create a file called `portlet.properties`.

2. In this file, place the following property:

   ```
   resource.actions.configs=META-INF/resource-actions/default.xml
   ```

3. Create a subfolder called `resource-actions` in the `src/main/resources/META-INF` folder.

4. Create a new file in this folder called `default.xml`.

5. Add the following `DOCTYPE` declaration to the top of the file:

   ```
   <?xml version="1.0"?>
   <!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action
   Mapping 7.0.0//EN" "http://www.liferay.com/dtd/liferay-resource-action-mapping_7_0_0.dtd">
   ```

6. Below the `DOCTYPE` declaration, add the following `resource-action-mapping` tags:

   ```
   <resource-action-mapping>

   </resource-action-mapping>
   ```

   You'll define your portlet permissions inside these tags.

7. Insert this block of code inside the `resource-action-mapping` tags:

   ```
   <portlet-resource>
       <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookAdminPortlet</portlet-name>
       <permissions>
           <supports>
               <action-key>ACCESS_IN_CONTROL_PANEL</action-key>
               <action-key>CONFIGURATION</action-key>
               <action-key>VIEW</action-key>
           </supports>
           <site-member-defaults>
               <action-key>VIEW</action-key>
           </site-member-defaults>
           <guest-defaults>
               <action-key>VIEW</action-key>
           </guest-defaults>
           <guest-unsupported>
               <action-key>ACCESS_IN_CONTROL_PANEL</action-key>
               <action-key>CONFIGURATION</action-key>
           </guest-unsupported>
       </permissions>
   </portlet-resource>
   ```

   This defines the default permissions for the Guestbook Admin portlet. It supports the actions `ACCESS_IN_CONTROL_PANEL`, `CONFIGURATION`, and `VIEW`. While anyone can view the app, guests and site members can't configure it or access it in the Control Panel. Since it's a Control Panel portlet, this effectively means that only administrators are able to access it.

8. Below the Guestbook Admin permissions, insert this block of code:

```
<portlet-resource>
    <portlet-name>com_liferay_docs_guestbook_portlet_GuestbookPortlet</portlet-name>
    <permissions>
        <supports>
            <action-key>ADD_TO_PAGE</action-key>
            <action-key>CONFIGURATION</action-key>
            <action-key>VIEW</action-key>
        </supports>
        <site-member-defaults>
            <action-key>VIEW</action-key>
        </site-member-defaults>
        <guest-defaults>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported />
    </permissions>
</portlet-resource>
```

This defines permissions for the Guestbook portlet. It supports the actions `ADD_TO_PAGE`, `CONFIGURATION`, and `VIEW`. Site members and guests get the `VIEW` permission by default.

9. Save the file.

Great job! You've now successfully designed and implemented a permissions scheme for your application. Next, you'll create the Java code to support permissions in the service layer.

24.2 Permissions in the Service Layer

```
<p>Implementing Permissions<br>Step 2 of 4</p>
```

The last step introduced the concept of *resources*. Resources are data stored with your entities that define how they can be accessed. For example, when the configuration in your `default.xml` files is applied to your application's entities in the database, resources are created. These resources are then used in conjunction with Liferay DXP's permissions system to determine who can do what to the entities.

Liferay DXP provides a complete API for managing resources that's integrated with Service Builder. This API is injected into your implementation classes automatically. To manage the resources, all you must do is call the API in the service's add and delete methods. Follow these steps to do this in your application:

1. In your `guestbook-service` module, open `GuestbookLocalServiceImpl.java` from the `com.liferay.docs.guestbook.servic` package.

2. Just before the `addGuestbook` method's `return` statement, add this code:

   ```
   resourceLocalService.addResources(user.getCompanyId(), groupId, userId,
       Guestbook.class.getName(), guestbookId, false, true, true);
   ```

   Note that the `resourceLocalService` object is already there, ready for you to use. This is one of several utilities that are injected automatically by Service Builder. You'll see the rest in the future.

   This code adds a resource to Liferay DXP's database to correspond with your entity (note that the `guestbookId` is included in the call). The three booleans at the end are settings. The first is whether to add portlet action permissions. This should only be `true` if the permission is for a portlet resource. Since this permission is for a model resource (an entity), it's `false`. The other two are settings for adding group and guest permissions. If you set these to `true`, you'll add the default permissions you defined in the permissions configuration file (`default.xml`) in the previous step. Since you definitely want to do this, these booleans are set to `true`.

3. Next, go to the `updateGuestbook` method. Add a similar bit of code in between `guestbookPersistence.update(guestbook);` and the `return` statement:

```
resourceLocalService.updateResources(serviceContext.getCompanyId(),
                serviceContext.getScopeGroupId(),
                Guestbook.class.getName(), guestbookId,
                serviceContext.getGroupPermissions(),
                serviceContext.getGuestPermissions());
```

4. Now you'll do the same for `deleteGuestbook`. Add this code in between `guestbook = deleteGuestbook(guestbook);` and the `return` statement:

```
resourceLocalService.deleteResource(serviceContext.getCompanyId(),
                Guestbook.class.getName(), ResourceConstants.SCOPE_INDIVIDUAL,
                guestbookId);
```

5. Hit [CTRL]+[SHIFT]+O to organize the imports and save the file.

6. Now you'll add resources for the `Entry` entity. Open `EntryLocalServiceImpl.java` from the same package. For `addEntry`, add a line of code that adds resources for this entity, just before the `return` statement:

```
resourceLocalService.addResources(user.getCompanyId(), groupId, userId,
    Entry.class.getName(), entryId, false, true, true);
```

7. For `deleteEntry`, add this code just before the `return` statement:

```
resourceLocalService.deleteResource(
               serviceContext.getCompanyId(), Entry.class.getName(),
               ResourceConstants.SCOPE_INDIVIDUAL, entryId);
```

8. Finally, find `updateEntry` and add its resource action, also just before the `return` statement:

```
resourceLocalService.updateResources(
      user.getCompanyId(), serviceContext.getScopeGroupId(),
      Entry.class.getName(), entryId, serviceContext.getGroupPermissions(),
      serviceContext.getGuestPermissions());
```

That's all it takes to add permissions resources. Future entities added to the database are fully permissions-enabled. Note, however, that any entities you've already added to your Guestbook application in the portal don't have resources and thus can't be protected by permissions. You'll fix this at the end of this section.

Next, you'll create helper classes to make it easier to check permissions.

24.3 Creating Permissions Helper Classes

```
<p>Implementing Permissions<br>Step 3 of 4</p>
```

You've now defined your permissions and made sure resources are added to the database so permissions can be checked. Now you'll create the helper classes needed to check permissions.

Here's how it works. You have a permission, such as `ADD_ENTRY`, and a resource, such as a `Guestbook`. For a user to add an entry to a guestbook, you must check whether that user has the `ADD_ENTRY` permission for that guestbook. Creating helper classes to check permissions on particular models and entities makes these checks more efficient. Creating such classes is therefore a best practice. Now you'll create these classes for the Guestbook application:

1. Right-click the guestbook-service module and select *New → Package*. Name the package com.liferay.docs.guestbook.service.permission. This is where you'll place your helper classes.

2. Right-click the new package and select *New → Class*. Name the class GuestbookModelPermission.

3. Replace this class's contents with the following code:

```
package com.liferay.docs.guestbook.service.permission;

import org.osgi.service.component.annotations.Component;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.security.auth.PrincipalException;
import com.liferay.portal.kernel.security.permission.BaseResourcePermissionChecker;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import com.liferay.portal.kernel.security.permission.ResourcePermissionChecker;

@Component(immediate = true, property = {
    "resource.name=" + GuestbookModelPermission.RESOURCE_NAME
}, service = ResourcePermissionChecker.class)

public class GuestbookModelPermission extends BaseResourcePermissionChecker {

    public static final String RESOURCE_NAME = "com.liferay.docs.guestbook";

    public static void check(
        PermissionChecker permissionChecker, long groupId, String actionId)
        throws PortalException {

        if (!contains(permissionChecker, groupId, actionId)) {
            throw new PrincipalException.MustHavePermission(
                permissionChecker, RESOURCE_NAME, groupId, actionId);
        }
    }

    public static boolean contains(
        PermissionChecker permissionChecker, long groupId, String actionId) {

        return permissionChecker.hasPermission(
            groupId, RESOURCE_NAME, groupId, actionId);
    }

    @Override
    public Boolean checkResource(
        PermissionChecker permissionChecker, long classPK, String actionId) {

        return contains(permissionChecker, classPK, actionId);
    }
}
```

This class is a component that extends BaseResourcePermissionChecker and defines two static methods (so you don't have to instantiate the class) that encapsulate the model you're checking permissions for. It also contains a boolean method that checks your resources. Liferay's PermissionChecker class does most of the work: you only need feed it the proper resource and action, such as ADD_ENTRY, and it returns whether the permission exists or not.

There are three implementations here: a check method that throws an exception if the user doesn't have permission, a contains method that returns a boolean that's true if the user has permission and false if not, and a checkResource method that calls the contains method.

Next, you'll create helpers for your two entities. Follow these steps to do so:

1. Create a class in the same package called GuestbookPermission.java.

2. Replace this class's contents with the following code:

```
package com.liferay.docs.guestbook.service.permission;

import com.liferay.docs.guestbook.model.Guestbook;
import com.liferay.docs.guestbook.service.GuestbookLocalService;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.security.auth.PrincipalException;
import com.liferay.portal.kernel.security.permission.BaseModelPermissionChecker;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

@Component(
    immediate = true,
    property = {"model.class.name=com.liferay.docs.guestbook.model.Guestbook"}
)
public class GuestbookPermission implements BaseModelPermissionChecker {

    public static void check(
        PermissionChecker permissionChecker, long guestbookId, String actionId)
        throws PortalException, SystemException {

        if (!contains(permissionChecker, guestbookId, actionId)) {
            throw new PrincipalException();
        }
    }

    public static void check(
        PermissionChecker permissionChecker, long groupId, long guestbookId,
        String actionId)
        throws PortalException {

        if (!contains(permissionChecker, groupId, actionId)) {
            throw new PrincipalException.MustHavePermission(
                permissionChecker, Guestbook.class.getName(), guestbookId,
                actionId);
        }
    }

    public static boolean contains(
        PermissionChecker permissionChecker, long groupId, long guestbookId, String actionId)
            throws PortalException {

        Guestbook guestbook = _guestbookLocalService.getGuestbook(guestbookId);

        return GuestbookModelPermission.contains(permissionChecker, groupId, actionId);
    }

    public static boolean contains(
        PermissionChecker permissionChecker, long guestbookId, String actionId)
        throws PortalException, SystemException {

        Guestbook guestbook
            = _guestbookLocalService.getGuestbook(guestbookId);
        return contains(permissionChecker, guestbook, actionId);
    }

    public static boolean contains(
        PermissionChecker permissionChecker, Guestbook guestbook, String actionId)
            throws PortalException, SystemException {

        return permissionChecker.hasPermission(
        guestbook.getGroupId(), Guestbook.class.getName(), guestbook.getGuestbookId(), actionId);

    }
    @Reference(unbind = "-")
    protected void setGuestbookLocalService(GuestbookLocalService guestbookLocalService) {
        _guestbookLocalService = guestbookLocalService;
    }

    private static GuestbookLocalService _guestbookLocalService;

    @Override
    public void checkBaseModel(
        PermissionChecker permissionChecker, long groupId, long guestbookId, String actionId) throws PortalException {
            check(permissionChecker, guestbookId, actionId);
    }
}
```

As you can see, this class is similar to GuestbookModelPermission. The difference is that GuestbookPermission is for the model/resource permission, so you supply the primary key of the entity you're checking permissions for (guestbookId). The check and contains methods in GuestbookPermission are also similar to those in GuestbookModelPermission. In both classes, the check method throws an exception if there's no permission, while the contains method returns a boolean denoting whether the current user has permission. The contains method in GuestbookPermission, however, also retrieves the entity to verify that it exists (if it doesn't, an exception is thrown).

Your final class is almost identical to GuestbookPermission, but it's for the Entry entity. Follow these steps to create it:

1. Create a class in the same package called EntryPermission.java.

2. Replace this class's contents with the following code:

```
package com.liferay.docs.guestbook.service.permission;

import com.liferay.docs.guestbook.model.Entry;
import com.liferay.docs.guestbook.service.EntryLocalService;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.security.auth.PrincipalException;
import com.liferay.portal.kernel.security.permission.BaseModelPermissionChecker;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

@Component(
    immediate = true,
    property = {"model.class.name=com.liferay.docs.guestbook.model.Entry"}
)
public class EntryPermission implements BaseModelPermissionChecker {

    public static void check(
        PermissionChecker permissionChecker, long entryId, String actionId)
        throws PortalException, SystemException {

        if (!contains(permissionChecker, entryId, actionId)) {
            throw new PrincipalException();
        }
    }

    public static boolean contains(
        PermissionChecker permissionChecker, long entryId, String actionId)
        throws PortalException, SystemException {

        Entry entry = _entryLocalService.getEntry(entryId);
```

```
            return contains (permissionChecker, entry, actionId);

        }

        public static boolean contains(
            PermissionChecker permissionChecker, Entry entry, String actionId) throws
            PortalException, SystemException {

            return permissionChecker.hasPermission(entry.getGroupId(), Entry.class.getName(), entry.getEntryId(), actionId);
        }

        @Reference(unbind = "-")
        protected void setEntryLocalService (EntryLocalService entryLocalService) {

            _entryLocalService = entryLocalService;
        }

        private static EntryLocalService _entryLocalService;

        @Override
        public void checkBaseModel(
            PermissionChecker permissionChecker, long groupId, long primaryKey, String actionId) throws PortalException {
                check(permissionChecker, primaryKey, actionId);
        }
    }
```

This class is almost identical to `GuestbookPermission`. The only difference is that `EntryPermission` is for the `Entry` entity.

Now that you have these classes, you must build services and export the permissions package so that other modules can access it. Follow these steps to do so:

1. Save the permissions helper classes you just created. From the Gradle Tasks panel on the right side of Liferay @ide@, run `buildService` from the `guestbook-service` module's `build` folder.

2. In the Project Explorer, open the `bnd.bnd` file from the root folder of the `guestbook-service` module.

3. In the graphical view, under the *Export Packages* section, click the plus button to add an export.

4. Select `com.liferay.docs.guestbook.service.permission` and click `OK`.

5. Save the file.

Congratulations! You've now created helper classes for your permissions. The only thing left is to implement permission checks in the application's view layer. You'll do this next.

24.4 Permissions in JSPs

```
<p>Implementing Permissions<br>Step 4 of 4</p>
```

User interface components can be wrapped in permission checks pretty easily. In this step, you'll learn how.

First go to the `init.jsp` in your `guestbook-web` project. Add the following imports to the file:

```
<%@ page import="com.liferay.docs.guestbook.service.permission.GuestbookModelPermission" %>
<%@ page import="com.liferay.docs.guestbook.service.permission.GuestbookPermission" %>
<%@ page import="com.liferay.docs.guestbook.service.permission.EntryPermission" %>
<%@ page import="com.liferay.portal.kernel.util.WebKeys" %>
<%@ page import="com.liferay.portal.kernel.security.permission.ActionKeys" %>
```

The first three are the permissions helper classes you just created. Now it's time to implement permission checks.

Checking Permissions in the UI

Recall that you want to restrict access to three areas in your application:

- The guestbook tabs across the top of your application
- The Add Guestbook button
- The Add Entry button

First, you'll create the guestbook tabs and check permissions for them. Follow these steps to do so:

1. Open `/guestbookwebportlet/view.jsp` and find the scriptlet that gets the `guestbookId` from the request. Just below this, add the following code:

```
<aui:nav cssClass="nav-tabs">

    <%
        List<Guestbook> guestbooks = GuestbookLocalServiceUtil.getGuestbooks(scopeGroupId);

            for (int i = 0; i < guestbooks.size(); i++) {

                Guestbook curGuestbook = (Guestbook) guestbooks.get(i);
                String cssClass = StringPool.BLANK;

                if (curGuestbook.getGuestbookId() == guestbookId) {
                    cssClass = "active";
                }

                if (GuestbookPermission.contains(
                    permissionChecker, curGuestbook.getGuestbookId(), "VIEW")) {

    %>

    <portlet:renderURL var="viewPageURL">
        <portlet:param name="mvcPath" value="/guestbookwebportlet/view.jsp" />
        <portlet:param name="guestbookId"
            value="<%=String.valueOf(curGuestbook.getGuestbookId())%>" />
    </portlet:renderURL>


    <aui:nav-item cssClass="<%=cssClass%>" href="<%=viewPageURL%>"
        label="<%=HtmlUtil.escape(curGuestbook.getName())%>" />

    <%
                }

            }
    %>

</aui:nav>
```

   This code gets a list of guestbooks from the database, iterates through them, checks the permission for each against the current user's roles, and adds the guestbooks the user can access to a list of tabs.

   You've now implemented your first permission check. As you can see, it's relatively straightforward thanks to the static methods in your helper classes. The code above shows the tab only if the current user has the `VIEW` permission for the guestbook.

   Next, you'll add permission checks to the Add Entry button.

2. Scroll down to the line that reads `<aui:button-row cssClass="guestbook-buttons">`. Just below this line, add the following line of code to check for the `ADD_ENTRY` permission:

```
<c:if test='<%= GuestbookPermission.contains(permissionChecker, guestbookId, "ADD_ENTRY") %>'>
```

3. After this is the code that creates the `addEntryURL` and the Add Entry button. After the `aui:button` tag and above the `</aui:button-row>` tag, add the closing tag for the `<c:if>` statement:

```
</c:if>
```

You've now implemented your permission check for the Add Entry button by using JSTL tags.

Next, you'll implement an `entry_actions.jsp` that's much like the one in the Guestbook Admin portlet. This will determine what options appear for logged in users who can see the actions menu in the portlet. Just like before, you'll wrap each `renderURL` in a `if` statement that checks the permissions against available actions. To do this, follow these steps:

1. In `src/main/resources/META-INF/resources/guestbookwebportlet`, create a file called `entry_actions.jsp`.

2. In this file, add the following code:

```
<%@include file="../init.jsp"%>

<%
String mvcPath = ParamUtil.getString(request, "mvcPath");

ResultRow row = (ResultRow)request.getAttribute(WebKeys.SEARCH_CONTAINER_RESULT_ROW);

Entry entry = (Entry)row.getObject();
%>

<liferay-ui:icon-menu>

    <portlet:renderURL var="viewEntryURL">
        <portlet:param name="entryId" value="<%= String.valueOf(entry.getEntryId()) %>" />
        <portlet:param name="mvcPath" value="/guestbookwebportlet/view_entry.jsp" />
    </portlet:renderURL>

    <liferay-ui:icon
        message="View"
        url="<%= viewEntryURL.toString() %>"
    />

    <c:if
        test="<%= EntryPermission.contains(permissionChecker, entry.getEntryId(), ActionKeys.UPDATE) %>">
        <portlet:renderURL var="editURL">
            <portlet:param name="entryId"
                value="<%= String.valueOf(entry.getEntryId()) %>" />
            <portlet:param name="mvcPath" value="/guestbookwebportlet/edit_entry.jsp" />
        </portlet:renderURL>

        <liferay-ui:icon image="edit" message="Edit"
            url="<%=editURL.toString() %>" />
    </c:if>

    <c:if
    test="<%=EntryPermission.contains(permissionChecker, entry.getEntryId(), ActionKeys.PERMISSIONS) %>">

        <liferay-security:permissionsURL
            modelResource="<%= Entry.class.getName() %>"
            modelResourceDescription="<%= entry.getMessage() %>"
            resourcePrimKey="<%= String.valueOf(entry.getEntryId()) %>"
            var="permissionsURL" />

        <liferay-ui:icon image="permissions" url="<%= permissionsURL %>" />

    </c:if>

    <c:if
        test="<%=EntryPermission.contains(permissionChecker, entry.getEntryId(), ActionKeys.DELETE) %>">

        <portlet:actionURL name="deleteEntry" var="deleteURL">
            <portlet:param name="entryId"
                value="<%= String.valueOf(entry.getEntryId()) %>" />
            <portlet:param name="guestbookId"
                value="<%= String.valueOf(entry.getGuestbookId()) %>" />
        </portlet:actionURL>

        <liferay-ui:icon-delete url="<%=deleteURL.toString() %>" />
    </c:if>

</liferay-ui:icon-menu>
```

This code defines several action buttons for viewing, updating, setting permissions on, and deleting entities. Each button is protected by a permissions check. If the current user can't perform the given action, the action doesn't appear.

3. Finally, in `view.jsp`, you must add the `entry_actions.jsp` as the last column in the Search Container. Find the line defining the Search Container row. It looks like this:

```
<liferay-ui:search-container-row
    className="com.liferay.docs.guestbook.model.Entry" modelVar="entry">
```

Below that line are two columns. After the second column, add a third:

```
<liferay-ui:search-container-column-jsp path="/guestbookwebportlet/entry_actions.jsp" align="right" />
```

4. Save all JSP files.

Excellent! You've now implemented all the permissions checks for the Guestbook portlet.

When testing the application, remember that any guestbook entries you created without resources won't work with permissions. Add new guestbooks and entries to test your application with different users. Administrative users see all the buttons, regular users see the Add Entry button, and guests see no buttons at all (but can navigate).

Note: You may see an error where the Guestbook portlet doesn't appear at all, and you see this error in the log:

```
Someone may be trying to circumvent the permission checker.
```

This is because any data you currently have in the Guestbook application doesn't have resources. In this case, you must drop and re-create your database. To do this, find your Liferay Workspace on your file system (it should be inside your Eclipse workspace). Inside the `bundles/data` folder is a `hypersonic` folder. Shut down Liferay DXP, remove everything from this folder, and then restart. After adding guestbook to a page, the portlet will work normally.

Now see if you can do the same for the Guestbook Admin portlet. Don't worry if you can't: at the end of this Learning Path is a link to the completed project for you to examine.

Great! You're all done with permissions. The next step is to integrate search and indexing into your application. This is a prerequisite for the much more powerful stuff to come.

Chapter 25

Displaying Messages and Errors

When users interact with your application, they perform tasks it defines, like saving or editing things. The Guestbook application is no different. Your application should also provide feedback on these operations so users can know if they worked. Up to now, you've been placing this information in logs that only administrators can access. Wouldn't it be better to show users these messages?

That's exactly what you'll do next, in three steps:

1. Create language keys for your messages.
2. Add the error messages to your action methods.
3. Report those error messages in your JSPs.

Ready to get started?

Let's Go!

25.1 Creating Language Keys

`<p>Displaying Messages and Errors<br>Step 1 of 3</p>`

Any modern application should place its messages and form field labels in a language keys file that can be duplicated and then translated into multiple languages. Here, you'll learn how to provide a *default* set of English language keys for your application. For more information on language keys and providing automatically translated language keys, see this tutorial.

Language keys are stored in the `Language.properties` file included in your `guestbook-web` module. `Language.properties` is the default, but you can create a number of translations by appending the ISO-639 language code to the file name (e.g., `Language_en.properties` for English or `Language_de.properties` for German). For now, stick to the default language keys.

Follow these steps to create your language keys:

1. Open `/src/main/resources/content/Language.properties` in your `guestbook-web` module. Remove the default keys in this file.

2. Paste in the following keys:

```
entry-added=Entry added successfully.
entry-deleted=Entry deleted successfully.
guestbook-added=Guestbook added successfully.
guestbook-updated=Guestbook updated successfully.
guestbook-deleted=Guestbook deleted successfully.
```

3. Save the file.

Your messages are now in place, and your application can use them. Next, you'll add them to your action methods.

25.2 Adding Failure and Success Messages

<p>Displaying Messages and Errors<br>Step 2 of 3</p>

To display correct feedback to users properly, you must edit your portlet classes to use Liferay DXP's `SessionMessages` and `SessionErrors` classes. These classes collect messages that the view layer shows to the user by using a simple tag.

You'll add these messages to code that runs when the user triggers a system function that can succeed or fail, such as creating, editing, or deleting an entry or guestbook. This generally happens in action methods. You must update these methods to handle failure and success states in `GuestbookPortlet.java` and `GuestbookAdminPortlet.java`. Start by updating `addEntry` and `deleteEntry` in `GuestbookPortlet.java`:

1. Find the `addEntry` method in `GuestbookPortlet.java`. In the first `try...catch` block's `try` section, add a success message just before the closing }:

   ```
   SessionMessages.add(request, "entryAdded");
   ```

   This uses Liferay's `SessionMessages` API to add a success message whenever a Guestbook is successfully added. It looks up the message you placed in the `Language.properties` file and inserts the message for the key `entry-added` (it automatically converts the key to camel case).

2. Below that, in the `catch` block, find the following code:

   ```
   System.out.println(e);
   ```

3. Beneath it, paste this line:

   ```
   SessionErrors.add(request, e.getClass().getName());
   ```

   Now you not only log the message to the console, you also use the `SessionErrors` object to show the message to the user.

Next, do the same for the `deleteEntry` method:

1. After the logic to delete the entry, add a success message:

   ```
   SessionMessages.add(request, "entryDeleted");
   ```

2. Find the same `Logger...` block of code in the `deleteEntry` method and after it, paste this line:

```
SessionErrors.add(request, e.getClass().getName());
```

3. Hit [CTRL]+[SHIFT]+O to import `com.liferay.portal.kernel.servlet.SessionErrors` and `com.liferay.portal.kernel.servlet.SessionMessages`. Save the file.

Well done! You've added the messages to `GuestbookPortlet`. Now you must update `GuestbookAdminPortlet.java`:

1. Open `GuestbookAdminPortlet.java` and look for the same cues.

2. Add the appropriate success messages to the try section of the `try...catch` in `addGuestbook`, `updateGuestbook`, and `deleteGuestbook`, respectively:

   ```
   SessionMessages.add(request, "guestbookAdded");
   ```

   ```
   SessionMessages.add(request, "guestbookUpdated");
   ```

   ```
   SessionMessages.add(request, "guestbookDeleted");
   ```

3. In the `catch` section of those same methods, find `Logger.getlogger...` and paste the `SessionErrors` block beneath it:

   ```
   SessionErrors.add(request, pe.getClass().getName());
   ```

4. Hit [CTRL]+[SHIFT]+O to import `SessionErrors` and `SessionMessages`. Save the file.

Great! The controller now makes relevant and detailed feedback available. Now all you need to do is publish this feedback in the view layer.

25.3 Adding Messages to JSPs

```
<p>Displaying Messages and Errors<br>Step 3 of 3</p>
```

Any messages the user should see are now stored in either `SessionMessages` or `SessionErrors`. Next, you'll make these messages appear in your JSPs.

1. In the `guestbook-web` module, open `guestbookwebportlet/view.jsp`. Add the following block of success messages to the top of the file, just below the `init.jsp` include statement:

   ```
   <liferay-ui:success key="entryAdded" message="entry-added" />
   <liferay-ui:success key="guestbookAdded" message="guestbook-added" />
   <liferay-ui:success key="entryDeleted" message="entry-deleted" />
   ```

   This tag accesses what's stored in `SessionMessages`. It has two attributes. The first is the `SessionMessages` key that you provided in the `GuestbookPortlet.java` class's add and delete methods. The second looks up the specified key in the `Language.properties` file. You could have specified a hard-coded message here, but it's far better to provide a localized key.

2. Now open `guestbookadminportlet/view.jsp`. Add the following block of success messages in the same spot below the include:

```
<liferay-ui:success key="guestbookAdded" message="guestbook-added" />
<liferay-ui:success key="guestbookUpdated" message="guestbook-updated" />
<liferay-ui:success key="guestbookDeleted" message="guestbook-deleted" />
```

Note that one of the `message` values is the same for both portlets. There's no need to write redundant messages–language keys are reusable.

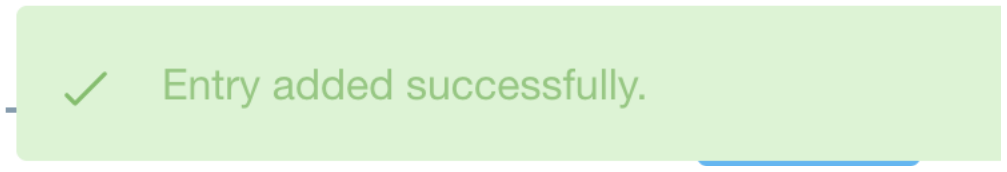

Entry added successfully.

Figure 25.1: Now the message will display the value you specified in `Language.properties`.

Congratulations! You've added useful feedback for operations in your application. Next, you'll add permission checking for your guestbooks and entries.

Chapter 26

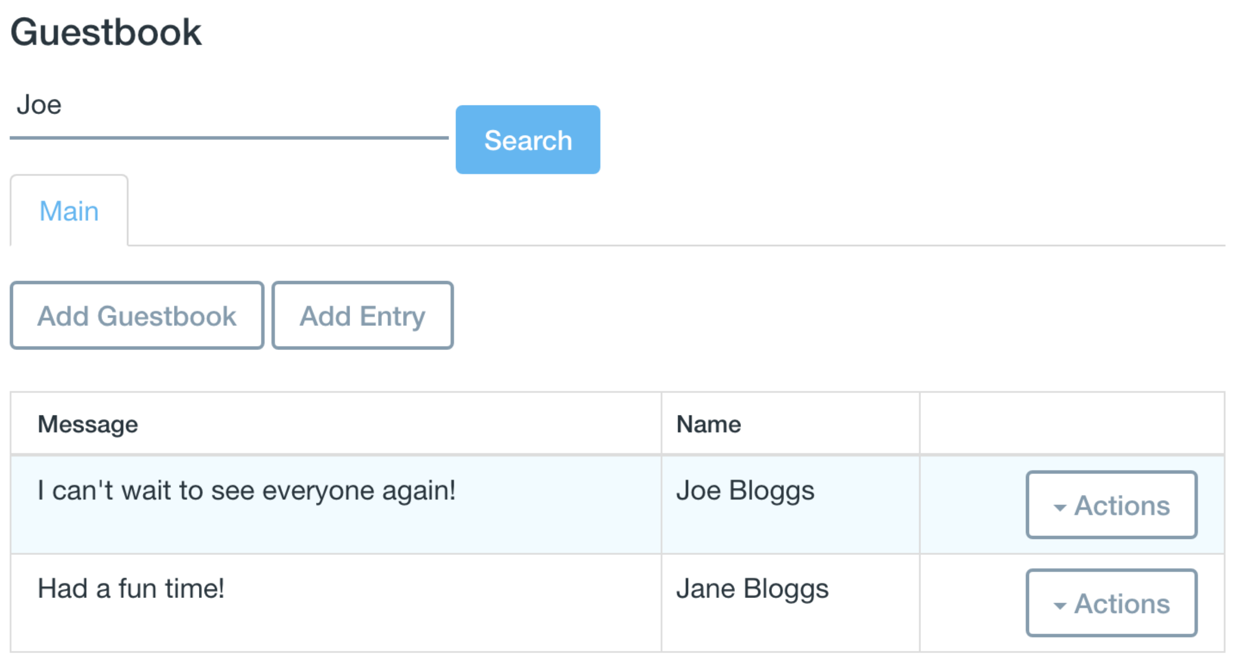
Leveraging Search

Now you have working Guestbook and Guestbook Admin portlets. The Guestbook portlet lets users add, edit, delete, and configure permissions for guestbook entries. The Guestbook Admin portlet lets site administrators create, edit, delete, and configure permissions for guestbooks. In the case of a very popular event (maybe a *Lunar Luau* dinner at the Lunar Resort), there could be many guestbook entries in the portlet, and users might want to search for entries that mentioned the delicious low-gravity ham that was served (melts in your mouth). Searching for the word *ham* should display these entries. In short, guestbook entries must be searchable via a search bar in the Guestbook portlet.

To enable search, you'll add an indexer for guestbooks and their entries. Although you probably won't have enough guestbooks in a site to warrant searching the Guestbook Admin portlet, creating a guestbook indexer has other benefits. In a later section, you'll asset-enable guestbooks and guestbook entries so Liferay DXP's Asset Publisher can display them. Enabling search is a prerequisite for this–you must index any entity that you want to make an asset.

But assets are for later. Right now it's time to create those indexers. Ready?

Let's Go!



Joe

Message	Name	
I can't wait to see everyone again!	Joe Bloggs	Actions
Had a fun time!	Jane Bloggs	Actions

Figure 26.1: You'll add a search bar to the Guestbook portlet so that users can search for guestbook entries. If a guestbook entry's message or name matches the search query, the entry is displayed in the search results.

Chapter 27

Enabling Search and Indexing for Guestbooks

In this section, you first create an indexer for guestbooks. You then modify the service layer to use this indexer to update the search index when a guestbook is persisted:

1. Create a `GuestbookIndexer` class that extends Liferay's `BaseIndexer` abstract class.

2. Update `GuestbookLocalServiceImpl`'s `addGuestbook`, `updateGuestbook`, and `deleteGuestbook` methods to invoke the guestbook indexer.

Since there's no reason to search for guestbooks in the UI, only the back-end work is necessary.
Let's Go!

27.1 Understanding Search and Indexing

`<p>Enabling Search and Indexing for Guestbooks<br>Step 1 of 3</p>`

By default, Liferay DXP uses Elasticsearch, a search engine backed by the popular Lucene search library, to implement its search and indexing functionality. To avoid the resource-hogging table merges necessary to search the database, using a search engine like Elasticsearch lets you convert searchable entities into *documents*. In Elasticsearch, documents are searchable database entities converted into JSON objects. After you implement an indexer for guestbook entries, Liferay DXP creates a document for each entry. This indexer specifies which guestbook entry fields to add to each guestbook entry document. All the guestbook entry documents are then added to an index. When the index is searched, a *hits* object is returned that contains pointers to the documents matching the search query. Searching for entities with a search engine via an index is faster than searching for entities in the database. Elasticsearch provides some additional features like relevancy scoring and fuzzy search queries.

Along with the search engine, Liferay DXP has its own search infrastructure. Liferay DXP adds to the existing Elasticsearch API for a few reasons:

- To ensure indexed documents include the fields needed by Liferay DXP (e.g., `entryClassName`, `entryClassPK`, `assetTagNames`, `assetCategories`, `companyId`, `groupId`, staging status).

- To ensure the scope of returned search results is appropriate by applying the right filters to search requests.
- To provide permission checking and hit summaries to display in the search portlet.

Next, you'll create the indexer for guestbooks.

27.2 Creating a Guestbook Indexer

<p>Enabling Search and Indexing for Guestbooks<br>Step 2 of 3</p>

First, update your `build.gradle` to have all of the necessary imports.

1. Open the `build.gradle` file in your `guestbook-service` project.
2. Add the following line below the other imports:

```
compileOnly group: "com.liferay", name: "com.liferay.registry.api", version: "1.0.0"
compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
```

3. Save the file and run `Refresh Gradle Project`.

Now that you have the additional dependencies configured, follow these steps to create the indexer for guestbooks:

1. Create a new package in the `guestbook-service` module project's `src/main/java` folder called `com.liferay.docs.guestbook.search`. In this package, create a new class called `GuestbookIndexer` that extends `com.liferay.portal.kernel.search.BaseIndexer` with `Guestbook` as a type argument. Add an `@Component` annotation to declare that the `GuestbookIndexer` class provides an implementation of the Indexer service. Also, define the `CLASS_NAME` variable by getting the name of the `Guestbook` model class. This is necessary to override the `getClassName` method from `BaseIndexer`. Liferay DXP uses this method to determine the object this Indexer indexes:

```
@Component(
    immediate = true,
    service = Indexer.class
)
public class GuestbookIndexer extends BaseIndexer<Guestbook> {

    public static final String CLASS_NAME = Guestbook.class.getName();
}
```

2. Add the `GuestbookIndexer` constructor:

```
public GuestbookIndexer() {
    setDefaultSelectedFieldNames(
        Field.ASSET_TAG_NAMES, Field.COMPANY_ID, Field.CONTENT,
        Field.ENTRY_CLASS_NAME, Field.ENTRY_CLASS_PK, Field.GROUP_ID,
        Field.MODIFIED_DATE, Field.SCOPE_GROUP_ID, Field.TITLE, Field.UID);
    setPermissionAware(true);
    setFilterSearch(true);
}
```

This constructor does several things:

- Sets the default selected field names. These fields are used to retrieve results documents from the search engine.
- Sets the default selected localized field names. This ensures that the localized version of the field is searched and returned.
- Makes the search results permissions-aware at search time, as well as in the index. Without this, a search query returns *all* matching guestbooks regardless of the user's permissions on the resource.
- Sets filter search to `true`, enabling a document-by-document check of the search results' `VIEW` permissions. This is redundant most of the time, but safeguards against unexpected problems like the search index becoming stale, or if permission inheritance doesn't happen fast enough. Most of Liferay DXP's internal apps use this setting. If not set, the indexer relies on the permissions information indexed in the search engine.

3. Since you extend the abstract class `BaseIndexer` instead of implementing the `Indexer` interface directly, you must override its abstract methods. In the `getClassName` method, return the `CLASS_NAME` constant defined at the top of the class:

```
@Override
public String getClassName() {
    return CLASS_NAME;
}
```

   This returns `com.liferay.docs.guestbook.model.Guestbook`.

4. Next, you must override `hasPermission`. Call the `contains` method of the `GuestbookPermission` helper class that you created in an earlier Learning Path section:

```
@Override
public boolean hasPermission(
        PermissionChecker permissionChecker, String entryClassName,
        long entryClassPK, String actionId)
    throws Exception {

    return GuestbookPermission.contains(
        permissionChecker, entryClassPK, ActionKeys.VIEW);
}
```

   Here, you ensure that the `VIEW` permission on guestbooks can be used to find and display appropriate search results.

5. Override the `postProcessContextBooleanFilter` method:

```
@Override
public void postProcessContextBooleanFilter(
        BooleanFilter contextBooleanFilter, SearchContext searchContext)
throws Exception {
    addStatus(contextBooleanFilter, searchContext);
}
```

   This method is invoked while the main search query is being constructed. The base implementation of `addStatus` in `BaseIndexer` adds the workflow status to the filter. This ensures that entities with the status `STATUS_IN_TRASH` aren't added to the query. You'll learn more about workflow later.

6. Override postProcessSearchQuery to add clauses to the ongoing search query. It's best to add the localized value of any full text fields that might contribute to search relevance. By specifying the localized search term, you ensure that the regular search term has the locale appended (e.g., title_en_US). For the guestbook entity, add the title field (Field.TITLE):

   ```
   @Override
   public void postProcessSearchQuery(
       BooleanQuery searchQuery, BooleanFilter fullQueryBooleanFilter,
       SearchContext searchContext)
       throws Exception {

       addSearchLocalizedTerm(searchQuery, searchContext, Field.TITLE, false);
   }
   ```

7. Override the doDelete() method, which deletes the document corresponding to the Guestbook object parameter. Call BaseIndexer's deleteDocument method with the guestbook's company ID and guestbook ID:

   ```
   @Override
   protected void doDelete(Guestbook guestbook) throws Exception {
       deleteDocument(guestbook.getCompanyId(), guestbook.getGuestbookId());
   }
   ```

8. Implement the doGetDocument method to select the entity's fields to build a search document that's indexed by the search engine. The main searchable field for guestbooks is the guestbook name, which is stored in a guestbook search document's title field:

   ```
   @Override
   protected Document doGetDocument(Guestbook guestbook)
       throws Exception {

       Document document = getBaseModelDocument(CLASS_NAME, guestbook);

       document.addDate(Field.MODIFIED_DATE, guestbook.getModifiedDate());

       Locale defaultLocale =
           PortalUtil.getSiteDefaultLocale(guestbook.getGroupId());
       String localizedField = LocalizationUtil.getLocalizedName(
           Field.TITLE, defaultLocale.toString());

       document.addText(localizedField, guestbook.getName());
       return document;
   }
   ```

   Because Liferay DXP supports localization, you should too. The above code gets the default locale from the site by passing the Guestbook's group ID to the getSiteDefaultLocale method, then using it to get the localized name of the guestbook's title field. The retrieved site locale is appended to the field (e.g., title_en_US), so the field gets passed to the search engine and goes through the right analysis and tokenization.

9. Implement the doGetSummary method to return a *summary*. A summary is a condensed, text-based version of the entity that can be displayed generically. You create it by combining key parts of the entity's data so users can browse through search results to find the entity they want. Call BaseIndexer's createSummary method, then use summary.setMaxContentLength to set the summary content's maximum size. Most Liferay DXP applications use a value of 200, so it's a good idea to use the same to ensure uniform result summaries:

```
@Override
protected Summary doGetSummary(
    Document document, Locale locale, String snippet,
    PortletRequest portletRequest, PortletResponse portletResponse) {

    Summary summary = createSummary(document);
    summary.setMaxContentLength(200);
    return summary;
}
```

10. Override the overloaded `doReindex` method, which gets called when an entity is updated or a user explicitly triggers a reindex. The first `doReindex` method takes a single object argument. Retrieve the associated document with `BaseIndexer`'s `getDocument` method, then invoke `IndexWriterHelper`'s `updateDocument` method to update (reindex) the document:

```
@Override
protected void doReindex(Guestbook guestbook)
    throws Exception {

    Document document = getDocument(guestbook);
    indexWriterHelper.updateDocument(
        getSearchEngineId(), guestbook.getCompanyId(), document,
        isCommitImmediately());
}
```

11. The second `doReindex` method takes two arguments: a `className` string, and a `classPK` long. In this method, you retrieve the guestbook corresponding to the primary key by calling `GuestbookLocalService`'s `getGuestbook` method, passing in the `classPK` parameter. Then pass the guestbook to the first `doReindex` method (see above):

```
@Override
protected void doReindex(String className, long classPK)
    throws Exception {

    Guestbook guestbook = _guestbookLocalService.getGuestbook(classPK);
    doReindex(guestbook);
}
```

12. The third (and final) `doReindex` method indexes all entities in the current Liferay DXP instance (`companyId`). It takes a string array (`ids`) as an argument. `GetterUtil.getLong(ids[0])` retrieves the first string in the array, casts it to a `long`, stores it in a `companyId` variable, and passes it as an argument to the `reindexGuestbooks` helper method:

```
@Override
protected void doReindex(String[] ids)
    throws Exception {

    long companyId = GetterUtil.getLong(ids[0]);
    reindexGuestbooks(companyId);
}
```

13. To reindex guestbooks, provide the helper method `reindexGuestbooks`. In this method, use an actionable dynamic query helper method to retrieve all the guestbooks in the Liferay DXP instance. Service Builder generated this query method for you when you built the services. Each guestbook's document is then retrieved and added to a collection:

```
protected void reindexGuestbooks(long companyId)
  throws PortalException {

  final IndexableActionableDynamicQuery indexableActionableDynamicQuery =
    _guestbookLocalService.getIndexableActionableDynamicQuery();

  indexableActionableDynamicQuery.setCompanyId(companyId);

  indexableActionableDynamicQuery.setPerformActionMethod(

    new ActionableDynamicQuery.PerformActionMethod<Guestbook>() {
      @Override
      public void performAction(Guestbook guestbook) {
        try {
          Document document = getDocument(guestbook);
          indexableActionableDynamicQuery.addDocuments(document);
        }
        catch (PortalException pe) {
          if (_log.isWarnEnabled()) {
            _log.warn(
              "Unable to index guestbook " +
                guestbook.getGuestbookId(),
              pe);
          }
        }
      }
    });
  indexableActionableDynamicQuery.setSearchEngineId(getSearchEngineId());
  indexableActionableDynamicQuery.performActions();
}
```

14. Get the log for the guestbook model and add the necessary service references at the bottom of the file:

```
private static final Log _log =
  LogFactoryUtil.getLog(GuestbookIndexer.class);

@Reference
protected IndexWriterHelper indexWriterHelper;

@Reference
private GuestbookLocalService _guestbookLocalService;
```

15. Organize your imports ([CTRL]+[SHIFT]+O), and save the file. It will have errors.

16. Export the `com.liferay.docs.guestbook.search` package in the `guestbook-service` module's `bnd.bnd` file. The export section should look like this:

```
Export-Package:
  com.liferay.docs.guestbook.service.permission,\
  com.liferay.docs.guestbook.search
```

The guestbook indexer class is complete! Next, you can update the service layer.

27.3 Handling Indexing in the Guestbook Service Layer

<p>Enabling Search and Indexing for Guestbooks<br>Step 3 of 3</p>

Whenever a guestbook database entity is added, updated, or deleted, the search index must be updated accordingly. The Liferay DXP annotations `@Indexable` and `@IndexableType` mark your service methods so documents can be updated or deleted. You must update the `addGuestbook`, `updateGuestbook`, and `deleteGuestbook` service methods with these annotations.

1. Open GuestbookLocalServiceImpl in the guestbook-service module's com.liferay.docs.guestbook.service.impl package, and add the following annotation above the method signature for the addGuestbook and updateGuestbook methods:

   ```
   @Indexable(type = IndexableType.REINDEX)
   public Guestbook addGuestbook(...)

   @Indexable(type = IndexableType.REINDEX)
   public Guestbook updateGuestbook(...)
   ```

   The @Indexable annotation indicates that an index update is required following the method execution. The GuestbookIndexer controls exactly how the indexing happens. Setting the @Indexable annotation type to IndexableType.REINDEX updates the document in the index that corresponds to the updated guestbook.

2. Add the following annotation above the method signature for the deleteGuestbook method:

   ```
   @Indexable(type = IndexableType.DELETE)
   public Guestbook deleteGuestbook(...)
   ```

   When a guestbook is deleted from the database, its document shouldn't remain in the search index. This ensures that it is deleted.

3. Add the necessary imports:

   ```
   import com.liferay.portal.kernel.search.Indexable;
   import com.liferay.portal.kernel.search.IndexableType;
   ```

   Save the file.

4. In the Gradle Tasks pane on the right-hand side of Liferay @ide@, double-click buildService in guestbook-service → build. This re-runs Service Builder to incorporate your changes to GuestbookLocalServiceImpl.

Great! Next, you'll enable search and indexing for guestbook entries.

Chapter 28

Enabling Search and Indexing for Guestbook Entries

Enabling search for guestbook entries in the Guestbook portlet takes two steps:

1. Create an EntryIndexer class that extends Liferay DXP's BaseIndexer abstract class.

2. Update EntryLocalServiceImpl's addEntry and deleteEntry methods to invoke the guestbook entry indexer.

When you finish, all the back-end search and indexing work for both entities will be complete, leaving only the UI changes to complete.

Let's Go!

28.1 Creating an Entry Indexer

<p>Enabling Search and Indexing for Guestbook Entries<br>Step 1 of 2</p>

The EntryIndexer class you'll complete here is very similar to the GuestbookIndexer class you completed in the previous section. Therefore, the instructions here only point out differences between the indexing of guestbooks and entries.

Follow these steps to create the entry indexer:

1. In the com.liferay.docs.guestbook.search package of your guestbook-service module project's src/main/java folder, create a new
   class called EntryIndexer that extends com.liferay.portal.kernel.search.BaseIndexer. Replace the default contents of EntryIndexer.java with the following code:

```
package com.liferay.docs.guestbook.search;

@Component(immediate = true, service = Indexer.class)
public class EntryIndexer extends BaseIndexer<Entry> {

    public static final String CLASS_NAME = Entry.class.getName();

    public EntryIndexer() {
```

```
    setDefaultSelectedFieldNames(
        Field.COMPANY_ID, Field.ENTRY_CLASS_NAME, Field.ENTRY_CLASS_PK,
        Field.UID, Field.SCOPE_GROUP_ID, Field.GROUP_ID);
    setDefaultSelectedLocalizedFieldNames(Field.TITLE, Field.CONTENT);
    setFilterSearch(true);
    setPermissionAware(true);
}

@Override
public String getClassName() {

    return CLASS_NAME;
}

@Override
public boolean hasPermission(
    PermissionChecker permissionChecker, String entryClassName,
    long entryClassPK, String actionId)
    throws Exception {

    return EntryPermission.contains(
        permissionChecker, entryClassPK, ActionKeys.VIEW);
}

@Override
public void postProcessContextBooleanFilter(
    BooleanFilter contextBooleanFilter, SearchContext searchContext)
    throws Exception {

    addStatus(contextBooleanFilter, searchContext);
}

@Override
public void postProcessSearchQuery(
    BooleanQuery searchQuery, BooleanFilter fullQueryBooleanFilter,
    SearchContext searchContext)
    throws Exception {

    addSearchLocalizedTerm(searchQuery, searchContext, "guestbookName", false);
    addSearchLocalizedTerm(searchQuery, searchContext, Field.TITLE, false);
    addSearchLocalizedTerm(searchQuery, searchContext, Field.CONTENT, false);
}

@Override
protected void doDelete(Entry entry)
    throws Exception {

    deleteDocument(entry.getCompanyId(), entry.getEntryId());
}

@Override
protected Document doGetDocument(Entry entry)
    throws Exception {

    Document document = getBaseModelDocument(CLASS_NAME, entry);
    document.addDate(Field.MODIFIED_DATE, entry.getModifiedDate());
    document.addText("email", entry.getEmail());

    Locale defaultLocale =
        PortalUtil.getSiteDefaultLocale(entry.getGroupId());
    String localizedTitle = LocalizationUtil.getLocalizedName(
        Field.TITLE, defaultLocale.toString());
    String localizedMessage = LocalizationUtil.getLocalizedName(
        Field.CONTENT, defaultLocale.toString());

    document.addText(localizedTitle, entry.getName());
    document.addText(localizedMessage, entry.getMessage());
```

```
        long guestbookId = entry.getGuestbookId();
        Guestbook guestbook = _guestbookLocalService.getGuestbook(guestbookId);
        String guestbookName= guestbook.getName();
        String localizedGbName = LocalizationUtil.getLocalizedName(
            "guestbookName", defaultLocale.toString());

        document.addText(localizedGbName, guestbookName);

        return document;
    }
```

This is not all the code, but it contains the heart of the functionality: the `doGetDocument` method and its helper methods. The email, date, localized title, and message fields (based on the site's default language) are indexed. Finally, you get the entry's guestbook and index the localized version of the `guestbookName` field. Always support localization where possible–this ensures your entities are searchable in any language.

2. The rest of the code is very similar to the `GuestbookIndexer`. Paste in the following code to finish the entry indexer class:

```
    @Override
    protected Summary doGetSummary(
        Document document, Locale locale, String snippet,
        PortletRequest portletRequest, PortletResponse portletResponse)
        throws Exception {

        Summary summary = createSummary(document);

        summary.setMaxContentLength(200);

        return summary;
    }

    @Override
    protected void doReindex(Entry entry)
        throws Exception {

        Document document = getDocument(entry);
        indexWriterHelper.updateDocument(
            getSearchEngineId(), entry.getCompanyId(), document,
            isCommitImmediately());
    }

    @Override
    protected void doReindex(String className, long classPK)
        throws Exception {

        Entry entry = _entryLocalService.getEntry(classPK);
        doReindex(entry);
    }

    @Override
    protected void doReindex(String[] ids)
        throws Exception {

        long companyId = GetterUtil.getLong(ids[0]);
        reindexEntries(companyId);
    }

    protected void reindexEntries(long companyId)
        throws PortalException {

        final IndexableActionableDynamicQuery indexableActionableDynamicQuery =
                _entryLocalService.getIndexableActionableDynamicQuery();

            indexableActionableDynamicQuery.setCompanyId(companyId);

            indexableActionableDynamicQuery.setPerformActionMethod(
                new ActionableDynamicQuery.PerformActionMethod<Entry>() {

                    @Override
                    public void performAction(Entry entry) {

                        try {
                            Document document = getDocument(entry);
                            indexableActionableDynamicQuery.addDocuments(document);
                        }
                        catch (PortalException pe) {
                            if (_log.isWarnEnabled()) {
                                _log.warn(
                                    "Unable to index entry " + entry.getEntryId(),
                                    pe);
                            }
                        }
                    }
                });
            indexableActionableDynamicQuery.setSearchEngineId(getSearchEngineId());
            indexableActionableDynamicQuery.performActions();
        }

        private static final Log _log = LogFactoryUtil.getLog(EntryIndexer.class);

        @Reference
        protected IndexWriterHelper indexWriterHelper;

        @Reference
        private EntryLocalService _entryLocalService;

        @Reference
        private GuestbookLocalService _guestbookLocalService;

    }
```

As with the guestbook, you must update the entry's service layer to support indexing when its service methods are called. That's your next step.

28.2 Handling Indexing in the Entry Service Layer

<p>Enabling Search and Indexing for Guestbook Entries<br>Step 2 of 2</p>

Whenever a guestbook entry is added, updated, or deleted, the corresponding document should also be updated or deleted. A minor update to each of the `addEntry`, `updateEntry`, and `deleteEntry` service methods for guestbook entries is all it takes.

Follow these steps to update the methods:

1. Open `EntryLocalServiceImpl` in the `guestbook-service` module's `com.liferay.docs.guestbook.service.impl` package, and add the annotation `@Indexable(type = IndexableType.REINDEX)` above the signature for the `addEntry` and `updateEntry` methods:

   ```
   @Indexable(type = IndexableType.REINDEX)
   public Entry addEntry(...)

   @Indexable(type = IndexableType.REINDEX)
   public Entry updateEntry(...)
   ```

The @Indexable annotation indicates that an index update is required following method execution. The EntryIndexer controls exactly how the indexing happens. Setting the @Indexable annotation's type to IndexableType.REINDEX updates the document in the index that corresponds to the updated entry.

2. Add the @Indexable(type = IndexableType.DELETE) annotation above the signature for the deleteEntry method. The indexable type IndexableType.DELETE ensures that the entry is deleted from the index:

   ```
   @Indexable(type = IndexableType.DELETE)
   public Entry deleteEntry(...)
   ```

3. Add the required imports:

   ```
   import com.liferay.portal.kernel.search.Indexable;
   import com.liferay.portal.kernel.search.IndexableType;
   ```

   Save the file.

4. In the Gradle Tasks pane on the right-hand side of Liferay @ide@, double-click buildService in guestbook-service → build. This re-runs Service Builder to incorporate your changes to EntryLocalServiceImpl.

Awesome! Both guestbooks and their entries now have search and indexing support in the back-end. Next, you'll enable search in the Guestbook portlet's front-end.

Chapter 29

Updating Your User Interface For Search

Updating the Guestbook portlet's user interface for search takes two steps:

1. Update the Guestbook portlet's default view JSP to display a search bar for submitting queries.
2. Create a new JSP for the Guestbook portlet to display search results.

You'll start by updating the Guestbook portlet's view JSP.
Let's Go!

29.1 Adding a Search Bar to the Guestbook Portlet

<p>Updating Your UI for Search<br>Step 1 of 2</p>

Follow these steps to create the search bar UI for the Guestbook portlet:

1. In `guestbook-web`, open the file `src/main/resources/META-INF/resources/guestbookwebportlet/view.jsp`. Add a render URL near the top of the file, just after the scriptlet that gets the `guestbookId` from the request:

   ```
   <liferay-portlet:renderURL varImpl="searchURL">
       <portlet:param name="mvcPath"
       value="/guestbookwebportlet/view_search.jsp" />
   </liferay-portlet:renderURL>
   ```

   The render URL points to `/guestbookwebportlet/view_search.jsp` (created in the next step). You construct the URL first because you must specify what happens when the user submits a search query.

2. Right after the render URL, create an AUI form that directs the user to the `view_search.jsp` page for viewing search results:

   ```
   <aui:form action="<%= searchURL %>" method="get" name="fm">
       <liferay-portlet:renderURLParams varImpl="searchURL" />

       <div class="search-form">
           <span class="aui-search-bar">
               <aui:input inlineField="<%= true %>" label=""
   ```

```
            name="keywords" size="30" title="search-entries" type="text"
            />

            <aui:button type="submit" value="search" />
        </span>
    </div>
</aui:form>
```

The tag `<liferay-portlet:renderURLParams varImpl="searchURL" />` includes the URL parameters of the `searchURL` as hidden input fields in the AUI form. This is important since the parameters of the `searchURL` are overwritten when the search query is submitted as a URL parameter.

The body of the search form consists of a `<div>` containing a `<span>` that contains two elements: the search bar and the search button. The `<aui:input>` tag defines the search bar. Its `name="keywords"` attribute specifies the name of the URL parameter that contains the search query. The `<aui:button>` tag defines the search button. The `type="submit"` attribute specifies that when the button is clicked (or the *Enter* key is pressed), the AUI form is submitted. The `value="search"` attribute specifies the name that appears on the button.

That's all there is to the search form! When the form is submitted, the `mvcPath` parameter pointing to the `view_search.jsp` is included in the URL along with the `keywords` parameter containing the search query. Now it's time to create the `view_search.jsp` form to display the search results.

29.2 Creating a Search Results JSP for the Guestbook Portlet

```
<p>Updating Your UI for Search<br>Step 2 of 2</p>
```

There are several design goals to implement in the search results JSP:

- Use a search container to display guestbook entries matching a search query.
- Make the Actions button available for each guestbook entry in the results, like it is in the main view's search container.
- Include the search bar so that users can edit and resubmit their queries without having to click the back link to go to the portlet's default view.

Follow these steps to create the search results JSP:

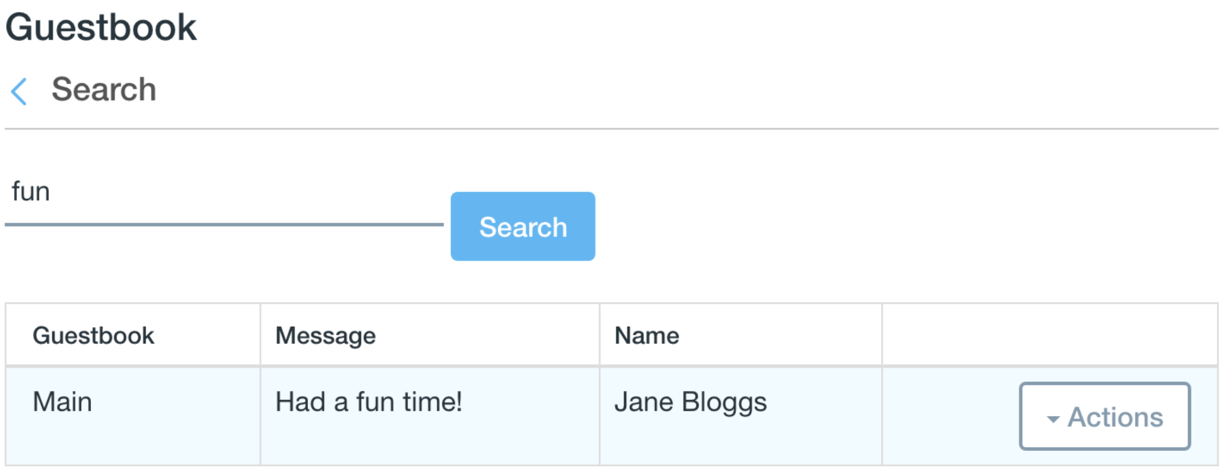
1. Create a new file called `view_search.jsp` in your `guestbook-web` module's `/guestbookwebportlet` folder. In this file, include the `init.jsp`:

   ```
   <%@include file="../init.jsp"%>
   ```

2. Extract the `keywords` and `guestbookId` parameters from the request. The `keywords` parameter contains the search query, and the `guestbookId` parameter contains the ID of the guestbook being searched:

   ```
   <%
     String keywords = ParamUtil.getString(request, "keywords");
     long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");
   %>
   ```

3. Define the `searchURL` and `viewURL` as `renderURL`s. Both use the `mvcPath` parameter that's available to Liferay MVC Portlets:



Guestbook	Message	Name	
Main	Had a fun time!	Jane Bloggs	Actions

Figure 29.1: The search results should appear in a search container, and the Actions button should appear for each entry. The search bar should also be displayed.

```
<liferay-portlet:renderURL varImpl="searchURL">
        <portlet:param name="mvcPath"
        value="/guestbookwebportlet/view_search.jsp" />
</liferay-portlet:renderURL>

<portlet:renderURL var="viewURL">
    <portlet:param
        name="mvcPath"
        value="/guestbookwebportlet/view.jsp"
    />
</portlet:renderURL>
```

The `searchURL` points to the current JSP: `view_search.jsp`. The `viewURL` points back to the Guestbook portlet's main view. These URLs are used in the AUI form that you'll create next.

4. Add this AUI form:

```
<aui:form action="<%= searchURL %>" method="get" name="fm">
    <liferay-portlet:renderURLParams varImpl="searchURL" />

<liferay-ui:header
    backURL="<%= viewURL.toString() %>"
    title="search"
/>

    <div class="search-form">
        <span class="aui-search-bar">
            <aui:input inlineField="<%= true %>" label="" name="keywords"
            size="30" title="search-entries" type="text" />

            <aui:button type="submit" value="search" />
        </span>
    </div>
</aui:form>
```

This form is identical to the one that you added to the Guestbook portlet's `view.jsp`, except that this one contains a `<liferay-ui:header>` tag that displays the Back icon next to the word *Search*. The `backURL` attribute in the header uses the viewURL defined above. Submitting the form invokes the searchURL with the user's search query added to the URL in the keywords parameter.

5. Start a scriptlet to get a search context and set some attributes in it:

```
<%
    SearchContext searchContext = SearchContextFactory
    .getInstance(request);

    searchContext.setKeywords(keywords);
    searchContext.setAttribute("paginationType", "more");
    searchContext.setStart(0);
    searchContext.setEnd(10);
```

   To execute a search, you need a SearchContext object. SearchContextFactory lets you create a SearchContext from the request object. Add the user's search query to the SearchContext by passing the keywords URL parameter to the setKeywords method. Then specify details about pagination and how the search results should be displayed.

6. Still in the scriptlet, obtain an Indexer to run a search. Retrieve the entry indexer from the map in Liferay DXP's indexer registry by passing in the indexer's class or class name:

```
Indexer indexer = IndexerRegistryUtil.getIndexer(Entry.class);
```

7. In the same scriptlet, use the indexer and the search context to run a search:

```
Hits hits = indexer.search(searchContext);

List<Entry> entries = new ArrayList<Entry>();

    for (int i = 0; i < hits.getDocs().length; i++) {
            Document doc = hits.doc(i);

            long entryId = GetterUtil
            .getLong(doc.get(Field.ENTRY_CLASS_PK));

            Entry entry = null;

            try {
                    entry = EntryLocalServiceUtil.getEntry(entryId);
            } catch (PortalException pe) {
                    _log.error(pe.getLocalizedMessage());
            } catch (SystemException se) {
                    _log.error(se.getLocalizedMessage());
            }

            entries.add(entry);
    }
```

   The search results return as Hits objects containing pointers to documents that correspond to guestbook entries. You then loop through the hit documents, retrieving the corresponding guestbook entries and adding them to a list.

8. Finish the scriptlet by retrieving a list of all the guestbooks that exist in the current site. Create a map between the guestbook IDs and the guestbook names.

```
        List<Guestbook> guestbooks = GuestbookLocalServiceUtil.getGuestbooks(scopeGroupId);

        Map<String, String> guestbookMap = new HashMap<String, String>();

        for (Guestbook guestbook : guestbooks) {
                guestbookMap.put(Long.toString(guestbook.getGuestbookId()), guestbook.getName());
        }
%>
```

Making this single service call and creating a map is more efficient than making separate service calls for each guestbook.

9. Display the search results in a search container:

```
<liferay-ui:search-container delta="10"
    emptyResultsMessage="no-entries-were-found"
    total="<%= entries.size() %>">
        <liferay-ui:search-container-results
                results="<%= entries %>"
/>
```

This specifies three attributes for the `<liferay-ui:search-container>` tag:

- `delta="10"`: specifies that at most, 10 entries can appear per page.
- `emptyResultsMessage`: specifies the message indicating there are no results.
- `total`: specifies the number of search results.

The `results` attribute of the tag `<liferay-ui:search-container-results>` specifies the search results. This is easy since you stored the entries resulting from the search in the `entries` list.

10. Use the `<liferay-ui:search-container-row>` tag to set the name of the class whose properties are displayed in each row:

```
<liferay-ui:search-container-row
        className="com.liferay.docs.guestbook.model.Entry"
        keyProperty="entryId" modelVar="entry" escapedModel="<%=true%>">
```

This uses the `className` attribute for the class name and specifies the entity's primary key attribute in the `keyProperty` attribute. The `modelVar` property specifies the name of the `Entry` variable that's available to each search container row. To ensure that each field of the `Entry` variable is escaped (sanitized), the `escapedModel` is `true`. This prevents potential hacks that could occur if users submitted malicious code into the Add Guestbook form, for example.

11. Inside the `<liferay-ui:search-container-row>` tag, specify the four columns to display: the guestbook entry's guestbook name, message, entry name, and the actions JSP. The guestbook name is retrieved from the map created in the scriptlet:

```
    <liferay-ui:search-container-column-text name="guestbook"
        value="<%=guestbookMap.get(Long.toString(entry.getGuestbookId()))%>" />

    <liferay-ui:search-container-column-text property="message" />

    <liferay-ui:search-container-column-text property="name" />

    <liferay-ui:search-container-column-jsp
        path="/guestbookwebportlet/entry_actions.jsp"
        align="right" />
</liferay-ui:search-container-row>
```

12. Use the `<liferay-ui:search-iterator>` tag to iterate through the search results and handle pagination. Close the search container tag:

```
        <liferay-ui:search-iterator />
</liferay-ui:search-container>
```

13. At the bottom of `view_search.jsp`, declare a `Log` object. You used this log in the `catch` clauses of the `try` clause that calls the `EntryLocalServiceUtil.getEntry` method to retrieve the guestbook entries. If this service call throws an exception, it's best to log the error so a server administrator can determine what went wrong. Liferay DXP's convention is to declare custom logs for individual classes or JSPs at the bottom of the file:

```
<%!
        private static Log _log = LogFactoryUtil.getLog("html.guestbookwebportlet.view_search_jsp");
%>
```

14. Finally, your `view_search.jsp` requires some extra imports. Add the following imports to `init.jsp`:

```
<%@ page import="com.liferay.portal.kernel.dao.search.SearchContainer" %>
<%@ page import="com.liferay.portal.kernel.exception.PortalException" %>
<%@ page import="com.liferay.portal.kernel.exception.SystemException" %>
<%@ page import="com.liferay.portal.kernel.language.LanguageUtil" %>
<%@ page import="com.liferay.portal.kernel.log.Log" %>
<%@ page import="com.liferay.portal.kernel.log.LogFactoryUtil" %>
<%@ page import="com.liferay.portal.kernel.search.Indexer" %>
<%@ page import="com.liferay.portal.kernel.search.IndexerRegistryUtil" %>
<%@ page import="com.liferay.portal.kernel.search.SearchContext" %>
<%@ page import="com.liferay.portal.kernel.search.SearchContextFactory" %>
<%@ page import="com.liferay.portal.kernel.search.Hits" %>
<%@ page import="com.liferay.portal.kernel.search.Document" %>
<%@ page import="com.liferay.portal.kernel.search.Field" %>
<%@ page import="com.liferay.portal.kernel.util.StringPool" %>
<%@ page import="com.liferay.portal.kernel.util.GetterUtil" %>
<%@ page import="com.liferay.portal.kernel.util.Validator" %>
<%@ page import="com.liferay.portal.kernel.util.PortalUtil" %>

<%@ page import="java.util.ArrayList" %>
<%@ page import="java.util.Map" %>
<%@ page import="java.util.HashMap" %>

<%@ page import="javax.portlet.PortletURL" %>
```

Good work! The Guestbook portlet now supports search! Now your users can find those Guestbook entries they were looking for.

The next section goes over Liferay DXP's asset framework, which provides shared functionality across different types of content like blog posts, message board posts, wiki articles, and more. This is the heart of integration with Liferay DXP's development platform.

Chapter 30

Assets: Integrating with Liferay's Framework

Liferay DXP's asset framework transforms entities into a common format that can be published anywhere in your site. Web content articles, blog posts, wiki articles, and documents are some asset-enabled entities that come out-of-the-box. By asset-enabling your own applications, you can take advantage of Liferay DXP's functionality for publishing your application's data across your site in the form of asset publisher entries, notifications, social activities, and more.

Liferay DXP's asset framework includes the following features:

- Tags and categories
- Comments and ratings
- Related assets (a.k.a. asset links)
- Faceted search
- Integration with Liferay DXP's Asset Publisher portlet
- Integration with Liferay DXP's Search portlet
- Integration with Liferay DXP's Tags Navigation, Tag Cloud, and Categories Navigation portlets

In this section, you'll asset-enable the guestbook and guestbook entry entities. You'll implement tags, categories, and related assets for guestbooks and guestbook entries. You'll implement comments and ratings in guestbook entries. You'll also learn how asset-enabled guestbooks and guestbook entries integrate with Liferay DXP core portlets like the Asset Publisher, Tags Navigation, Tag Cloud, and Categories Navigation portlets. Ready to start?

Let's Go!

Chapter 31

Enabling Assets at the Service Layer

<p>Enabling Assets at the Service Layer<br>Step 1 of 3</p>

Each row in the AssetEntry table represents an asset and has an entryId primary key, and classNameId and classPK foreign keys. The classNameId specifies the asset's type. For example, an asset with a classNameId of JournalArticle means that the asset represents a web content article (in Liferay DXP, JournalArticle is the back-end name for a web content article). An asset's classPK is the primary key of the entity represented by the asset.

Follow these steps to make Liferay DXP's asset services available to your entities' service layers:

1. In the guestbook-service module's service.xml file, add the following references directly above the closing </entity> tags for Guestbook and Entry:

   ```
   <reference package-path="com.liferay.portlet.asset" entity="AssetEntry" />
   <reference package-path="com.liferay.portlet.asset" entity="AssetLink" />
   ```

   As mentioned above, you must use Liferay DXP's AssetEntry service for your application to add asset entries that correspond to guestbooks and guestbook entries. You also use Liferay DXP's AssetLink service for your application to support related assets. *Asset links* are Liferay DXP's back-end term for related assets.

2. You must add finders–two for Guestbooks and two for Entitys–so your assets show in Asset Publisher. Add these below the existing finders for the Guestbook entity:

   ```
   <finder name="Status" return-type="Collection">
       <finder-column name="status" />
   </finder>

   <finder name="G_S" return-type="Collection">
       <finder-column name="groupId" />
       <finder-column name="status" />
   </finder>
   ```

   Add these below the existing finders for the Entry entity:

```
<finder name="Status" return-type="Collection">
    <finder-column name="status" />
</finder>
<finder name="G_S" return-type="Collection">
    <finder-column name="groupId" />
    <finder-column name="status" />
</finder>
```

3. Right-click `build.gradle` and select *Gradle → Refresh Gradle Project*.

4. Run the `buildService` Gradle task. This task causes the objects referenced above to be injected into your services for use.

Great! Next, you'll handle assets in your service layer.

31.1 Handling Assets at the Guestbook Service Layer

<p>Enabling Assets at the Service Layer<br>Step 2 of 3</p>

In this section, you'll update the guestbook service layer to use assets. You must update the add, update, and delete methods of your project's `GuestbookLocalServiceImpl`. Follow these steps to do so:

1. Open your project's `GuestbookLocalServiceImpl` class and find the `addGuestbook` method. Add the call to add the asset entries below the call that adds resources:

```
AssetEntry assetEntry = assetEntryLocalService.updateEntry(userId,
                groupId, guestbook.getCreateDate(),
                guestbook.getModifiedDate(), Guestbook.class.getName(),
                guestbookId, guestbook.getUuid(), 0,
                serviceContext.getAssetCategoryIds(),
                serviceContext.getAssetTagNames(), true, true, null, null, null, null,
                ContentTypes.TEXT_HTML, guestbook.getName(), null, null, null,
                null, 0, 0, null);

assetLinkLocalService.updateLinks(userId, assetEntry.getEntryId(),
                serviceContext.getAssetLinkEntryIds(),
                AssetLinkConstants.TYPE_RELATED);
```

   Calling `assetEntryLocalService.updateEntry` adds a new row (corresponding to the guestbook that's being added) to the `AssetEntry` table in Liferay DXP's database. `AssetEntryLocalServiceImpl`'s `updateEntry` method both adds and updates asset entries because it checks to see whether the asset entry already exists in the database and then takes the appropriate action. If you check the Javadoc for Liferay DXP's `AssetEntryLocalServiceUtil.updateEntry`, you'll see that this method is overloaded. Now, why did you use a version of this method with such a long method signature? Because there's only one version of `updateEntry` that takes a `title` parameter (to set the asset entry's title). Since you want to set the asset title to `guestbook.getName()`, that's the version you use.

   Later, you'll update the Guestbook Admin portlet's form for adding guestbooks to allow the selection of related assets, which are stored in the database's `AssetLink` table. The `assetLinkLocalService.updateLinks` call adds the appropriate entries to the table so related assets work for your guestbook entities. The `updateEntry` method adds and updates asset entries the same way `updateLink` adds and updates asset links.

2. Next, add the asset calls to `GuestbookLocalServiceImpl`'s `updateGuestbook` method, directly after the resource call:

```
AssetEntry assetEntry = assetEntryLocalService.updateEntry(guestbook.getUserId(),
                guestbook.getGroupId(), guestbook.getCreateDate(),
                guestbook.getModifiedDate(), Guestbook.class.getName(),
                guestbookId, guestbook.getUuid(), 0,
                serviceContext.getAssetCategoryIds(),
                serviceContext.getAssetTagNames(), true, true, guestbook.getCreateDate(),
                null, null, null, ContentTypes.TEXT_HTML, guestbook.getName(), null, null,
                null, null, 0, 0, serviceContext.getAssetPriority());

assetLinkLocalService.updateLinks(serviceContext.getUserId(),
                assetEntry.getEntryId(), serviceContext.getAssetLinkEntryIds(),
                AssetLinkConstants.TYPE_RELATED);
```

Here, `assetEntryLocalService.updateEntry` updates an existing asset entry and `assetLinkLocalService.updateLinks` adds or updates that entry's asset links (related assets).

3. Next, add the asset calls to the `deleteGuestbook` method, directly after the resource calls:

```
AssetEntry assetEntry = assetEntryLocalService.fetchEntry(
                Guestbook.class.getName(), guestbookId);

assetLinkLocalService.deleteLinks(assetEntry.getEntryId());

assetEntryLocalService.deleteEntry(assetEntry);
```

Here, you use the guestbook's class name and ID to retrieve the corresponding asset entry. Then you delete that asset entry's asset links and the asset entry itself.

4. Finally, organize your imports, save the file, and run Service Builder to apply the changes.

Next, you'll do the same thing for guestbook entries.

31.2 Handling Assets at the Entry Service Layer

<p>Enabling Assets at the Service Layer<br>Step 3 of 3</p>

Now you must update the guestbook entry entity's service methods. In these methods, the calls you'll make to `assetEntryLocalService` and `assetLinkLocalService` are identical to the ones you made in the guestbook entity's service methods. Follow these steps:

1. Open `EntryLocalServiceImpl` and add the asset calls to the `addEntry` method after the resource calls:

```
AssetEntry assetEntry = assetEntryLocalService.updateEntry(userId,
                groupId, entry.getCreateDate(), entry.getModifiedDate(),
                Entry.class.getName(), entryId, entry.getUuid(), 0,
                serviceContext.getAssetCategoryIds(),
                serviceContext.getAssetTagNames(), true, true, null, null, null, null,
                ContentTypes.TEXT_HTML, entry.getMessage(), null, null, null,
                null, 0, 0, null);

assetLinkLocalService.updateLinks(userId, assetEntry.getEntryId(),
                serviceContext.getAssetLinkEntryIds(),
                AssetLinkConstants.TYPE_RELATED);
```

2. Next, add the asset calls to the `updateEntry` method after the resource calls:

```
        AssetEntry assetEntry = assetEntryLocalService.updateEntry(userId,
                        serviceContext.getScopeGroupId(),
                        entry.getCreateDate(), entry.getModifiedDate(),
                        Entry.class.getName(), entryId, entry.getUuid(),
                        0, serviceContext.getAssetCategoryIds(),
                        serviceContext.getAssetTagNames(), true, true,
                        entry.getCreateDate(), null, null, null,
                        ContentTypes.TEXT_HTML, entry.getMessage(), null,
                        null, null, null, 0, 0,
                        serviceContext.getAssetPriority());

        assetLinkLocalService.updateLinks(userId, assetEntry.getEntryId(),
                        serviceContext.getAssetLinkEntryIds(),
                        AssetLinkConstants.TYPE_RELATED);
```

3. Add the asset calls to the `deleteEntry` method after the resource calls:

```
AssetEntry assetEntry = assetEntryLocalService.fetchEntry(
                Entry.class.getName(), entryId);

assetLinkLocalService.deleteLinks(assetEntry.getEntryId());

assetEntryLocalService.deleteEntry(assetEntry);
```

4. Organize your imports, save the file, and run Service Builder.

5. Finally, add these language keys to the `guestbook-web/src/main/resource/content/Language.properties` file:

```
model.resource.com.liferay.docs.guestbook.model.Guestbook=Guestbook
model.resource.com.liferay.docs.guestbook.model.Entry=Guestbook Entry
```

Excellent! You've asset-enabled your guestbook and guestbook entry entities at the service layer. Your next step is to implement asset renderers for these entities so they can be fully integrated into Liferay DXP's asset framework. Every asset needs an asset renderer class so the Asset Publisher portlet can display it.

Chapter 32

Implementing Asset Renderers

Assets are generic versions of entities, so they contain fields like `title`, `description`, and `summary`. Liferay DXP uses these fields to display assets. Asset Renderers translate an entity into an asset via these fields. For Liferay DXP to display your entities as assets, you must therefore create and register Asset Renderer classes for your guestbook and guestbook entry entities. Without these classes, Liferay DXP can't display your entities in Asset Publisher, Notifications, Activities, or anywhere else that displays assets.

Your next task is to create these Asset Renderers. Ready to begin?

Let's Go!

32.1 Implementing a Guestbook Asset Renderer

`<p>Implementing Asset Renderers<br>Step 1 of 2</p>`

Liferay DXP's asset renderers follow the factory pattern, so you must create a `GuestbookAssetRendererFactory` that instantiates the `GuestbookAssetRenderer`'s private guestbook object. Here, you'll create both classes.

Get started by creating the Asset Renderer class first.

Creating the AssetRenderer Class

Follow these steps to create the `GuestbookAssetRenderer` class:

1. Create a new package called `com.liferay.docs.guestbook.asset` in the `guestbook-service` module's `src/main/java` folder. In this package, create a `GuestbookAssetRenderer` class that extends Liferay DXP's `BaseJSPAssetRenderer` class. Extending this class gives you a head-start on implementing the `AssetRenderer` interface. Start with this code:

```
package com.liferay.docs.guestbook.asset;

import com.liferay.asset.kernel.model.BaseJSPAssetRenderer;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.model.LayoutConstants;
import com.liferay.portal.kernel.portlet.LiferayPortletRequest;
import com.liferay.portal.kernel.portlet.LiferayPortletResponse;
import com.liferay.portal.kernel.portlet.PortletURLFactoryUtil;
import com.liferay.portal.kernel.security.permission.ActionKeys;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import com.liferay.portal.kernel.util.HtmlUtil;
import com.liferay.portal.kernel.util.PortalUtil;
import com.liferay.portal.kernel.util.StringUtil;
import com.liferay.docs.guestbook.portlet.constants.GuestbookPortletKeys;
import com.liferay.docs.guestbook.model.Guestbook;
import com.liferay.docs.guestbook.service.permission.GuestbookPermission;
import java.util.Locale;
import javax.portlet.PortletRequest;
import javax.portlet.PortletResponse;
import javax.portlet.PortletURL;
import javax.portlet.WindowState;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class GuestbookAssetRenderer extends BaseJSPAssetRenderer<Guestbook> {

}
```

2. Add the constructor and the guestbook class variable next. Most of the methods in this class are simply getters that return fields from this private guestbook object:

```
public GuestbookAssetRenderer(Guestbook guestbook) {

            _guestbook = guestbook;
}

private Guestbook _guestbook;
```

3. The `BaseJSPAssetRenderer` abstract class that you're extending contains dummy implementations of the `hasEditPermission` and `hasViewPermission` methods that you must override. Override these dummy implementations with actual permission checks using the `GuestbookPermission` class that you created earlier:

```
@Override
public boolean hasEditPermission(PermissionChecker permissionChecker)
throws PortalException {

  long guestbookId = _guestbook.getGuestbookId();
  return GuestbookPermission.contains(permissionChecker, guestbookId,
  ActionKeys.UPDATE);
}

@Override
public boolean hasViewPermission(PermissionChecker permissionChecker)
throws PortalException {

  long guestbookId = _guestbook.getGuestbookId();
  return GuestbookPermission.contains(permissionChecker, guestbookId,
  ActionKeys.VIEW);
}
```

4. Add the following getter methods to retrieve information about the guestbook asset:

```
@Override
public Guestbook getAssetObject() {
  return _guestbook;
}

@Override
public long getGroupId() {
  return _guestbook.getGroupId();
}

@Override
public long getUserId() {

  return _guestbook.getUserId();
}

@Override
public String getUserName() {
  return _guestbook.getUserName();
}

@Override
public String getUuid() {
  return _guestbook.getUuid();
}

@Override
public String getClassName() {
  return Guestbook.class.getName();
}

@Override
public long getClassPK() {
  return _guestbook.getGuestbookId();
}

@Override
public String getSummary(PortletRequest portletRequest, PortletResponse
  portletResponse) {
    return "Name: " + _guestbook.getName();
}

@Override
public String getTitle(Locale locale) {
  return _guestbook.getName();
}

@Override
public boolean include(HttpServletRequest request, HttpServletResponse
  response, String template) throws Exception {
    request.setAttribute("GUESTBOOK", _guestbook);
    request.setAttribute("HtmlUtil", HtmlUtil.getHtml());
    request.setAttribute("StringUtil", new StringUtil());
    return super.include(request, response, template);
}
```

The final method makes several utilities, as well as the `Guestbook` entity, available to Liferay DXP in the `HttpServletRequest` object.

5. Override the `getJspPath` method. This method returns a string that represents the path to the JSP that renders the guestbook asset. When the Asset Publisher displays an asset's full content, it invokes the asset renderer class's `getJspPath` method and passes a `template` string parameter that equals `"full_content"`. This returns `/asset/guestbook/full_content.jsp` when the `full_content` template string is passed as a parameter. You'll create this JSP later when updating your application's user interface:

```
  @Override
  public String getJspPath(HttpServletRequest request, String template) {

    if (template.equals(TEMPLATE_FULL_CONTENT)) {
      request.setAttribute("gb_guestbook", _guestbook);
      return "/asset/guestbook/" + template + ".jsp";
    } else {
      return null;
    }
  }
```

6. Override the getURLEdit method. This method returns a URL for editing the asset:

```
@Override
public PortletURL getURLEdit(LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse) throws Exception {
  PortletURL portletURL = liferayPortletResponse.createLiferayPortletURL(
      getControlPanelPlid(liferayPortletRequest), GuestbookPortletKeys.GUESTBOOK,
      PortletRequest.RENDER_PHASE);
  portletURL.setParameter("mvcRenderCommandName", "/guestbookwebportlet/edit_guestbook");
  portletURL.setParameter("guestbookId", String.valueOf(_guestbook.getGuestbookId()));
  portletURL.setParameter("showback", Boolean.FALSE.toString());

  return portletURL;
}
```

7. Override the getURLViewInContext method. This method returns a URL to view the asset in its native application:

```
@Override
public String getURLViewInContext(LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse, String noSuchEntryRedirect) throws Exception {
  try {
    long plid = PortalUtil.getPlidFromPortletId(_guestbook.getGroupId(),
        GuestbookPortletKeys.GUESTBOOK);

    PortletURL portletURL;
    if (plid == LayoutConstants.DEFAULT_PLID) {
      portletURL = liferayPortletResponse.createLiferayPortletURL(getControlPanelPlid(liferayPortletRequest),
          GuestbookPortletKeys.GUESTBOOK, PortletRequest.RENDER_PHASE);
    } else {
      portletURL = PortletURLFactoryUtil.create(liferayPortletRequest,
          GuestbookPortletKeys.GUESTBOOK, plid, PortletRequest.RENDER_PHASE);
    }

    portletURL.setParameter("mvcRenderCommandName", "/guestbookwebportlet/view");
    portletURL.setParameter("guestbookId", String.valueOf(_guestbook.getGuestbookId()));

    String currentUrl = PortalUtil.getCurrentURL(liferayPortletRequest);

    portletURL.setParameter("redirect", currentUrl);

    return portletURL.toString();

  } catch (PortalException e) {

  } catch (SystemException e) {
  }

  return noSuchEntryRedirect;
}
```

8. Override the getURLView method. This method returns a URL to view the asset from within the Asset Publisher:

```
@Override
public String getURLView(LiferayPortletResponse liferayPortletResponse,
WindowState windowState) throws Exception {

  return super.getURLView(liferayPortletResponse, windowState);
}
```

9. Save the class.

10. You have an error in your class, because the `guestbook-service` project doesn't have access to the `GuestbookPortletKeys` object that's in the `guestbook-web` project.

    It is logical to think this could be corrected by including the project as a dependency in `guestbook-service`'s `build.gradle` file, but that creates a circular dependency. `guestbook-web` already depends on `guestbook-service`, so you can't make `guestbook-service` depend circularly on `guestbook-web`.

    So now what do you do?

    Make sure you've opened both `guestbook-api` and `guestbook-web` projects. Drag the `com.liferay.docs.guestbook.portlet` package from the `guestbook-web` project and drop it on the `guestbook-api` project's `src/main/java` folder. Blamo! You fixed the problem. `guestbook-service` depends on `guestbook-api` and implements its interfaces. `guestbook-web` depends on both. Now you have only linear dependencies.

Next you can create the `AssetRendererFactory` class.

Creating the GuestbookAssetRendererFactory Class

Follow these steps to create the `GuestbookAssetRendererFactory`:

1. In the `com.liferay.docs.guestbook.asset` package, create a class called `GuestbookAssetRendererFactory` that extends Liferay DXP's `BaseAssetRendererFactory` class. Replace its code with this starter code:

```
package com.liferay.docs.guestbook.asset;

import com.liferay.asset.kernel.model.AssetRenderer;
import com.liferay.asset.kernel.model.AssetRendererFactory;
import com.liferay.asset.kernel.model.BaseAssetRendererFactory;
import com.liferay.docs.guestbook.model.Guestbook;
import com.liferay.docs.guestbook.service.EntryLocalService;
import com.liferay.docs.guestbook.service.GuestbookLocalService;
import com.liferay.docs.guestbook.service.permission.GuestbookPermission;
import com.liferay.docs.guestbook.constants.GuestbookPortletKeys;
import com.liferay.portal.kernel.util.WebKeys;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.portlet.LiferayPortletRequest;
import com.liferay.portal.kernel.portlet.LiferayPortletResponse;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import com.liferay.portal.kernel.theme.ThemeDisplay;

import javax.portlet.PortletRequest;
import javax.portlet.PortletURL;
import javax.servlet.ServletContext;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;


@Component(immediate = true,
  property = {"javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK},
  service = AssetRendererFactory.class
  )
```

```
public class GuestbookAssetRendererFactory extends
  BaseAssetRendererFactory<Guestbook> {

  public GuestbookAssetRendererFactory() {
    setClassName(CLASS_NAME);
    setLinkable(_LINKABLE);
    setPortletId(GuestbookPortletKeys.GUESTBOOK);
    setSearchable(true);
    setSelectable(true);
  }
```

This code contains the class declaration and the constructor. It sets the class name it creates an `AssetRenderer` for, a portlet ID, and a boolean (`_LINKABLE`) set to `true`. The boolean denotes the methods that provide URLs in the generated `AssetRenderer` are implemented.

2. Implement the `getAssetRenderer` method, which constructs new `GuestbookAssetRenderer` instances for specific guestbooks. It uses the `classPK` (primary key) parameter to retrieve the guestbook from the database. It then calls the `GuestbookAssetRenderer`'s constructor, passing the retrieved guestbook as an argument:

```
@Override
public AssetRenderer<Guestbook> getAssetRenderer(long classPK, int type)
throws PortalException {

  Guestbook guestbook = _guestbookLocalService.getGuestbook(classPK);

  GuestbookAssetRenderer guestbookAssetRenderer =
  new GuestbookAssetRenderer(guestbook);

  guestbookAssetRenderer.setAssetRendererType(type);
  guestbookAssetRenderer.setServletContext(_servletContext);

  return guestbookAssetRenderer;
}
```

3. You're extending `BaseAssetRendererFactory`, an abstract class that implements the `AssetRendererFactory` interface. To ensure that your custom asset is associated with the correct entity, each asset renderer factory must implement the `getClassName` and `getType` methods (among others):

```
  @Override
  public String getClassName() {
    return CLASS_NAME;
  }

  @Override
  public String getType() {
    return TYPE;
  }
```

4. Implement the `hasPermission` method via the `GuestbookPermission` class:

```
@Override
public boolean hasPermission(PermissionChecker permissionChecker,
long classPK, String actionId) throws Exception {

  Guestbook guestbook = _guestbookLocalService.getGuestbook(classPK);
  return GuestbookPermission.contains(permissionChecker, guestbook,
  actionId);
}
```

5. Add the remaining code to create the portlet URL for the asset and specify whether it's linkable:

```
        @Override
        public PortletURL getURLAdd(LiferayPortletRequest liferayPortletRequest,
            LiferayPortletResponse liferayPortletResponse, long classTypeId) {
          PortletURL portletURL = null;

          try {
            ThemeDisplay themeDisplay = (ThemeDisplay)
            liferayPortletRequest.getAttribute(WebKeys.THEME_DISPLAY);

            portletURL = liferayPortletResponse.createLiferayPortletURL(getControlPanelPlid(themeDisplay),
                GuestbookPortletKeys.GUESTBOOK, PortletRequest.RENDER_PHASE);
            portletURL.setParameter("mvcRenderCommandName", "/guestbookwebportlet/edit_guestbook");
            portletURL.setParameter("showback", Boolean.FALSE.toString());
          } catch (PortalException e) {
          }

          return portletURL;
        }

        @Override
        public boolean isLinkable() {
          return _LINKABLE;
        }

        @Override
        public String getIconCssClass() {
            return "bookmarks";
        }

      @Reference(target = "(osgi.web.symbolicname=com.liferay.docs.guestbook.portlet)",
              unbind = "-")
      public void setServletContext(ServletContext servletContext) {
              _servletContext = servletContext;
          }
          private ServletContext _servletContext;

      @Reference(unbind = "-")
          protected void setGuestbookLocalService(GuestbookLocalService guestbookLocalService) {
              _guestbookLocalService = guestbookLocalService;
      }

      private GuestbookLocalService _guestbookLocalService;
      private static final boolean _LINKABLE = true;
      public static final String CLASS_NAME = Guestbook.class.getName();
      public static final String TYPE = "guestbook";
  }
```

6. Organize imports (Ctrl-Shift-O) and save the file.

Great! The guestbook asset renderer is complete. Next, you'll create the entry asset renderer.

32.2 Implementing an Entry Asset Renderer

<p>Implementing Asset Renderers<br>Step 2 of 2</p>

The classes you'll create here are nearly identical to the `GuestbookAssetRenderer` and `GuestbookAssetRendererFactory` classes you created for guestbooks in the previous step. This step provides the code needed for guestbook entries. Please review the previous sections for more information on this code.

Creating the EntryAssetRenderer Class

In the `com.liferay.docs.guestbook.asset` package, create an `EntryAssetRenderer` class that extends Liferay DXP's `BaseJSPAssetRenderer` class. Replace the contents of your `EntryAssetRenderer` class with the following code:

```
package com.liferay.docs.guestbook.asset;

import com.liferay.asset.kernel.model.BaseJSPAssetRenderer;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.exception.SystemException;
import com.liferay.portal.kernel.model.LayoutConstants;
import com.liferay.portal.kernel.portlet.LiferayPortletRequest;
import com.liferay.portal.kernel.portlet.LiferayPortletResponse;
import com.liferay.portal.kernel.portlet.PortletURLFactoryUtil;
import com.liferay.portal.kernel.security.permission.ActionKeys;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import com.liferay.portal.kernel.util.HtmlUtil;
import com.liferay.portal.kernel.util.PortalUtil;
import com.liferay.portal.kernel.util.StringUtil;
import com.liferay.docs.guestbook.portlet.constants.GuestbookPortletKeys;
import com.liferay.docs.guestbook.model.Entry;
import com.liferay.docs.guestbook.service.permission.EntryPermission;
import java.util.Locale;
import javax.portlet.PortletRequest;
import javax.portlet.PortletResponse;
import javax.portlet.PortletURL;
import javax.portlet.WindowState;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class EntryAssetRenderer extends BaseJSPAssetRenderer<Entry> {

  public EntryAssetRenderer(Entry entry) {

    _entry = entry;
  }

  @Override
  public boolean hasViewPermission(PermissionChecker permissionChecker)
  throws PortalException {

    long entryId = _entry.getEntryId();
    return EntryPermission.contains(permissionChecker, entryId,
    ActionKeys.VIEW);
  }

  @Override
  public Entry getAssetObject() {
    return _entry;
  }

  @Override
  public long getGroupId() {
    return _entry.getGroupId();
  }

  @Override
  public long getUserId() {

    return _entry.getUserId();
  }

  @Override
  public String getUserName() {
    return _entry.getUserName();
  }
```

```
@Override
public String getUuid() {
  return _entry.getUuid();
}

@Override
public String getClassName() {
  return Entry.class.getName();
}

@Override
public long getClassPK() {
  return _entry.getEntryId();
}

@Override
public String getSummary(PortletRequest portletRequest,
PortletResponse portletResponse) {
  return "Name: " + _entry.getName() + ". Message: " + _entry.getMessage();
}

@Override
public String getTitle(Locale locale) {
  return _entry.getMessage();
}

@Override
public boolean include(HttpServletRequest request,
HttpServletResponse response, String template) throws Exception {
  request.setAttribute("ENTRY", _entry);
  request.setAttribute("HtmlUtil", HtmlUtil.getHtml());
  request.setAttribute("StringUtil", new StringUtil());
  return super.include(request, response, template);
}

@Override
public String getJspPath(HttpServletRequest request, String template) {

  if (template.equals(TEMPLATE_FULL_CONTENT)) {
    request.setAttribute("gb_entry", _entry);

    return "/asset/entry/" + template + ".jsp";
  } else {
    return null;
  }
}

@Override
public PortletURL getURLEdit(LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse) throws Exception {
  PortletURL portletURL = liferayPortletResponse.createLiferayPortletURL(
      getControlPanelPlid(liferayPortletRequest), GuestbookPortletKeys.GUESTBOOK,
      PortletRequest.RENDER_PHASE);
  portletURL.setParameter("mvcRenderCommandName", "/guestbookwebportlet/edit_entry");
  portletURL.setParameter("entryId", String.valueOf(_entry.getEntryId()));
  portletURL.setParameter("showback", Boolean.FALSE.toString());

  return portletURL;
}

@Override
public String getURLViewInContext(LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse, String noSuchEntryRedirect)
    throws Exception {
  try {
    long plid = PortalUtil.getPlidFromPortletId(_entry.getGroupId(),
        GuestbookPortletKeys.GUESTBOOK);
      PortletURL portletURL;
      if (plid == LayoutConstants.DEFAULT_PLID) {
        portletURL = liferayPortletResponse.createLiferayPortletURL(getControlPanelPlid(liferayPortletRequest),
            GuestbookPortletKeys.GUESTBOOK, PortletRequest.RENDER_PHASE);
      } else {
        portletURL = PortletURLFactoryUtil.create(liferayPortletRequest,
            GuestbookPortletKeys.GUESTBOOK, plid, PortletRequest.RENDER_PHASE);
      }

      portletURL.setParameter("mvcRenderCommandName", "/guestbookwebportlet/view");
      portletURL.setParameter("entryId", String.valueOf(_entry.getEntryId()));

      String currentUrl = PortalUtil.getCurrentURL(liferayPortletRequest);

      portletURL.setParameter("redirect", currentUrl);

      return portletURL.toString();

    } catch (PortalException e) {

    } catch (SystemException e) {
    }

    return noSuchEntryRedirect;
  }

  @Override
  public String getURLView(LiferayPortletResponse liferayPortletResponse,
  WindowState windowState) throws Exception {

    return super.getURLView(liferayPortletResponse, windowState);
  }

  @Override
  public boolean isPrintable() {
        return true;
  }

  private Entry _entry;
}
```

This class is similar to the `GuestbookAssetRenderer` class. For the `EntryAssetRenderer.getSummary` method, you return a summary that displays the entry name (the name of the user who created the entry) and the entry message.

`GuestbookAssetRenderer.getSummary` returns a summary that displays the guestbook name. `EntryAssetRenderer.getTitle` returns the entry message. `GuestbookAssetRenderer.getTitle` returns the guestbook name. The rest of the methods of `EntryAssetRenderer` are nearly identical to those of `GuestbookAssetRenderer`.

Creating the EntryAssetRendererFactory Class

Next, you must create the guestbook entry asset renderer's factory class. In the `com.liferay.docs.guestbook.asset` package, create a class called `EntryAssetRendererFactory` that extends Liferay DXP's `BaseAssetRendererFactory` class. Replace its contents with the following code:

```
package com.liferay.docs.guestbook.asset;

import com.liferay.asset.kernel.model.AssetRenderer;
import com.liferay.asset.kernel.model.AssetRendererFactory;
import com.liferay.asset.kernel.model.BaseAssetRendererFactory;
import com.liferay.docs.guestbook.model.Entry;
import com.liferay.docs.guestbook.service.EntryLocalService;
import com.liferay.docs.guestbook.service.permission.EntryPermission;
import com.liferay.docs.guestbook.portlet.constants.GuestbookPortletKeys;
import com.liferay.portal.kernel.util.WebKeys;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.portlet.LiferayPortletRequest;
import com.liferay.portal.kernel.portlet.LiferayPortletResponse;
import com.liferay.portal.kernel.portlet.LiferayPortletURL;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import com.liferay.portal.kernel.theme.ThemeDisplay;

import javax.portlet.PortletRequest;
import javax.portlet.PortletURL;
import javax.portlet.WindowState;
import javax.portlet.WindowStateException;
import javax.servlet.ServletContext;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

@Component(
    immediate = true,
    property = {"javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK},
    service = AssetRendererFactory.class
)
public class EntryAssetRendererFactory extends BaseAssetRendererFactory<Entry> {

    public EntryAssetRendererFactory() {
        setClassName(CLASS_NAME);
        setLinkable(_LINKABLE);
        setPortletId(GuestbookPortletKeys.GUESTBOOK);
        setSearchable(true);
        setSelectable(true);
    }

    @Override
    public AssetRenderer<Entry> getAssetRenderer(long classPK, int type)
        throws PortalException {

        Entry entry = _entryLocalService.getEntry(classPK);

        EntryAssetRenderer entryAssetRenderer = new EntryAssetRenderer(entry);

        entryAssetRenderer.setAssetRendererType(type);
        entryAssetRenderer.setServletContext(_servletContext);

        return entryAssetRenderer;
    }

    @Override
    public String getClassName() {
        return CLASS_NAME;
    }

    @Override
    public String getType() {
        return TYPE;
    }

    @Override
    public boolean hasPermission(PermissionChecker permissionChecker,
        long classPK, String actionId) throws Exception {

        Entry entry = _entryLocalService.getEntry(classPK);
        return EntryPermission.contains(permissionChecker, entry, actionId);
    }

    @Override
    public PortletURL getURLAdd(LiferayPortletRequest liferayPortletRequest,
        LiferayPortletResponse liferayPortletResponse, long classTypeId) {

        PortletURL portletURL = null;

        try {
            ThemeDisplay themeDisplay = (ThemeDisplay) liferayPortletRequest.getAttribute(WebKeys.THEME_DISPLAY);

            portletURL = liferayPortletResponse.createLiferayPortletURL(getControlPanelPlid(themeDisplay),
                GuestbookPortletKeys.GUESTBOOK, PortletRequest.RENDER_PHASE);
            portletURL.setParameter("mvcRenderCommandName", "/guestbookwebportlet/edit_entry");
            portletURL.setParameter("showback", Boolean.FALSE.toString());
        } catch (PortalException e) {
        }

        return portletURL;
    }

    @Override
    public PortletURL getURLView(LiferayPortletResponse liferayPortletResponse, WindowState windowState) {

        LiferayPortletURL liferayPortletURL
            = liferayPortletResponse.createLiferayPortletURL(
                GuestbookPortletKeys.GUESTBOOK, PortletRequest.RENDER_PHASE);

        try {
            liferayPortletURL.setWindowState(windowState);
        } catch (WindowStateException wse) {

        }
        return liferayPortletURL;
    }

    @Override
    public boolean isLinkable() {
        return _LINKABLE;
    }

    @Override
    public String getIconCssClass() {
        return "pencil";
    }

    @Reference(target = "(osgi.web.symbolicname=com.liferay.docs.guestbook.portlet)",
        unbind = "-")
    public void setServletContext (ServletContext servletContext) {
        _servletContext = servletContext;
    }

    @Reference(unbind = "-")
    protected void setEntryLocalService(EntryLocalService entryLocalService) {
        _entryLocalService = entryLocalService;
    }

    private EntryLocalService _entryLocalService;
    private ServletContext _servletContext;
    private static final boolean _LINKABLE = true;
    public static final String CLASS_NAME = Entry.class.getName();
    public static final String TYPE = "entry";

}
```

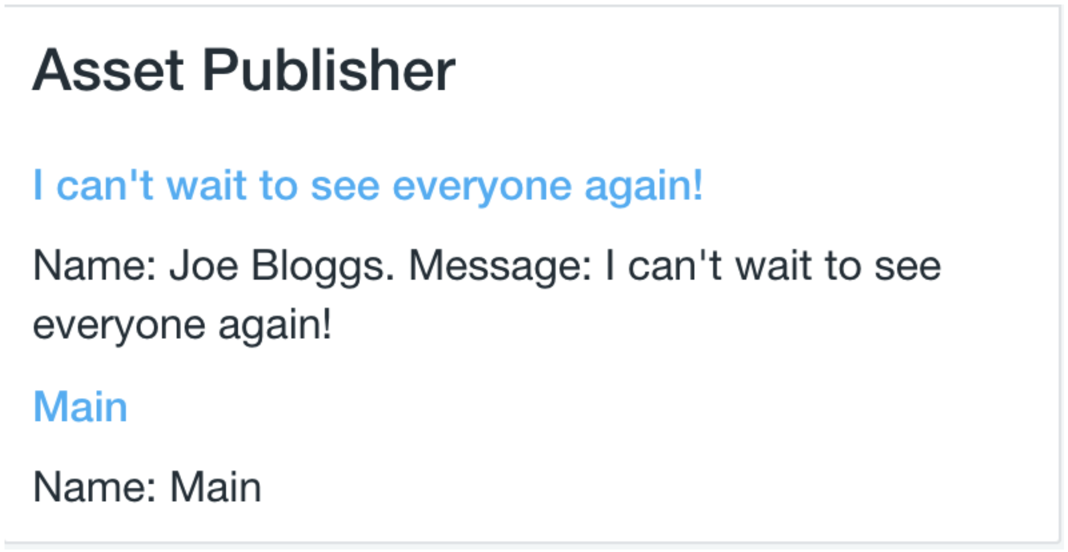
Exporting the Asset Package

The container needs to make the asset renderers and their factories available to Liferay DXP when it needs them. To do this, you must export the package.

Open the `guestbook-service` module's `bnd.bnd` file and add the asset package to the `Export-Package` declaration. When you're finished, it should look like this:

```
Export-Package: com.liferay.docs.guestbook.asset,\
                com.liferay.docs.guestbook.service.permission,\
                com.liferay.docs.guestbook.search
```

Now your guestbook project's entities are fully asset-enabled. To test the functionality, add the Asset Publisher portlet to a page and add a few guestbooks and guestbook entries. Edit a few of them, too. Then, check the Asset Publisher portlet. The Asset Publisher, by default, dynamically displays assets of any kind from the current site.


Asset Publisher

I can't wait to see everyone again!

Name: Joe Bloggs. Message: I can't wait to see everyone again!

Main

Name: Main

Figure 32.1: After you've implemented and registered your asset renderers for your custom entities, the Asset Publisher can display your entities.

Confirm that the Asset Publisher displays the guestbooks and guestbook entries that you added.

Great! In the next section, you'll update your portlets' user interfaces to use several features of Liferay DXP's asset framework: comments, ratings, tags, categories, and related assets.

Chapter 33

Adding Asset Features to Your User Interface

```
<p>Adding Asset Features to Your UI<br>Step 1 of 5</p>
```

Now that your guestbook and guestbook entry entities are asset-enabled, you're ready to use Liferay DXP's asset functionality in your application. You'll start by implementing comments, ratings, tags, categories, and related assets for guestbooks. Then you'll do the same for guestbook entries. All the back-end support for these features is provided by Liferay DXP. Your only task is to update your applications' user interfaces to use these features.

In this section, you'll create several new JSPs that need new imports. Add the following imports to the `guestbook-web` module project's `init.jsp` file:

```
<%@ page import="java.util.Map" %>
<%@ page import="java.util.HashMap" %>

<%@ page import="com.liferay.asset.kernel.service.AssetEntryLocalServiceUtil" %>
<%@ page import="com.liferay.asset.kernel.service.AssetTagLocalServiceUtil" %>

<%@ page import="com.liferay.asset.kernel.model.AssetEntry" %>
<%@ page import="com.liferay.asset.kernel.model.AssetTag" %>

<%@ page import="com.liferay.portal.kernel.util.ListUtil" %>
```

It's simpler to add these imports now so you don't run into errors as you're working through this section.

33.1 Creating JSPs for Displaying Custom Assets in the Asset Publisher

```
<p>Adding Asset Features to Your UI<br>Step 2 of 5</p>
```

Before proceeding, you must tie up a loose end from the previous step. Remember that you implemented `getJspPath` methods in your `GuestbookAssetRenderer` and `EntryAssetRenderer` classes. These methods return paths to the JSPs the Asset Publisher uses to display the assets' full content. The `getJspPath` method of `GuestbookAssetRenderer` returns `"/asset/guestbook/full_content.jsp"`, and the `getJspPath` method of `EntryAssetRenderer` returns `"/asset/entry/full_content.jsp"`. It's time to create these JSPs.

Follow these steps:

1. In the `guestbook-web` module project, create a new folder called `asset` under the `resources/META-INF/resources` folder. Add two folders to this new folder: `entry` and `guestbook`.

2. Create a new file called `full_content.jsp` in the `/asset/guestbook` folder. This JSP displays a guestbook asset's full content. Add the following code to this file:

   ```
   <%@include file="../../init.jsp"%>

   <%
   Guestbook guestbook = (Guestbook)request.getAttribute("gb_guestbook");

   guestbook = guestbook.toEscapedModel();
   %>

   <dl>
           <dt>Name</dt>
           <dd><%= guestbook.getName() %></dd>
   </dl>
   ```

   This JSP grabs the guestbook object from the request and displays the guestbook's name. In `GuestbookAssetRenderer`, the `getJspPath` method used the following to add the `gb_guestbook` request attribute:

   ```
   request.setAttribute("gb_guestbook", _guestbook);
   ```

   The guestbook's `toEscapedModel` method belongs to the `GuestbookModelImpl` class, which was generated by Service Builder. This method returns a *safe* guestbook object (a guestbook in which each field is HTML-escaped). Calling `guestbook = guestbook.toEscapedModel()` before displaying the guestbook name ensures that your JSP won't display malicious code that's masquerading as a guestbook name.

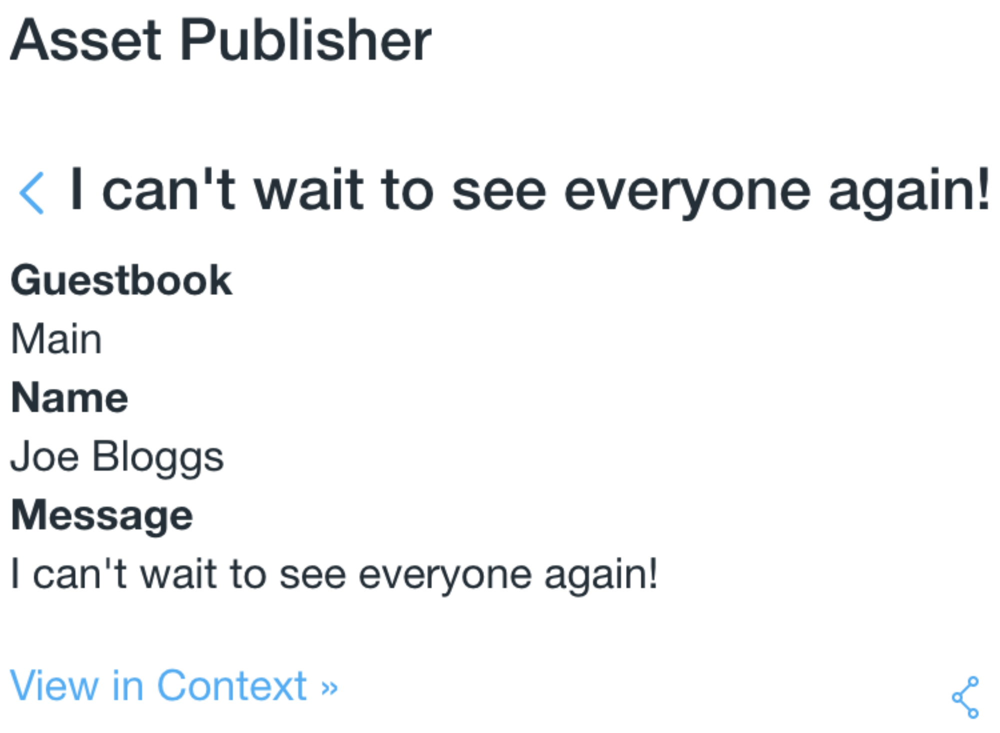
3. Next, in the `/asset/entry` folder, create a `full_content.jsp` for displaying a guestbook entry asset's full content. Add the following code to this file:

   ```
   <%@include file="../../init.jsp"%>

   <%
   Entry entry = (Entry)request.getAttribute("gb_entry");

   entry = entry.toEscapedModel();
   %>

   <dl>
           <dt>Guestbook</dt>
           <dd><%= GuestbookLocalServiceUtil.getGuestbook(entry.getGuestbookId()).getName() %></dd>
           <dt>Name</dt>
           <dd><%= entry.getName() %></dd>
           <dt>Message</dt>
           <dd><%= entry.getMessage() %></dd>
   </dl>
   ```

This JSP is almost as simple as the one for guestbooks. The only difference is that you're displaying three fields of the guestbook entry entity as opposed to one field of the guestbook entity.

Test your new JSPs by clicking a guestbook's or guestbook entry's title in the Asset Publisher. The Asset Publisher renders `full_content.jsp`:

By default, when displaying an asset's full view, the Asset Publisher displays additional links for Twitter, Facebook, and Google Plus. These links publicize your asset on social media. The *Back* icon and the *View in Context* link return you to the Asset Publisher's default view.


Asset Publisher

< I can't wait to see everyone again!

Guestbook
Main
Name
Joe Bloggs
Message
I can't wait to see everyone again!

Figure 33.1: When you click the title for a guestbook or guestbook entry in the Asset Publisher, your `full_content.jsp` should be displayed.

33.2 Enabling Tags, Categories, and Related Assets for Guestbooks

`<p>Adding Asset Features to Your UI<br>Step 3 of 5</p>`

Since you already asset-enabled guestbooks at the service layer, guestbook entities can now use Liferay DXP's back-end support for tags and categories. All that's left is to enable tags and categories in the UI. In this step, you'll update the Guestbook Admin portlet's `edit_guestbook.jsp` so admins can add, edit, or remove tags and categories when adding or updating a guestbook.

Enabling Asset Features

Follow these steps:

1. In the `guestbook-web` module's `/guestbookadminportlet/edit_guestbook.jsp`, add the tags `<liferay-ui:asset-categories-error />` and `<liferay-ui:asset-tags-error/>` to the `aui:form` below the closing `</aui:fieldset>` tag:

   ```
   <liferay-ui:asset-categories-error />
   <liferay-ui:asset-tags-error />
   ```

   These tags display error messages if an error occurs with the tags or categories submitted in the form.

2. Below the error tags, add a `<liferay-ui:panel>` tag with the following attributes:

```
<liferay-ui:panel defaultState="closed" extended="<%= false %>"
  id="guestbookCategorizationPanel" persistState="<%= true %>"
  title="categorization">

</liferay-ui:panel>
```

The `<liferay-ui:panel>` tag generates a collapsible section.

3. Add input fields for tags and categories inside the panel section you just created. Specify the `assetCategories` and `assetTags` types for the `<aui:input />` tags to tell Liferay DXP that these input tags represent asset categories and asset tags. You can group related input fields together with an `<aui:fieldset>` tag. Liferay DXP shows the appropriate selectors for tags and categories and displays the tags and categories that have already been added to the guestbook:

```
<aui:fieldset>
  <aui:input name="categories" type="assetCategories" />

  <aui:input name="tags" type="assetTags" />
</aui:fieldset>
```

4. Add a second `<liferay-ui:panel>` tag under the existing one. In this new tag, add an `<aui:fieldset>` tag containing a `<liferay-ui:asset-links>` tag. To display the correct asset links (the selected guestbook's related assets), set the `className` and `classPK` attributes:

```
<liferay-ui:panel defaultState="closed" extended="<%= false %>"
  id="guestbookAssetLinksPanel" persistState="<%= true %>"
  title="related-assets">
  <aui:fieldset>
    <liferay-ui:input-asset-links
      className="<%= Guestbook.class.getName() %>"
      classPK="<%= guestbookId %>" />
  </aui:fieldset>
</liferay-ui:panel>
```

Test the updated `edit_guestbook.jsp` page by navigating to the Guestbook Admin portlet in the Control Panel and clicking *Add Guestbook*. You'll see a field for adding tags and a selector for selecting related assets.

Don't do anything with these fields yet, because you're not done implementing assets. Next, you'll enable tags and categories for guestbook entries.

33.3 Enabling Tags, Categories, and Related Assets for Guestbook Entries

<p>Adding Asset Features to Your UI<br>Step 4 of 5</p>

Enabling tags, categories, and related assets for guestbook entries is similar to enabling them for guestbooks. It's so similar, you can refer back to the previous step for a detailed explanation.

Open your `guestbook-web` module's `guestbookwebportlet/edit_entry.jsp` file. Replace its content with the following code:

```
<%@ include file="../init.jsp" %>

<%
long entryId = ParamUtil.getLong(renderRequest, "entryId");
```

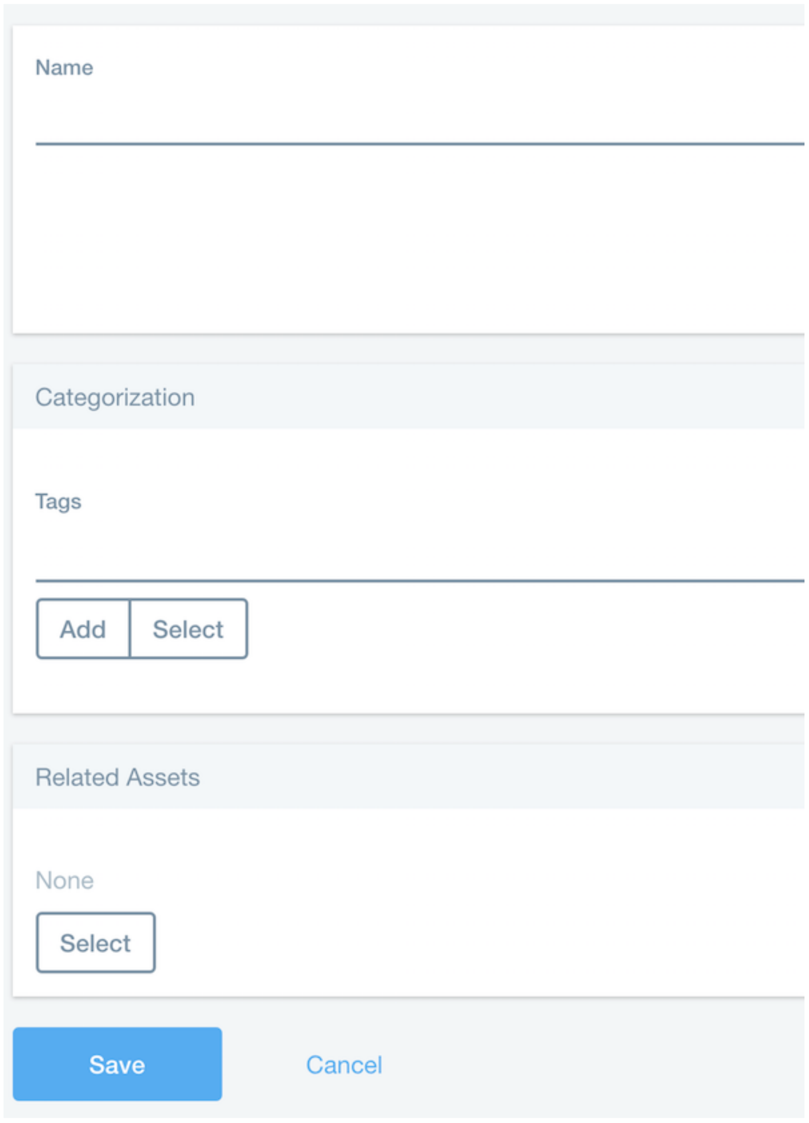


Figure 33.2: Once you've updated your Guestbook Admin portlet's `edit_guestbook.jsp` page, you'll see forms for adding tags and selecting related assets.

```
Entry entry = null;

if (entryId > 0) {
    entry = EntryLocalServiceUtil.getEntry(entryId);
}

long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");
%>

<portlet:renderURL var="viewURL">
    <portlet:param
        name="mvcPath"
        value="/guestbookwebportlet/view.jsp"
    />
</portlet:renderURL>

<liferay-ui:header
    backURL="<%= viewURL.toString() %>"
    title="<%= entry == null ? "Add Entry" : entry.getName() %>"
```

```
/>

<portlet:actionURL name="addEntry" var="addEntryURL" />

<aui:form action="<%= addEntryURL %>" name="fm">
    <aui:model-context bean="<%= entry %>" model="<%= Entry.class %>" />

        <aui:fieldset>
            <aui:input name="name" />

            <aui:input name="email" />

            <aui:input name="message" />

            <aui:input name="entryId" type="hidden" />

            <aui:input name="guestbookId" type="hidden"
            value=
            "<%= entry == null ? guestbookId : entry.getGuestbookId() %>" />
        </aui:fieldset>

<liferay-ui:asset-categories-error />
                    <liferay-ui:asset-tags-error />
                    <liferay-ui:panel defaultState="closed"
                    extended="<%= false %>" id="entryCategorizationPanel"
                    persistState="<%= true %>" title="categorization">
                            <aui:fieldset>
                                    <aui:input name="categories"
                                    type="assetCategories" />

                                    <aui:input name="tags" type="assetTags" />
                            </aui:fieldset>
                    </liferay-ui:panel>

                    <liferay-ui:panel defaultState="closed"
                    extended="<%= false %>" id="entryAssetLinksPanel"
                    persistState="<%= true %>" title="related-assets">
                            <aui:fieldset>
                                    <liferay-ui:input-asset-links
                                            className=
                                            "<%= Entry.class.getName() %>"
                                            classPK="<%= entryId %>"
                                    />
                            </aui:fieldset>
                    </liferay-ui:panel>

    <aui:button-row>
        <aui:button type="submit" />

        <aui:button onClick="<%= viewURL.toString() %>" type="cancel" />
    </aui:button-row>
</aui:form>
```

Test your JSP by using the Guestbook portlet to add and update Guestbook entries. Try adding and removing tags, categories, and related assets. All these operations should work.

Well done! Next, you'll enable comments and ratings for guestbook entries.

33.4 Enabling Comments and Ratings for Guestbook Entries

```
<p>Adding Asset Features to Your UI<br>Step 5 of 5</p>
```

Liferay DXP's asset framework lets users comment on and rate assets. As with tags, categories, and related assets, you must update the user interface to expose these features. It's best to separate the page where users comment on and rate assets from the page where users edit assets. You shouldn't have to edit an entry to comment on it; that not only makes no sense, it's a security problem. Comments and ratings should be added in a view mode only.

Follow these steps to enable comments and ratings on guestbook entries:

1. Create a new file called `view_entry.jsp` in your `guestbook-web` module project's `/guestbookwebportlet` folder.

2. Add a Java scriptlet to the file you just created. In this scriptlet, use an `entryId` request attribute to get an entry object. For security reasons, convert this object to an escaped model as discussed in the earlier step Creating JSPs for Displaying Customs Assets in the Asset Publisher:

```
<%@ include file="../init.jsp"%>

<%
  long entryId = ParamUtil.getLong(renderRequest, "entryId");

  long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");

  Entry entry = null;

  if (entryId > 0) {
    entry = EntryLocalServiceUtil.getEntry(entryId);

    entryId = entry.getEntryId();
  }

  entry = EntryLocalServiceUtil.getEntry(entryId);
  entry = entry.toEscapedModel();

  AssetEntry assetEntry =
  AssetEntryLocalServiceUtil.getEntry(Entry.class.getName(),
  entry.getEntryId());
```

3. Next, update the breadcrumb entry with the the current entry's name:

```
String currentURL = PortalUtil.getCurrentURL(request);
PortalUtil.addPortletBreadcrumbEntry(request, entry.getMessage(),
currentURL);
```

4. At the end of the scriptlet, add the names of the current entry's existing asset tags as keywords to the portal page. These tag names appear in a `<meta content="[tag names here]" lang="en-US" name="keywords" />` element in your portal page's `<head>` section. These keywords can help search engines find and index your page:

```
  PortalUtil.setPageSubtitle(entry.getMessage(), request);
  PortalUtil.setPageDescription(entry.getMessage(), request);

  List<AssetTag> assetTags =
  AssetTagLocalServiceUtil.getTags(Entry.class.getName(),
  entry.getEntryId());
  PortalUtil.setPageKeywords(ListUtil.toString(assetTags, "name"),
  request);
%>
```

5. After the scriptlet, specify the URLs for the page and back link:

```
<liferay-portlet:renderURL varImpl="viewEntryURL">
  <portlet:param name="mvcPath"
    value="/guestbookwebportlet/view_entry.jsp" />
  <portlet:param name="entryId" value="<%=String.valueOf(entryId)%>" />
</liferay-portlet:renderURL>

<liferay-portlet:renderURL varImpl="viewURL">
  <portlet:param name="mvcPath"
    value="/guestbookwebportlet/view.jsp" />
</liferay-portlet:renderURL>

<liferay-ui:header backURL="<%=viewURL.toString()%>"
  title="<%=entry.getName()%>"
/>
```

6. Next, define the page's main content. Display the guestbook's name, and the entry's name and message with the `<dl>`, `<dt>`, and `<dd>` tags:

```
<dl>
  <dt>Guestbook</dt>
  <dd><%=GuestbookLocalServiceUtil.getGuestbook(entry.getGuestbookId()).getName()%></dd>
  <dt>Name</dt>
  <dd><%=entry.getName()%></dd>
  <dt>Message</dt>
  <dd><%=entry.getMessage()%></dd>
</dl>
```

   This is the same way you defined the page's main content in `/guestbookwebportlet/full_content.jsp`.

7. Next, use a `<liferay-ui:panel-container>` tag to create a panel container. Inside this tag, use a `<liferay-ui:panel>` tag to create a panel to display the comments and ratings components:

```
<liferay-ui:panel-container extended="<%=false%>"
  id="guestbookCollaborationPanelContainer" persistState="<%=true%>">
  <liferay-ui:panel collapsible="<%=true%>" extended="<%=true%>"
    id="guestbookCollaborationPanel" persistState="<%=true%>"
    title="Collaboration">
```

8. Add the ratings and comments components via the `<liferay-ui:ratings>` and `<liferay-ui:discussion>` tags, respectively. The latter tag needs an action URL (in this case, `invokeTaglibDiscussion`) for its `formAction` attribute. The action URL adds the comment after the user enters a comment and clicks *Reply*:

```
    <liferay-ui:ratings className="<%=Entry.class.getName()%>"
      classPK="<%=entry.getEntryId()%>" type="stars" />

    <br />

    <portlet:actionURL name="invokeTaglibDiscussion"
    var="discussionURL" />

    <liferay-ui:discussion className="<%=Entry.class.getName()%>"
      classPK="<%=entry.getEntryId()%>"
      formAction="<%=discussionURL%>" formName="fm2"
      ratingsEnabled="<%=true%>" redirect="<%=currentURL%>"
      userId="<%=entry.getUserId()%>" />

  </liferay-ui:panel>
</liferay-ui:panel-container>
```

9. To restrict comments and ratings access to logged-in users, wrap the whole panel container in a `<c:if>` tag that tests the expression `themeDisplay.isSignedIn()`:

   ```
   <c:if test="<%= themeDisplay.isSignedIn() %>">
       ... your panel container ...
   </c:if>
   ```

   Make sure you add the closing `</c:if>` tag after the closing `</liferay-ui:panel-container>` tag.

```
**Note:** Discussions (comments) are implemented as message board messages
in Liferay DXP. In the `MBMessage` table, there's a `classPK` column. This
`classPK` represents the `entryId` of the guestbook entry the comment
belongs to. Ratings are stored in the `RatingsEntry` table. Similarly, the
`RatingsEntry` table contains a `classPK` column. This `classPK` represents
the `entryId` of the guestbook entry the rating belongs to. Using a
`classPK` foreign key in one table to represent the primary key of another
table is a common pattern that's used throughout Liferay DXP.
```

Next, you'll update the guestbook actions to use the new view.

Updating the Entry Actions JSP

Nothing links to your `view_entry.jsp` page—it's currently orphaned. Fix this by adding the *View* option to the Actions Menu. Open the `/guestbookwebportlet/entry_actions.jsp` and find the following line:

```
<liferay-ui:icon-menu>
```

Add the following lines below it:

```
<portlet:renderURL var="viewEntryURL">
  <portlet:param name="entryId"
    value="<%= String.valueOf(entry.getEntryId()) %>" />
  <portlet:param name="mvcPath"
    value="/guestbookwebportlet/view_entry.jsp" />
</portlet:renderURL>

<liferay-ui:icon message="View" url="<%= viewEntryURL.toString() %>" />
```

Here, you create a URL that points to `view_entry.jsp`. Test this link by selecting the *View* option in a guestbook entry's Actions Menu. Then test that comments and ratings work as expected.

Excellent! You've asset-enabled the guestbook and guestbook entry entities, and enabled tags, categories, and related assets for both entities. You've also enabled comments and ratings for guestbook entry entities! Great job!

Your next task is to generate web services. This makes it possible to write other clients (such as mobile applications) for the Guestbook application.

Chapter 34

Generating Web Services

Assets opened the door to support many features of Liferay DXP's development framework. There's more to cover, but the Guestbook app's back-end is now mature enough to widen its appeal.

Right now, the Guestbook app's back-end services can only be accessed by modules in the same OSGi container. If you want to read or post Guestbook entries, you have to write and deploy native code on the server. This is fine for some, but others want more: web clients on a different platform; standalone applications using Electron or some other framework; or mobile apps for Android or iOS. Web services power all of these.

You now have an application with back-end services and a front-end web client running in the same container. Web services make it possible to have multiple front-end clients on multiple platforms that access the same back-end. This makes it possible to build *headless* applications on Liferay DXP, with multiple front-ends elsewhere.

Next, you'll use Service Builder to create the Guestbook app's web services. When you finish, authorized clients can then consume these web services.

Ready to start?

Let's Go!

34.1 Creating Remote Services with Service Builder

`<p>Creating Remote Services<br>Step 1 of 1</p>`

Earlier, you used Service Builder to generate the Guestbook's model, persistence, and service layers. Services generated by Service Builder can come in two flavors: local and remote. The local services you already used can only be invoked locally from the same OSGi container. Remote services can be invoked by any application with permission to access your server via the web. Remote services are published as JSON or SOAP.

For more information, click here to see the Service Builder Web Services section of tutorials.

Creating web services for the Guestbook application takes two steps:

1. Generate the web services with Service Builder.

2. Expose the services you want and wrap them in permission checks.

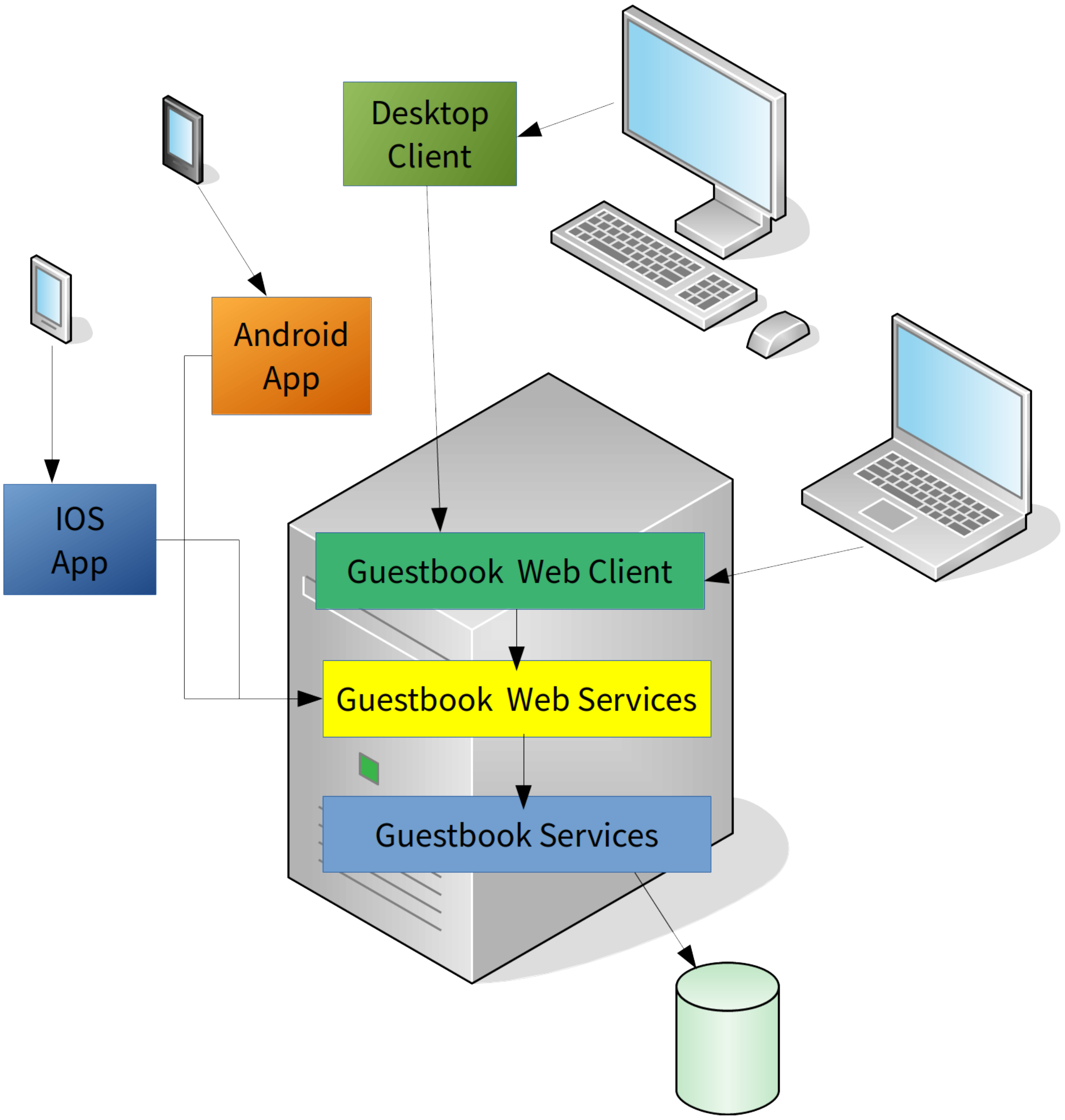


Figure 34.1: Liferay DXP makes it easy to write multi-client applications.

There's a level of security in the assumption that local services can only be called by other services in the container. For example, the web app does permission and validation checks before calling services. To access these services, developers must be able to deploy their modules on the server. You don't have these assurances when you expose services to the web, so you must check for permission before calling a service.

But first you must tell Service Builder to generate web services. Follow these steps:

1. Open `service.xml` from the `guestbook-service` module. Find the tags for the `Guestbook` and `Entry` entities:

   ```
   <entity name="Guestbook" local-service="true" uuid="true">

   <entity name="Entry" local-service="true" uuid="true">
   ```

2. As described in the `service.xml` DTD, `local-service` defaults to `false` and `remote-service` defaults to `true`. It helps other developers who read your code to specify what services are generated. Therefore, add `remote-service="true"` to the entity tags of the `Guestbook` and `Entry` entities:

   ```
   <entity name="Guestbook" local-service="true" remote-service="true" uuid="true">

   <entity name="Entry" local-service="true" remote-service="true" uuid="true">
   ```

3. In the *Gradle Tasks* window on the right-hand side of Liferay @ide@, expand the service module's build folder. Run Service Builder by double-clicking *buildService*. When Service Builder finishes, refresh the `guestbook-api` and `guestbook-service` modules in the Project Explorer.

You may be interested to know that Service Builder did absolutely nothing. Since remote services are generated by default, you've always had their stubs in your project. All you did was make it explicit in the code.

By implementing local services, you separated concerns. Local services can assume things like permission checks have already been done before they're called. This separates the business logic from the permissions logic. If instead you implemented everything in the remote services, this separation wouldn't exist.

You may see code from other developers, however, who didn't implement the local service, and instead elected to place all their business and permission logic in the remote service. This works, but makes the code less readable. With the concerns separated, a business logic bug is contained in the local service, and a permissions bug is contained in the remote service.

To expose remote services, you'll implement methods in the `-ServiceImpl` classes instead of the `-LocalServiceImpl` classes. Since the primary concern is permissions, however, first create a helper class to hold the permissions:

1. In the `src/main/java` folder, create the new package `com.liferay.docs.guestbook.util`. In this new package, create this `ActionKeys` class:

   ```
   package com.liferay.docs.guestbook.util;

   public class ActionKeys extends
                   com.liferay.portal.kernel.security.permission.ActionKeys {

           public static final String ADD_ENTRY = "ADD_ENTRY";
           public static final String ADD_GUESTBOOK = "ADD_GUESTBOOK";
   }
   ```

The ADD_ENTRY and ADD_GUESTBOOK strings reference the permissions defined in the guestbook-service module's docroot/WEB-INF/src/resource-actions/default.xml file earlier in this Learning Path. It's a best practice to create strings to refer to permissions in a class called ActionKeys that extends com.liferay.portal.kernel.security.permission.ActionKeys. The parent ActionKeys contains strings that are used to refer to portal permissions. These include strings for common permissions such as VIEW, UPDATE, DELETE, and so on.

2. Add the following methods to the GuestbookServiceImpl class; then organize the imports by selecting *Source → Organize Imports*:

```
public Guestbook addGuestbook(long userId, String name,
    ServiceContext serviceContext) throws SystemException,
    PortalException {

    return guestbookLocalService.addGuestbook(userId, name, serviceContext);
}

public Guestbook deleteGuestbook(long guestbookId,
    ServiceContext serviceContext) throws PortalException,
    SystemException {

    return guestbookLocalService.deleteGuestbook(guestbookId, serviceContext);
}

public List<Guestbook> getGuestbooks(long groupId) throws SystemException {
        return guestbookLocalService.getGuestbooks(groupId);
}

public List<Guestbook> getGuestbooks(long groupId, int start, int end)
                throws SystemException {
        return guestbookLocalService.getGuestbooks(groupId, start, end);
}

public int getGuestbooksCount(long groupId) throws SystemException {
        return guestbookLocalService.getGuestbooksCount();
}

public Guestbook updateGuestbook(long userId, long guestbookId,
    String name, ServiceContext serviceContext) throws PortalException,
    SystemException {

    return guestbookLocalService.updateGuestbook(userId, guestbookId,
        name, serviceContext);
}
```

These are stub remote service methods that expose each guestbook local service method. For now, the remote service method implementations call the local service implementations. Later, you'll add permission checks to these methods, to wrap them in the same permissions you created in the UI. Service calls have no UI, so you must check for permission to access them. For now, you're exposing the services to confirm they work and are accessible.

3. Add the following methods to the EntryServiceImpl class, then organize the imports as you did in step 2:

```
public Entry addEntry(long userId, long guestbookId, String name,
        String email, String message, ServiceContext serviceContext)
        throws PortalException, SystemException {

    return entryLocalService.addEntry(userId, guestbookId, name, email,
             message, serviceContext);
}

public Entry deleteEntry(long entryId, ServiceContext serviceContext)
                throws PortalException, SystemException {

    return entryLocalService.deleteEntry(entryId, serviceContext);
}

public List<Entry> getEntries(long groupId, long guestbookId)
                throws SystemException {

    return entryLocalService.getEntries(groupId, guestbookId);
}

public List<Entry> getEntries(long groupId, long guestbookId, int start,
                int end) throws SystemException {

    return entryLocalService.getEntries(groupId, guestbookId, start, end);
}

public int getEntriesCount(long groupId, long guestbookId)
                throws SystemException {

    return entryLocalService.getEntriesCount(groupId, guestbookId);
}

public Entry updateEntry(long userId, long guestbookId, long entryId,
                String name, String email, String message,
                ServiceContext serviceContext) throws PortalException,
                SystemException {

    return entryLocalService.updateEntry(userId, guestbookId, entryId,
                        name, email, message, serviceContext);
}
```

Like you did for guestbooks, you've now created method stubs for guestbook entries. Each method implemented here exposes a service to the web. You'll add permission checks in the next section.

4. Run Service Builder and refresh the API and service modules. Then redeploy the `guestbook-*` modules.

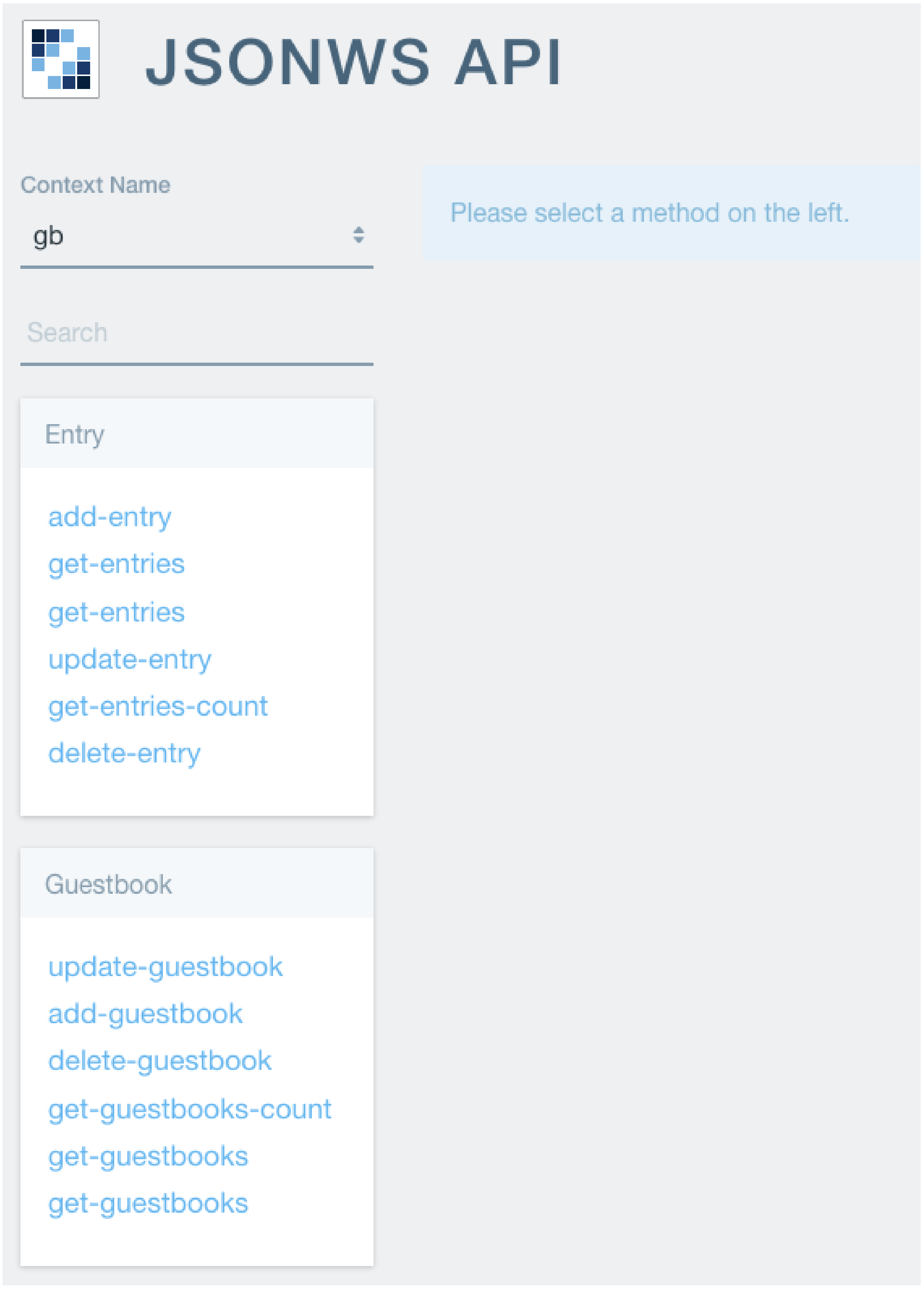
First, make sure you're logged in as a user that can read guestbooks. Navigate to Liferay DXP's JSONWS page (`http://[host name]:[port number]/api/jsonws`) and click the *Context Name* selector. The Guestbook app's context, `gb`, appears as an option. Select it and confirm that your remote service methods appear in the list.

To test that your remote services work, choose a method to invoke. Pick a simple method that doesn't require a Service Context parameter, like `getGuestbooksCount(long groupId)`. To find the appropriate `groupId` (the ID of the site containing the Guestbook app), navigate to that site in your browser and select *Configuration → Site Settings* from the Site Menu on the left. The site ID is listed at the top of the Site Settings page. Now return to the JSONWS page and enter the site ID into the group ID field and click *Invoke*. Confirm that the correct number of guestbooks is returned. Great! Your remote services work.

Next, you'll build a WSDD (Web Service Deployment Descriptor) document for your remote services to make them available via SOAP (Simple Object Access Protocol).

Follow these steps to do so:

1. In your Liferay workspace's `settings.gradle` file, add imports for `ServiceBuilderPlugin` and `WSDDBuilderPlugin` before the buildscript block. Then add the `gradle.beforeProject` closure at the bottom of the file:



JSONWS API

Context Name

gb

Search

Please select a method on the left.

Entry

add-entry
get-entries
get-entries
update-entry
get-entries-count
delete-entry

Guestbook

update-guestbook
add-guestbook
delete-guestbook
get-guestbooks-count
get-guestbooks
get-guestbooks

Figure 34.2: After you've added remote service methods to your project's *`ServiceImpl` classes, run Service Builder and redeploy your modules. Then check that your remote services are accessible.

```
import com.liferay.gradle.plugins.service.builder.ServiceBuilderPlugin
import com.liferay.gradle.plugins.wsdd.builder.WSDDBuilderPlugin

...

gradle.beforeProject {
    project ->

    project.plugins.withType(ServiceBuilderPlugin) {
        project.apply plugin: WSDDBuilderPlugin
    }
}
```

   Refresh your workspace's Gradle files: right click `settings.gradle` in the Project Explorer and select *Gradle → Refresh Gradle Project*.

2. In the *Gradle Tasks* window on the right-hand side of Liferay @ide@, expand the service module's *build* folder. Build the WSDD by double-clicking *buildWSDD*. If `buildWSDD` is missing, shut down your server and then restart Liferay @ide@. The `buildWSDD` command appears as described.

   The WSDD builder generates a WSDD JAR file in the `guestbook-service` module's `build/libs` folder. Because this folder isn't visible in @ide@, you must access it from the file system. The project's modules are in the Eclipse workspace on the file system. Here's the full file path to the WSDD JAR in your Eclipse workspace:

```
com-liferay-docs-guestbook/modules/guestbook/guestbook-service/build/libs/com.liferay.docs.guestbook.service-wsdd-1.0.0.jar
```

   If this file is missing, run `buildWSDD` again to generate it.

3. Deploy the WSDD JAR file to Liferay DXP, which is in the Liferay Workspace's `bundles` folder. To do this, copy and paste the WSDD JAR file into this folder in your Eclipse workspace on your file system:

```
com-liferay-docs-guestbook/bundles/deploy
```

   Return to Liferay @ide@ and check the console to make sure deployment completes successfully.

4. Go to `http://[host name]:[port number]/o/com.liferay.docs.guestbook.service/api/axis` in your browser to view the Guestbook app's SOAP web services. If you're running Liferay DXP locally on port 8080, this is http://localhost:8080/o/com.liferay.docs.guestbook.service/api/axis.

   This page contains links to the WSDL (Web Services Description Language) documents for the Guestbook and Entry remote service methods. WSDL files describe details about the remote service methods, including the type of data these methods require.

If you want to make your app's services available for remote invocation via SOAP, generating WSDD and WSDL files is required. For example, the Liferay Mobile SDK relies on the WSDD and WSDL to discover your Liferay DXP app's remote services. For the Liferay Mobile SDK to create a mobile client that can access your Liferay DXP app's web services, you must therefore generate a WSDD and WSDL for your app.

Next, you'll learn how to secure your web services. Unless you secure your web services by implementing permission checks, any user can add, update, or delete guestbooks or guestbook entries, and you certainly don't want that.

Chapter 35

Implementing Permission Checks

Now that your guestbook and guestbook entry web services exist, you must implement permission checks for them. Implementing permission checks for a web service ensures that only users with the correct permissions can invoke the web service. To implement permission checks in your remote services, you'll use the `GuestbookModelPermission`, `GuestbookPermission`, and `EntryPermission` helper classes that you created earlier. These classes provide helper methods for checking permissions. The helper methods in `GuestbookModelPermission` check top-level model permissions. For example, you can use `GuestbookModelPermission`'s helper methods to check if a user can add a new guestbook or guestbook entry. If, on the other hand, you must check if a user can update or delete an existing guestbook or guestbook entry, you'll use `GuestbookPermission` or `EntryPermission`.

Once you've secured your remote services with permission checks, you'll update your portlet classes to call remote services instead of local services. This prevents attackers from trying to bypass your app's UI by playing with URL parameters to access sensitive portions of your app.

Let's Go!

35.1 Implementing Permission Checks at the Service Layer

```
<p>Implementing Permission Checks<br>Step 1 of 2</p>
```

First, you'll add permission checks to `GuestbookServiceImpl`:

1. In `GuestbookServiceImpl`, replace the `addGuestbook`, `deleteGuestbook`, and `updateGuestbook` methods with these versions that contain the permission checks:

```
public Guestbook addGuestbook(long userId, String name,
            ServiceContext serviceContext) throws SystemException,
            PortalException {

        GuestbookModelPermission.check(getPermissionChecker(),
                    serviceContext.getScopeGroupId(), ActionKeys.ADD_GUESTBOOK);

        return guestbookLocalService.addGuestbook(userId, name, serviceContext);
}

public Guestbook deleteGuestbook(long guestbookId,
            ServiceContext serviceContext) throws PortalException,
            SystemException {
```

```
        GuestbookPermission.check(getPermissionChecker(), guestbookId,
                        ActionKeys.DELETE);

        return guestbookLocalService.deleteGuestbook(guestbookId, serviceContext);
}

public Guestbook updateGuestbook(long userId, long guestbookId,
                String name, ServiceContext serviceContext) throws PortalException,
                SystemException {

        GuestbookPermission.check(getPermissionChecker(), guestbookId,
                        ActionKeys.UPDATE);

        return guestbookLocalService.updateGuestbook(userId, guestbookId, name,
                        serviceContext);
}
```

Organize imports to add the imports for the *Permissions and ActionKeys classes.

These methods add permission checks to the remote service methods by calling the check helper methods in GuestbookModelPermission and GuestbookPermission. Remember that these methods throw exceptions, so if the user doesn't have permission, processing stops at the permission check. The GuestbookModelPermission.check method takes three parameters:

- a PermissionChecker object
- a groupId
- an actionId string

The GuestbookModelPermission.check and EntryModelPermission.check methods take three parameters:

- a PermissionChecker object
- an entity ID (either guestbookId or entryId)
- an actionId string

BaseServiceImpl contains a getPermissionChecker method that returns a PermissionChecker object. This is accessible since GuestbookServiceImpl extends GuestbookServiceBaseImpl, which extends BaseServiceImpl. The serviceContext method getScopeGroupId returns a groupId. The actionId string comes from your ActionKeys class. Using an ActionKeys field is less error prone than than manually typing the string's name every time you want to check a permission. Using an ActionKeys string also avoids creating a duplicate string.

2. Open EntryServiceImpl and replace the addEntry, deleteEntry, and updateEntry methods with ones that contain permission checks:

```
public Entry addEntry(long userId, long guestbookId, String name,
                String email, String message, ServiceContext serviceContext)
                throws PortalException, SystemException {

        GuestbookModelPermission.check(getPermissionChecker(),
                        serviceContext.getScopeGroupId(), ActionKeys.ADD_ENTRY);

        return entryLocalService.addEntry(userId, guestbookId, name, email,
                        message, serviceContext);
}
```

```
public Entry deleteEntry(long entryId, ServiceContext serviceContext)
                throws PortalException, SystemException {

        EntryPermission.check(getPermissionChecker(), entryId, ActionKeys.DELETE);

        return entryLocalService.deleteEntry(entryId, serviceContext);
}

public Entry updateEntry(long userId, long guestbookId, long entryId,
                String name, String email, String message,
                ServiceContext serviceContext) throws PortalException,
                SystemException {

        EntryPermission.check(getPermissionChecker(), entryId, ActionKeys.UPDATE);

        return entryLocalService.updateEntry(userId, guestbookId, entryId,
                        name, email, message, serviceContext);
}
```

As in step 1, organize imports to add the imports for the *Permissions and ActionKeys classes.

The permission checks in these methods work the same as those in GuestbookServiceImpl. For addEntry, you use GuestbookModelPermission.check for the permission check, since adding a guestbook entry is a top level model action. For deleteEntry and updateEntry, you use EntryPermission.check since these operations each require a specific permission on a specific entity.

3. Open GuestbookServiceImpl and replace both getGuestbooks methods (this method is overloaded) and the getGuestbookCount method with these:

```
public List<Guestbook> getGuestbooks(long groupId) throws SystemException {
        return guestbookPersistence.filterFindByGroupId(groupId);
}

public List<Guestbook> getGuestbooks(long groupId, int start, int end)
                throws SystemException {

        return guestbookPersistence.filterFindByGroupId(groupId, start, end);
}

public int getGuestbooksCount(long groupId) throws SystemException {
        return guestbookPersistence.filterCountByGroupId(groupId);
}
```

4. Open EntryServiceImpl and replace the getEntries methods (this method is overloaded) and the getGuestbookCount method with these:

```
public List<Entry> getEntries(long groupId, long guestbookId)
                throws SystemException {

        return entryPersistence.filterFindByG_G(groupId, guestbookId);
}

public List<Entry> getEntries(long groupId, long guestbookId, int start,
                int end) throws SystemException {

        return entryPersistence.filterFindByG_G(groupId, guestbookId, start,
                        end);
}

public int getEntriesCount(long groupId, long guestbookId)
                throws SystemException {

        return entryPersistence.filterCountByG_G(groupId, guestbookId);
}
```

5. Run Service Builder and refresh the API and service modules.

All remote service methods should include permission checks. In steps 1 and 2, you directly invoked permission checks for the remote service methods addGuestbook, deleteGuestbook, updateGuestbook, addEntry, deleteEntry, and updateEntry by using the check methods of the permissions utility classes: GuestbookModelPermission, GuestbookPermission, and EntryPermission. In steps 3 and 4, you indirectly invoked permission checks for the remote service methods getGuestbooks, getGuestbooksCount, getEntries, and getEntriesCount by calling the filterFindBy* and *filterCountBy* methods of GuestbookPersistenceImpl and EntryPersistenceImpl. The filterFindBy* and *filterCountBy* methods are generated by Service Builder if the following conditions are met:

- The entity has a simple primitive primary key
- The entity has permission checks registered in an XML file in your project's docroot/WEB-INF/src/resource-actions directory
- The entity has userId and groupId fields
- The finder method has a groupId argument in its method signature

Since it would be a very expensive operation to retrieve a large list of guestbook or guestbook entry entities and run permission checks on each one, Service Builder generates the filterFindBy* and filterCountBy* helper methods in the persistence layer to handle permission checks. The permission checking of these helper methods is done in the database, resulting in a less expensive operation. The filterFindBy* and filterCountBy* methods work just like the ordinary findBy* and countBy* methods in the *PersistenceImpl classes, except that the filterFindBy* and filterCountBy* methods include permission checks. Instances of the *PersistenceImpl classes are made available as Spring beans in the *ServiceImpl classes. These beans are named guestbookPersistence and entryPersistence in GuestbookServiceImpl and EntryServiceImpl, respectively.

Awesome! You're almost done. The only thing left is to secure the service calls you make in the web client.

35.2 Securing Service Calls at the Portlet Layer

<p>Implementing Permission Checks<br>Step 2 of 2</p>

Your remote services are now secure for direct use. Your web application, however, is another story–it's still calling local services. In a perfect world, that would be fine, but this isn't a perfect world.

Previously, you implemented portlet action methods such as addGuestbook, addEntry, deleteEntry, and so on in GuestbookPortlet and GuestbookAdminPortlet classes. These methods call local services. For example, in the addEntry method of GuestbookPortlet, you used the following call to add a new guestbook entry:

```
_entryLocalService.addEntry(serviceContext.getUserId(), guestbookId, userName,
    email, message, serviceContext);
```

Calling local services from an app's portlet layer isn't recommended because they don't contain permission checks. Nefarious individuals messing with URL parameters might be able to access protected areas of your app. To secure your app, only call *remote* services from the portlet layer, because that's the layer that has permission checking. Thus, to secure service calls at the portlet layer, all you have to do is replace the local service calls with remote service calls.

Note: An alternative approach to securing service calls at the portlet layer is to check permissions at the portlet layer manually. To do this, get a ThemeDisplay from the ActionRequest (ThemeDisplay themeDisplay

= (ThemeDisplay) request.getAttribute(WebKeys.THEME_DISPLAY);) and get a PermissionChecker from the ThemeDisplay (PermissionChecker permissionChecker = themeDisplay.getPermissionChecker();). If the user passes the permission check, then call the local service method. However, it's best to avoid rewriting permission checks whenever possible. For this reason, if you must wrap a service call in a permission check, implement that service method as a remote service and add the permission check to the remote service. This is the pattern Liferay DXP uses and that you have followed in this Learning Path.

Use the following steps to secure the service calls in the GuestbookPortlet class:

1. Replace the _guestbookLocalService and _entryLocalService variable declarations with these:

   ```
   private GuestbookService _guestbookService;
   private EntryService _entryService;
   ```

2. Replace the GuestbookLocalService and EntryLocalService imports with the these:

   ```
   import com.liferay.docs.guestbook.service.GuestbookService;
   import com.liferay.docs.guestbook.service.EntryService;
   ```

3. Replace all instances of _guestbookLocalService and _entryLocalService with _guestbookService and _entryService, respectively. Also make sure to replace any GuestbookLocalService and EntryLocalService method arguments with GuestbookService and EntryService, respectively.

Now follow the same steps to change guestbook services in GuestbookAdminPortlet. Note that service calls and variables for guestbook entries aren't necessary in GuestbookAdminPortlet.

To check that you haven't made a mistake in your GuestbookPortlet class, refer to the following complete class:

```
package com.liferay.docs.guestbook.portlet;

import java.io.IOException;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.Portlet;
import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

import com.liferay.docs.guestbook.model.Entry;
import com.liferay.docs.guestbook.model.Guestbook;
import com.liferay.docs.guestbook.portlet.constants.GuestbookPortletKeys;
import com.liferay.docs.guestbook.service.GuestbookService;
import com.liferay.docs.guestbook.service.EntryService;

import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet;
import com.liferay.portal.kernel.service.ServiceContext;
import com.liferay.portal.kernel.service.ServiceContextFactory;
import com.liferay.portal.kernel.servlet.SessionErrors;
import com.liferay.portal.kernel.servlet.SessionMessages;
import com.liferay.portal.kernel.util.ParamUtil;
```

```
import com.liferay.portal.kernel.util.PortalUtil;

/**
 * @author sezovr
 */
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.social",
        "com.liferay.portlet.instanceable=false",
        "com.liferay.portlet.scopeable=true",
        "javax.portlet.display-name=Guestbook",
        "javax.portlet.expiration-cache=0",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/guestbookwebportlet/view.jsp",
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user",
        "javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK,
        "javax.portlet.supports.mime-type=text/html"
    },
    service = Portlet.class
)
public class GuestbookPortlet extends MVCPortlet {

    public void addEntry(ActionRequest request, ActionResponse response)
                    throws PortalException {

                ServiceContext serviceContext = ServiceContextFactory.getInstance(
                    Entry.class.getName(), request);

                String userName = ParamUtil.getString(request, "name");
                String email = ParamUtil.getString(request, "email");
                String message = ParamUtil.getString(request, "message");
                long guestbookId = ParamUtil.getLong(request, "guestbookId");
                long entryId = ParamUtil.getLong(request, "entryId");

            if (entryId > 0) {

                try {

                    _entryService.updateEntry(
                        serviceContext.getUserId(), guestbookId, entryId, userName,
                        email, message, serviceContext);

                    SessionMessages.add(request, "entryAdded");

                    response.setRenderParameter(
                        "guestbookId", Long.toString(guestbookId));

                }
                catch (Exception e) {
                    System.out.println(e);

                    SessionErrors.add(request, e.getClass().getName());

                    PortalUtil.copyRequestParameters(request, response);

                    response.setRenderParameter(
                        "mvcPath", "/guestbookwebportlet/edit_entry.jsp");
                }

            }
            else {

                try {
                    _entryService.addEntry(
                        serviceContext.getUserId(), guestbookId, userName, email,
                        message, serviceContext);
                    SessionMessages.add(request, "entryAdded");

                    response.setRenderParameter(
                        "guestbookId", Long.toString(guestbookId));

                }
                catch (Exception e) {
                    SessionErrors.add(request, e.getClass().getName());

                    PortalUtil.copyRequestParameters(request, response);

                    response.setRenderParameter(
                        "mvcPath", "/guestbookwebportlet/edit_entry.jsp");
                }
            }
        }

    public void deleteEntry(ActionRequest request, ActionResponse response) throws PortalException {
        long entryId = ParamUtil.getLong(request, "entryId");
        long guestbookId = ParamUtil.getLong(request, "guestbookId");

        ServiceContext serviceContext = ServiceContextFactory.getInstance(
            Entry.class.getName(), request);

        try {

            response.setRenderParameter(
                "guestbookId", Long.toString(guestbookId));

            _entryService.deleteEntry(entryId, serviceContext);
            SessionMessages.add(request, "entryDeleted");

        }

        catch (Exception e) {
            Logger.getLogger(GuestbookPortlet.class.getName()).log(
                Level.SEVERE, null, e);
            SessionErrors.add(request, e.getClass().getName());


        }
    }


    @Override
    public void render(
            RenderRequest renderRequest, RenderResponse renderResponse)
        throws IOException, PortletException {

        try {
            ServiceContext serviceContext = ServiceContextFactory.getInstance(
                Guestbook.class.getName(), renderRequest);

            long groupId = serviceContext.getScopeGroupId();

            long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");

            List<Guestbook> guestbooks = _guestbookService.getGuestbooks(
                groupId);

            if (guestbooks.isEmpty()) {
                Guestbook guestbook = _guestbookService.addGuestbook(
                    serviceContext.getUserId(), "Main", serviceContext);

                guestbookId = guestbook.getGuestbookId();
            }
```

```
            if (guestbookId == 0) {
                guestbookId = guestbooks.get(0).getGuestbookId();
            }

            renderRequest.setAttribute("guestbookId", guestbookId);
        }
        catch (Exception e) {
            throw new PortletException(e);
        }

        super.render(renderRequest, renderResponse);
    }


    @Reference(unbind = "-")
    protected void setEntryService(EntryService entryService) {
        _entryService = entryService;
    }

    @Reference(unbind = "-")
    protected void setGuestbookService(GuestbookService guestbookService) {
        _guestbookService = guestbookService;
    }

    private GuestbookService _guestbookService;
    private EntryService _entryService;


}
```

To check that you haven't made a mistake in your `GuestbookAdminPortlet` class, refer to the following complete class:

```
package com.liferay.docs.guestbook.portlet;

import java.util.logging.Level;
import java.util.logging.Logger;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.Portlet;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

import com.liferay.docs.guestbook.model.Guestbook;
import com.liferay.docs.guestbook.portlet.constants.GuestbookPortletKeys;
import com.liferay.docs.guestbook.service.GuestbookService;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet;
import com.liferay.portal.kernel.service.ServiceContext;
import com.liferay.portal.kernel.service.ServiceContextFactory;
import com.liferay.portal.kernel.servlet.SessionErrors;
import com.liferay.portal.kernel.servlet.SessionMessages;
import com.liferay.portal.kernel.util.ParamUtil;

@Component(
    immediate = true,
    property = {
            "com.liferay.portlet.display-category=category.hidden",
            "com.liferay.portlet.scopeable=true",
            "javax.portlet.display-name=Guestbooks",
            "javax.portlet.expiration-cache=0",
            "javax.portlet.init-param.portlet-title-based-navigation=true",
            "javax.portlet.init-param.template-path=/",
            "javax.portlet.init-param.view-template=/guestbookadminportlet/view.jsp",
            "javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK_ADMIN,
            "javax.portlet.resource-bundle=content.Language",
            "javax.portlet.security-role-ref=administrator",
            "javax.portlet.supports.mime-type=text/html",
            "com.liferay.portlet.add-default-resource=true"
    },
    service = Portlet.class
)
public class GuestbookAdminPortlet extends MVCPortlet {

    public void addGuestbook(ActionRequest request, ActionResponse response)
            throws PortalException {

            ServiceContext serviceContext = ServiceContextFactory.getInstance(
              Guestbook.class.getName(), request);

            String name = ParamUtil.getString(request, "name");

            try {
              _guestbookService.addGuestbook(
                serviceContext.getUserId(), name, serviceContext);
              SessionMessages.add(request, "guestbookAdded");

            }
            catch (PortalException e) {

            Logger.getLogger(GuestbookAdminPortlet.class.getName()).log(
              Level.SEVERE, null, e);
            SessionErrors.add(request, e.getClass().getName());


              response.setRenderParameter(
                "mvcPath", "/guestbookadminportlet/edit_guestbook.jsp");
            }
          }

    public void updateGuestbook(ActionRequest request, ActionResponse response)
            throws PortalException {

            ServiceContext serviceContext = ServiceContextFactory.getInstance(
              Guestbook.class.getName(), request);

            String name = ParamUtil.getString(request, "name");
            long guestbookId = ParamUtil.getLong(request, "guestbookId");

            try {
              _guestbookService.updateGuestbook(
                serviceContext.getUserId(), guestbookId, name, serviceContext);
              SessionMessages.add(request, "guestbookUpdated");


            } catch (PortalException e) {

              Logger.getLogger(GuestbookAdminPortlet.class.getName()).log(
              Level.SEVERE, null, e);
              SessionErrors.add(request, e.getClass().getName());


              response.setRenderParameter(
                "mvcPath", "/guestbookadminportlet/edit_guestbook.jsp");
            }
          }

    public void deleteGuestbook(ActionRequest request, ActionResponse response)
            throws PortalException {

            ServiceContext serviceContext = ServiceContextFactory.getInstance(
              Guestbook.class.getName(), request);
            long guestbookId = ParamUtil.getLong(request, "guestbookId");

            try {
              _guestbookService.deleteGuestbook(guestbookId, serviceContext);
              SessionMessages.add(request, "guestbookDeleted");


            }
            catch (PortalException e) {

              Logger.getLogger(GuestbookAdminPortlet.class.getName()).log(
                  Level.SEVERE, null, e);
              SessionErrors.add(request, e.getClass().getName());


            }
          }


          private GuestbookService _guestbookService;

          @Reference(unbind = "-")
          protected void setGuestbookService(GuestbookService guestbookService) {
          _guestbookService = guestbookService;

          }


}
```

Now that you've implemented permission checks at the portlet layer, users without the proper permissions can't add, update, or delete a guestbook or guestbook entry entity. Even if a user manually entered a URL pointing to one of your portlet action methods, the portlet action now calls a remote service. The permission check in the remote service aborts the unauthorized user's request. Excellent work on securing your application's services!

From here, you can diverge. The later steps of this Learning Path (some still in progress) introduce you to other parts of Liferay DXP's development framework: workflow, the Recycle Bin, Lexicon, Friendly URLs, Staging and import/export, the Message Bus, and distributing your application on Marketplace. If you're interested in these topics, continue. But there's also another option.

Now that you've implemented web services, you can create a mobile Guestbook app with Liferay Screens. That Learning Path is available from the navigation on the left.

Chapter 36

Using Workflow

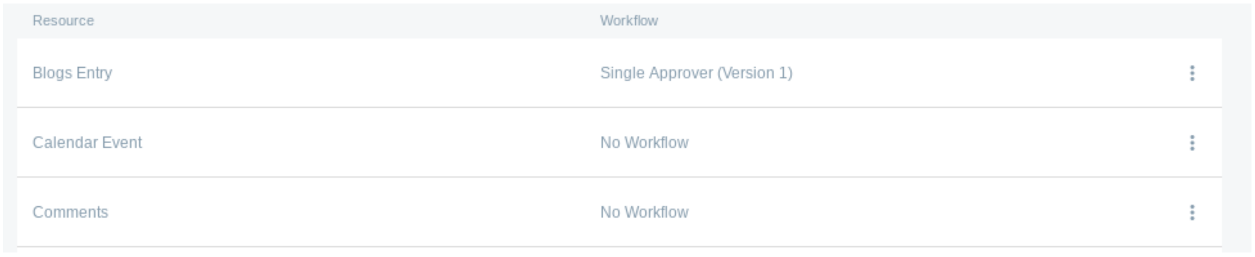
The Guestbook application accepts submissions from any logged in user, so there's no telling what people could post. Illegal data, objectionable content, the entire contents of Don Quixote: all of these and more are possibilities. You can make sure user posts don't run afoul of the law or policy by enabling *workflow* in your application.

Workflow is a review process that ensures a submitted entity isn't published before it's reviewed. To prevent posting objectionable content, an initially submitted Guestbook entry should be marked as a *draft* and sent through the workflow framework. It comes back to the application code ready to have any relevant fields updated in the database based on its status. The view layer must filter entities by status to display only reviewed entities.

Note: The exact review process is defined separately from the code that enables workflow. An XML file provides the definition of a workflow in Liferay DXP. If you're a Liferay Digital Enterprise subscriber, you have access to the Kaleo Workflow Designer, which offers a convenient drag-and-drop user interface for designing workflow definition files. You can read more about this in Liferay DXP's documentation. Liferay DXP comes with a workflow definition called the *Single Approver* definition, but you can write your own workflow definitions according to your organization's requirements.

A few additional definitions are included in Liferay DXP's source code, which you can use to see how workflow definitions are defined. To discover how to access these files, see here.

This section instructs the reader in workflow-enabling the Guestbook App's `Guestbook` and `Entry` entities to ensure that only approved content is published after review.



Resource	Workflow
Blogs Entry	Single Approver (Version 1)
Calendar Event	No Workflow
Comments	No Workflow

Figure 36.1: Enable workflow in your assets, just like Liferay DXP's own assets.

There are five steps to enabling workflow:

1. Update the service layer to set each entity's status fields.
2. Send the entity to Liferay DXP's workflow framework.
3. Add *getter* methods that account for an entity's workflow status.
4. Handle the entity as it returns from the workflow framework.
5. Update the user interface to account for workflow status.

The first three steps happen in the service layer, so that's a good place to start. Let's Go!

Chapter 37

Supporting Workflow at the Service Layer

When you asset enabled the Guestbook Application, you added four database columns in the Guestbook entities (e.g., GB_Entry) that keep track of workflow status (they're already added; celebrate!). The necessary fields are status, statusByUserName, statusByUserId, and statusDate. The columns are defined in the guestbook-service module's service.xml file.

```
<column name="status" type="int" />
<column name="statusByUserId" type="long" />
<column name="statusByUserName" type="String" />
<column name="statusDate" type="Date" />
```

The status field tells you the current status of the entity (it defaults to 0, which evaluates to *approved*). The other status fields store the date of the last change (statusDate) along with the ID and name of the user (statusByUserId and statusByUserName) who made the update.

Although the status columns are in the Guestbook application's entity tables, you must update the local service implementation's add methods to set them, and while you're there, send the entity to the workflow framework. You'll also write a method to update the status fields when the entity returns from the workflow framework, along with getters that take workflow status as a parameter. That sounds like a lot of work, but thanks to Service Builder, you must change only three files: service.xml, GuestbookLocalServiceImpl, and EntryLocalServiceImpl.

Let's Go!

37.1 Setting the Guestbook Status

```
<p>Supporting Workflow at the Service Layer<br>Step 1 of 3</p>
```

Before now, you set the status of all added guestbooks to approved in the service layer. Now you'll set it to draft and pass it to the workflow framework.

1. From guestbook-service, open GuestbookLocalServiceImpl and add the status fields below the existing setter methods in the addGuestbook method:

    ```
    guestbook.setStatus(WorkflowConstants.STATUS_DRAFT);
    guestbook.setStatusByUserId(userId);
    guestbook.setStatusByUserName(user.getFullName());
    guestbook.setStatusDate(serviceContext.getModifiedDate(null));
    ```

This manually populates the status fields and sets the workflow status as a draft in the GB_Entry database table. At this point they're identical to the similarly named non-status counterparts (like setUserId and setStatusByUserId), but they'll be updated independently in the updateStatus method you write later.

2. Still in the addGuestbook method, place the following code right before the return statement:

```
WorkflowHandlerRegistryUtil.startWorkflowInstance(guestbook.getCompanyId(),
            guestbook.getGroupId(), guestbook.getUserId(), Guestbook.class.getName(),
            guestbook.getPrimaryKey(), guestbook, serviceContext);
```

   The call to startWorkflowInstance detects whether workflow is installed and enabled. If it isn't, the added entity is automatically marked as approved. The startWorkflowInstance call also calls your GuestbookWorkflowHandler class, which you'll create later.

3. Organize imports (*[CTRL]+[SHIFT]+O*), and save your work.

The startWorkflowInstance method is where your entity enters the workflow framework, but you're not finished yet. Just like you wouldn't drop your child off at college and then change your number and move to a new address, you're not going to abandon your Guestbook entity (yet).

Exert control over how the status fields are updated in the database. Create an updateStatus method in GuestbookLocalServiceImpl, immediately following the deleteGuestbook method. Here's the first half of it:

```
public Guestbook updateStatus(long userId, long guestbookId, int status,
       ServiceContext serviceContext) throws PortalException,
       SystemException {

    User user = userLocalService.getUser(userId);
    Guestbook guestbook = getGuestbook(guestbookId);

    guestbook.setStatus(status);
    guestbook.setStatusByUserId(userId);
    guestbook.setStatusByUserName(user.getFullName());
    guestbook.setStatusDate(new Date());

    guestbookPersistence.update(guestbook);
```

If this method is called, it's because your entity is returning from the workflow framework, and it's time to update the status values in the database. Set the status fields, then persist the updated entity to the database. Before saving, finish the method:

```
    if (status == WorkflowConstants.STATUS_APPROVED) {

        assetEntryLocalService.updateVisible(Guestbook.class.getName(),
                guestbookId, true);

    } else {

        assetEntryLocalService.updateVisible(Guestbook.class.getName(),
                guestbookId, false);
    }

    return guestbook;
}
```

This if statement determines the visibility of the asset based on its workflow status. If it's approved, the assetEntryLocalService.updateVisible method sets the guestbook in question to true so it can be displayed in the Asset Publisher and in the search results. Otherwise (else) it sets the visibility to false to ensure that unapproved guestbooks aren't displayed to users in the Asset Publisher or the Search portlet.

Organize imports (*[CTRL]+[SHIFT]+O*) and save your work. Then run the buildService Gradle task.

There's one more update to make in the deleteGuestbook method. When deleting, you must clean up the workflow system's database tables to avoid leaving orphaned entries when the backing entity is deleted. Before making the method call, open service.xml and add the following tag below the existing <reference> tags:

```
<reference entity="WorkflowInstanceLink" package-path="com.liferay.portal" />
```

Save and run Service Builder. It injects the WorkflowInstanceLinkLocalService service into a protected variable in GuesbookLocalServiceBaseImpl. Since GuestbookLocalServiceImpl extends the base class, you can use it directly. Back in GuesbookLocalServiceImpl, find the deleteGuestbook method and put this method call right before the return statement:

```
workflowInstanceLinkLocalService.deleteWorkflowInstanceLinks(
    guestbook.getCompanyId(), guestbook.getGroupId(),
    Guestbook.class.getName(), guestbook.getGuestbookId());
```

Save the file and run *Refresh Gradle Project*. Now the guestbook entity's service layer populates the status fields in the database, sends the entity into the workflow framework, and cleans up when it's deleted. You'll do the same thing for guestbook entries next.

37.2 Setting the Entry Workflow Status

```
<p>Supporting Workflow at the Service Layer<br>Step 2 of 3</p>
```

Now you'll set the status fields, introduce entries to the workflow framework, and add the updateStatus method to EntryLocalServiceImpl. It works the same as it did for guestbooks.

Add the following lines in the addEntry method, immediately after the current setter methods (e.g., entry.setMessage(message)):

```
entry.setStatus(WorkflowConstants.STATUS_DRAFT);
entry.setStatusByUserId(userId);
entry.setStatusByUserName(user.getFullName());
entry.setStatusDate(serviceContext.getModifiedDate(null));
```

Still in the addEntry method, place the following code right before the return statement:

```
WorkflowHandlerRegistryUtil.startWorkflowInstance(entry.getCompanyId(),
            entry.getGroupId(), entry.getUserId(), Entry.class.getName(),
            entry.getPrimaryKey(), entry, serviceContext);
```

The startWorkflowInstance call eventually directs the workflow processing to your EntryWorkflowHandler class, which you'll create later. That class is responsible for making sure the entity is updated in the database (via an updateStatus method), but it's best practice to make persistence calls in the service layer. Thus you'll need a corresponding updateStatus method here in EntryLocalServiceImpl. Add this method to the bottom of the class:

```
 public Entry updateStatus(long userId, long guestbookId, long entryId, int status,
        ServiceContext serviceContext) throws PortalException,
        SystemException {

    User user = userLocalService.getUser(userId);
    Entry entry = getEntry(entryId);

    entry.setStatus(status);
    entry.setStatusByUserId(userId);
    entry.setStatusByUserName(user.getFullName());
    entry.setStatusDate(new Date());

    entryPersistence.update(entry);

    if (status == WorkflowConstants.STATUS_APPROVED) {

        assetEntryLocalService.updateVisible(Entry.class.getName(),
                entryId, true);

    } else {

        assetEntryLocalService.updateVisible(Entry.class.getName(),
                entryId, false);
    }

    return entry;
}
```

Organize imports (*[CTRL]+[SHIFT]+O*), save your work, and run Service Builder.

As with Guestbooks, you must add a call to `deleteWorkflowInstanceLinks` in the entry's delete method to avoid leaving orphaned database entries in the `workflowinstancelinks` table. First add the following `<reference>` tag to `service.xml`, this time in the entry entity section, below the existing reference tags:

```
<reference entity="WorkflowInstanceLink" package-path="com.liferay.portal" />
```

Save, run Service Builder, and then add the following method call to the `deleteEntry` method in `EntryLocalServiceImpl`, right before the `return` statement:

```
workflowInstanceLinkLocalService.deleteWorkflowInstanceLinks(
    entry.getCompanyId(), entry.getGroupId(),
    Entry.class.getName(), entry.getEntryId());
```

Now both entities support the status of the entity and can handle it as it enters the workflow framework and as it returns from the workflow framework. There's one more update to make in the local service implementation classes: adding getter methods that take the status as a parameter. Later you'll use these methods in the view layer so you can display only approved guestbooks and entries.

37.3 Retrieving Guestbooks and Entries by Status

```
<p>Supporting Workflow at the Service Layer<br>Step 3 of 3</p>
```

The service implementation for both entities now supports adding the status fields to the database tables. There's one more update to make in the service layer, but to understand why, you must think about the view layer. When the Guestbook portlet displays entries, you must make sure it doesn't show entries that haven't been approved. Currently, the entry's view layer shows all guestbooks:

```
    List<Guestbook> guestbooks = GuestbookLocalServiceUtil
            .getGuestbooks(scopeGroupId);
```

There's a problem: the getter only takes the `scopeGroupId` as a parameter, so there's no way to get guestbooks by their status.

Likewise, unapproved entries must not be displayed, but the view layer currently gets all entries:

```
<liferay-ui:search-container total="<%=EntryLocalServiceUtil.getEntriesCount()%>">
<liferay-ui:search-container-results
    results="<%=EntryLocalServiceUtil.getEntries(scopeGroupId.longValue(),
                guestbookId, searchContainer.getStart(),
                searchContainer.getEnd())%>" />
```

The solution is to implement for guestbooks and entries a getter that takes the `status` field as a parameter. Thankfully, Service Builder makes it easy.

Open the `guestbook-service` module's `service.xml` file.

1. For the entry entity, remove the following finder:

   ```
   <finder name="G_S" return-type="Collection">
     <finder-column name="groupId" />
     <finder-column name="status" />
   </finder>
   ```

2. Add this finder in its place:

   ```
   <finder name="G_G_S" return-type="Collection">
     <finder-column name="groupId" />
     <finder-column name="guestbookId" />
     <finder-column name="status" />
   </finder>
   ```

Run service builder (double-click `guestbook-service/build/buildService` in the Gradle Tasks pane of IDE). Service Builder generates finder methods in the persistence layer that take the specified fields (for example, `status`) as parameters.

Don't call the persistence layer directly in the application code. Instead expose the new persistence methods in the service layer.

Open `GuesbookLocalServiceImpl` and add this getter:

```
public List<Guestbook> getGuestbooks(long groupId, int status)
    throws SystemException {

    return guestbookPersistence.findByG_S(
        groupId, WorkflowConstants.STATUS_APPROVED);
}
```

This getter gets only approved guestbooks. That's why you hard code the workflow constant `STATUS_APPROVED` into the status parameter when calling the persistence method. Now open `EntryLocalServiceImpl` and add these two getters:

```
public List<Entry> getEntries(
    long groupId, long guestbookId, int status, int start, int end)
    throws SystemException {

    return entryPersistence.findByG_G_S(
        groupId, guestbookId, WorkflowConstants.STATUS_APPROVED);
}

public int getEntriesCount(
    long groupId, long guestbookId, int status)
```

```
    throws SystemException {

    return entryPersistence.countByG_G_S(
        groupId, guestbookId, WorkflowConstants.STATUS_APPROVED);
}
```

You'll replace the existing methods with these `getEntries` and `getEntriesCount` methods in the view layer, ensuring that only approved entries are displayed.

The work here relates to the UI updates you'll make later. Next, implement workflow handlers so that you can call the `updateStatus` service method when the entity returns from the workflow framework.

Chapter 38

Handling Workflow

The guestbook project's service layer is now updated to handle workflow. It now properly sets the status fields for guestbooks and guestbook entries, gets entities by their statuses, and sends entities to Liferay DXP's workflow framework whenever the `addGuestbook` or `addEntry` methods are called. Recall that you still have an uncalled service method, `updateStatus`, for both entities. In this section you'll implement workflow handlers, classes that interact with Liferay DXP's workflow framework and your service layer (by calling `updateStatus` on the appropriate entity).

There's a handy abstract class you can extend to make the job easier, called `BaseWorkflowHandler`. You'll do this next for both entities of the guestbook project, starting with guestbooks.

Let's Go!

38.1 Creating a Workflow Handler for Guestbooks

<p>Handling Workflow<br>Step 1 of 2</p>

Each workflow enabled entity needs a `WorkflowHandler`. Create a new package in the `guestboook-service` module called `com.liferay.docs.guestbook.workflow`, then create the `GuestbokWorkflowHandler` class in it. Extend `BaseWorkflowHandler` and pass in `Guestbook` as the type parameter:

```
public class GuestbookWorkflowHandler extends BaseWorkflowHandler<Guestbook> {
```

Make it a Component class:

```
@Component(immediate = true, service = WorkflowHandler.class)
```

There are three abstract methods to implement: `getClassName`, `getType`, and `updateStatus`.

```
    @Override
    public String getClassName() {
        return Guestbook.class.getName();
    }
```

`getClassName` returns the guestbook entity's fully qualified class name (`com.liferay.docs.guestbook.model.Guestbook`).

```
    @Override
    public String getType(Locale locale) {
        return _resourceActions.getModelResource(locale, getClassName());
    }
```

getType returns the model resource name (model.resource.com.liferay.docs.guestbook.model.Guestbook). The meat of the workflow handler is in the updateStatus method:

```
@Override
public Guestbook updateStatus(
        int status, Map<String, Serializable> workflowContext)
    throws PortalException {

    long userId = GetterUtil.getLong(
        (String)workflowContext.get(WorkflowConstants.CONTEXT_USER_ID));
    long resourcePrimKey = GetterUtil.getLong(
        (String)workflowContext.get(
            WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));

    ServiceContext serviceContext = (ServiceContext)workflowContext.get(
        "serviceContext");

    return _guestbookLocalService.updateStatus(
        userId, resourcePrimKey, status, serviceContext);
}
```

When you crafted the service layer's updateStatus method (see the last section for more details), you specified parameters that must be passed to the method. Here you're making sure that those parameters are available to pass to the service call. Get the userId and resourcePrimKey from GetterUtil. Its getLong method takes a String, which you can get from the workflowContext Map using WorkflowConstants for the context user ID and the context entry class PK.

Make sure you inject the ResourceActions service into a private variable at the end of the class, using the @Reference annotation:

```
@Reference(unbind = "-")
protected void setResourceActions(ResourceActions resourceActions) {

    _resourceActions = resourceActions;
}

private ResourceActions _resourceActions;
```

Inject a GuestbookLocalService into a private variable using the @Reference annotation.

```
    @Reference(unbind = "-")
    protected void setGuestbookLocalService(
        GuestbookLocalService guestbookLocalService) {

        _guestbookLocalService = guestbookLocalService;
    }

    private GuestbookLocalService _guestbookLocalService;

}
```

Organize imports (*[CTRL]+[SHIFT]+O*) and save your work.

Now the Guestbook Application updates the database with the necessary status information, interacting with Liferay's workflow classes to make sure each entity is properly handled by Liferay DXP. At this point you can enable workflow for the Guestbook inside Liferay DXP and see how it works. Navigate to *Control Panel → Workflow Configuration*. The Guestbook entity appears among Liferay DXP's native entities. Enable the Single Approver Workflow for Guestbooks; then go to the Guestbook Admin portlet and add a new Guestbook. A notification appears next to your user name in the product menu. You receive a notification from the workflow that a task is ready for review. Click it, and you're taken to the My Workflow Tasks portlet, where you can complete the review task.

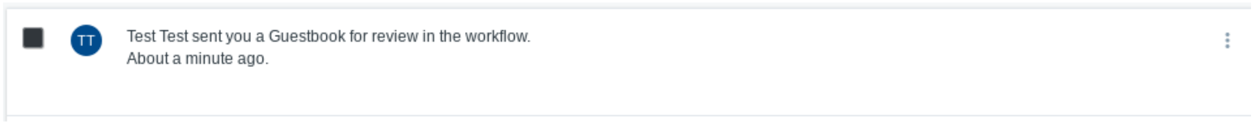

Test Test sent you a Guestbook for review in the workflow.
About a minute ago.

Figure 38.1: Click the workflow notification in the Notifications portlet to review the guestbook submitted to the workflow.

To complete the review, click the actions button (⋮) from My Workflow Tasks and select *Assign to Me*. Click the actions button again and select *Approve*.

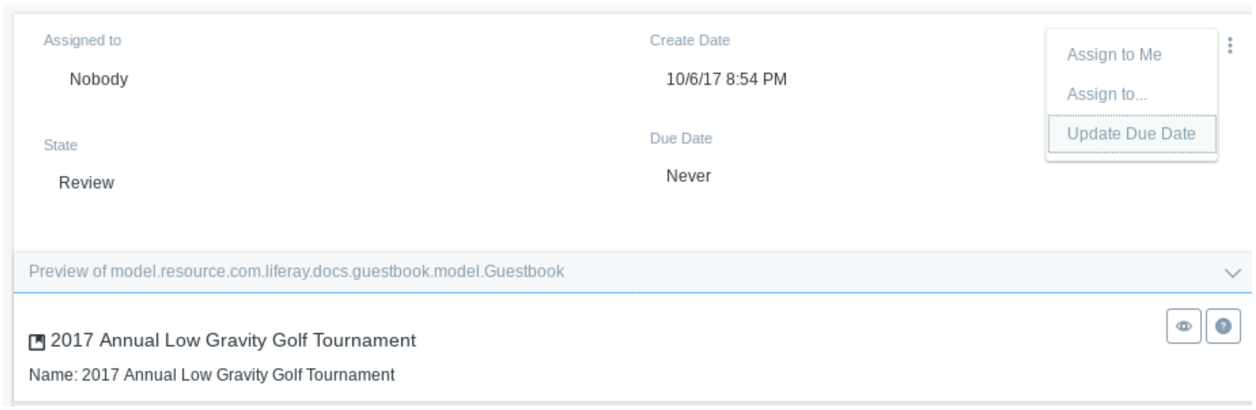

Assigned to
Nobody
Create Date
10/6/17 8:54 PM
State
Review
Due Date
Never
Preview of model.resource.com.liferay.docs.guestbook.model.Guestbook
2017 Annual Low Gravity Golf Tournament
Name: 2017 Annual Low Gravity Golf Tournament

Figure 38.2: Click the workflow notification in the Notifications portlet to review the guestbook submitted to the workflow.

Right now the workflow process for guestbooks is functional, but the UI isn't adapted for it. You'll write the workflow handler for guestbook entries next, and then update the UI to account for each entity's workflow status.

38.2 Creating a Workflow Handler for Guestbook Entries

`<p>Handling Workflow<br>Step 2 of 2</p>`

The entry's workflow handler is almost identical to the guestbook's. Create a new class in the `com.liferay.docs.guestbook.workflow` package of the `guestbook-service` module. Name it `EntryWorkflowHandler` and extend `BaseWorkflowHandler`. Decorate it with a Component annotation and implement the same three methods you implemented in the `GuestbookWorkflowHandler`. Paste this in as the class body:

```
@Component(immediate = true, service = WorkflowHandler.class)
public class EntryWorkflowHandler extends BaseWorkflowHandler<Entry> {

    @Override
    public String getClassName() {

        return Entry.class.getName();

    }

    @Override
    public String getType(Locale locale) {

        return _resourceActions.getModelResource(locale, getClassName());
    }

    @Override
    public Entry updateStatus(
        int status, Map<String, Serializable> workflowContext)
        throws PortalException {

        long userId = GetterUtil.getLong(
            (String) workflowContext.get(WorkflowConstants.CONTEXT_USER_ID));
        long resourcePrimKey = GetterUtil.getLong(
            (String) workflowContext.get(
                WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));

        ServiceContext serviceContext =
            (ServiceContext) workflowContext.get("serviceContext");

        long guestbookId =
            _entryLocalService.getEntry(resourcePrimKey).getGuestbookId();

        return _entryLocalService.updateStatus(
            userId, guestbookId, resourcePrimKey, status, serviceContext);
    }

    @Reference(unbind = "-")
    protected void setEntryLocalService(EntryLocalService entryLocalService) {

        _entryLocalService = entryLocalService;
    }

    @Reference(unbind = "-")
    protected void setResourceActions(ResourceActions resourceActions) {

        _resourceActions = resourceActions;
    }

    private EntryLocalService _entryLocalService;
    private ResourceActions _resourceActions;
}
```

There is nothing unique about this code as compared with the guestbook's workflow handler, except that we need the `gustbookId` for the entry. That's easily obtained by getting the `Entry` object with `entryLocalService`, then getting its `guestbookId`. See the last article for the rest of the handler's implementation details.

Organize imports with *CTRL+SHIFT+O* and save the file.

The back-end of the guestbook project is fully workflow enabled. All that's left is to update the Guestbook Application's UI to handle workflow status.

Chapter 39

Displaying Approved Workflow Items

There's not much left to do. Both entities in the guestbook project's back-end are workflow enabled, so it's time to update the UI. The Guestbook Admin portlet and the Guestbook portlet each requires its own display strategy.

The Guestbook Admin application is accessed by administrators, so it can display all guestbooks that have been submitted, even if they're not marked as approved. However, adding a *Status* field to the search container makes sense. That way admins can see which guestbooks are already approved, which are drafts, which are pending, etc.

The Guestbook application is meant to be viewed by site members and even guests (unauthenticated users of your site). Here it's smart to display only approved guestbooks and approved entries.

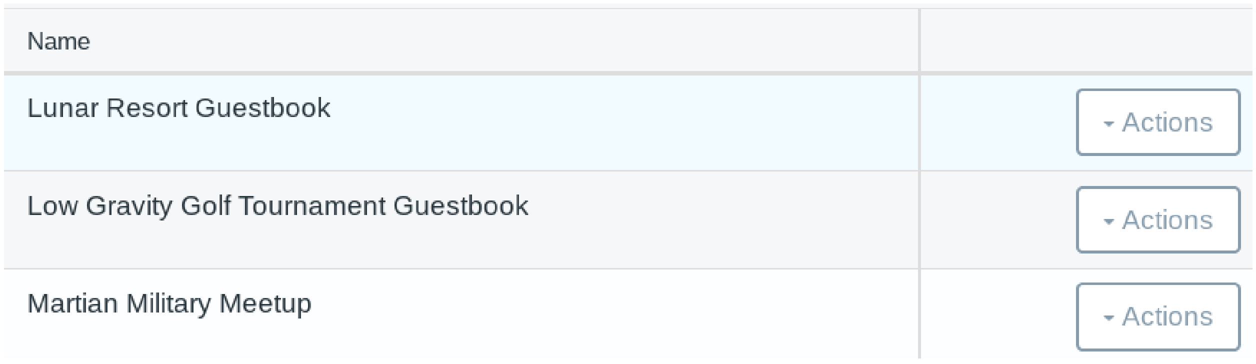
Start by updating the Guestbook Admin UI.

Let's Go!

39.1 Displaying Guestbook Status

<p>Displaying Approved Workflow Items<br>Step 1 of 2</p>

The Guestbook Admin application's main view currently has a search container with two columns: the guestbook name and the guestbook actions button.



Name	
Lunar Resort Guestbook	Actions
Low Gravity Golf Tournament Guestbook	Actions
Martian Military Meetup	Actions

Figure 39.1: The Guestbook Admin's main view currently shows the name of the guestbook and its actions button.

1. Add a third column between the two existing ones: call it *Status*. Open

   ```
   guestbook-web/src/main/reosurces/META-INF/resources/guestbookadminportlet/view.jsp
   ```

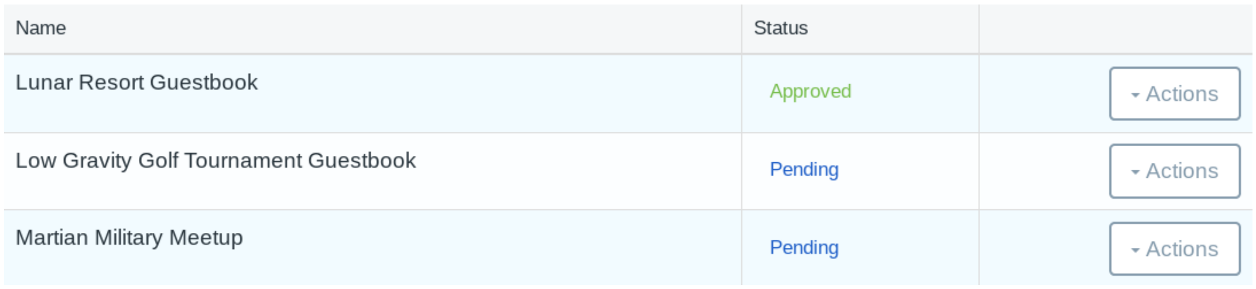
2. Find the existing search-container-column definitions:

   ```
   <liferay-ui:search-container-column-text property="name" />

   <liferay-ui:search-container-column-jsp align="right"
       path="/guestbookadminportlet/guestbook_actions.jsp" />
   ```

3. Put the following new column between the existing columns:

   ```
   <liferay-ui:search-container-column-status property="status" />
   ```

Save the file and wait for the web module to redeploy. With the addition of one line in the JSP, the Guestbook Admin application now displays the guestbook's workflow status.



Name	Status	
Lunar Resort Guestbook	Approved	Actions
Low Gravity Golf Tournament Guestbook	Pending	Actions
Martian Military Meetup	Pending	Actions

Figure 39.2: The Guestbook Admin's main view, displaying the status of each guestbook.

Now move on to the Guestbook application's view layer.

39.2 Displaying Approved Entries

```
<p>Displaying Approved Workflow Items<br>Step 2 of 2</p>
```

The Guestbook application needs to be updated so that only guestbooks and entries with a status of *approved* appear in the UI.
Change the getters used to retrieve both entities in the view layer.

1. You need a new import, so first open

   ```
   guestbook-web/src/main/resources/META-INF/resources/init.jsp
   ```

   and add this line:

   ```
   <%@ page import="com.liferay.portal.kernel.workflow.WorkflowConstants"%>
   ```

2. Now open

```
guestbook-web/src/main/resources/META-INF/resources/guestbookwebportlet/view.jsp
```

Find the scriptlet that retrieves guestbooks:

```
<%
    List<Guestbook> guestbooks = GuestbookLocalServiceUtil
                .getGuestbooks(scopeGroupId);
        for (int i = 0; i < guestbooks.size(); i++) {
            Guestbook curGuestbook = (Guestbook) guestbooks.get(i);
            String cssClass = StringPool.BLANK;
            if (curGuestbook.getGuestbookId() == guestbookId) {
                cssClass = "active";
            }
            if (GuestbookPermission.contains(
                permissionChecker, curGuestbook.getGuestbookId(), "VIEW")) {

%>
```

Change it so it calls the getter you added that takes workflow status into account. All you need to do is change this method call

```
List<Guestbook> guestbooks = GuestbookLocalServiceUtil
            .getGuestbooks(scopeGroupId);
```

to

```
List<Guestbook> guestbooks = GuestbookLocalServiceUtil
            .getGuestbooks(scopeGroupId, WorkflowConstants.STATUS_APPROVED);
```

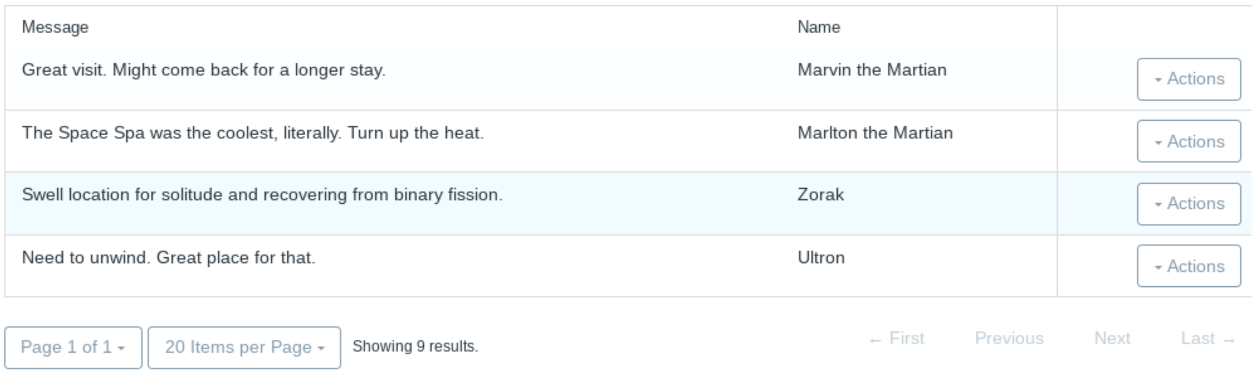
Save the file, and now only approved guestbooks are displayed in the Guestbook application.

3. Next, update the entry's UI in the same `view.jsp`. Find the tags that set the search container's total and its results:

```
<liferay-ui:search-container total="<%=EntryLocalServiceUtil.
                getEntriesCount()%>">
<liferay-ui:search-container-results results=
                "<%=EntryLocalServiceUtil.getEntries
                (scopeGroupId.longValue(),
                guestbookId, searchContainer.getStart(),
                searchContainer.getEnd())%>" />
```

Replace the getters to use the ones that take workflow status as a parameter, and pass `WorkflowConstants.STATUS_APPROVED` as the status. Here's what it looks like when you're finished:

```
<liferay-ui:search-container total="<%=EntryLocalServiceUtil.
                getEntriesCount(scopeGroupId.longValue(),
                guestbookId, WorkflowConstants.STATUS_APPROVED)%>">
<liferay-ui:search-container-results results=
                "<%=EntryLocalServiceUtil.getEntries(
                scopeGroupId.longValue(), guestbookId,
                WorkflowConstants.STATUS_APPROVED,
                searchContainer.getStart(), searchContainer.getEnd())%>" />
```



Message	Name	
Great visit. Might come back for a longer stay.	Marvin the Martian	Actions
The Space Spa was the coolest, literally. Turn up the heat.	Marlton the Martian	Actions
Swell location for solitude and recovering from binary fission.	Zorak	Actions
Need to unwind. Great place for that.	Ultron	Actions

Page 1 of 1 · 20 Items per Page · Showing 9 results.

Figure 39.3: If you don't update the counter method to account for workflow status, it displays an incorrect count in the search container.

Now only approved entries are displayed, and the search container's counter only counts the approved entries. If you update the `getEntries` call but not the `getEntriesCount` call, the count that's displayed includes approved entries and entries with any other workflow status, and it won't match the total that's displayed at the bottom of the search container.

Now Guestbooks and Entries are now fully workflow enabled, to the great relief of the Lunar Resort's site administrators. You've saved them a lot of headaches dealing with inappropriate content, primarily submitted by visitors from Mars. Those Martians really need some lessons in netiquette.

Chapter 40

Enabling Staging and Export/Import

Your Guestbook application creates guestbooks and entries that are immediately published when they're saved. Sites constantly change, however, so it's crucial to have an area where updates can be planned and tested before publishing to your audience. Staging enables changing your Site behind the scenes without affecting the live Site. When you're done, you can publish all the changes at once.

Next, you'll implement Staging support in your Guestbook app so its entries can be tracked during the Staging phase of your publishing process.

Export/Import facilitates extracting data so it can be imported into another Liferay DXP installation. Behind the scenes, Export/Import is used during the Staging process. When publishing your staged content to the live Site, you're essentially importing content from the staged Site and exporting it to the live Site. Since the Export/Import framework is programmatically similar to Staging, you can implement it with Staging.

Ready to support Staging in your Guestbook app?

Let's Go!

40.1 Creating Staged Models

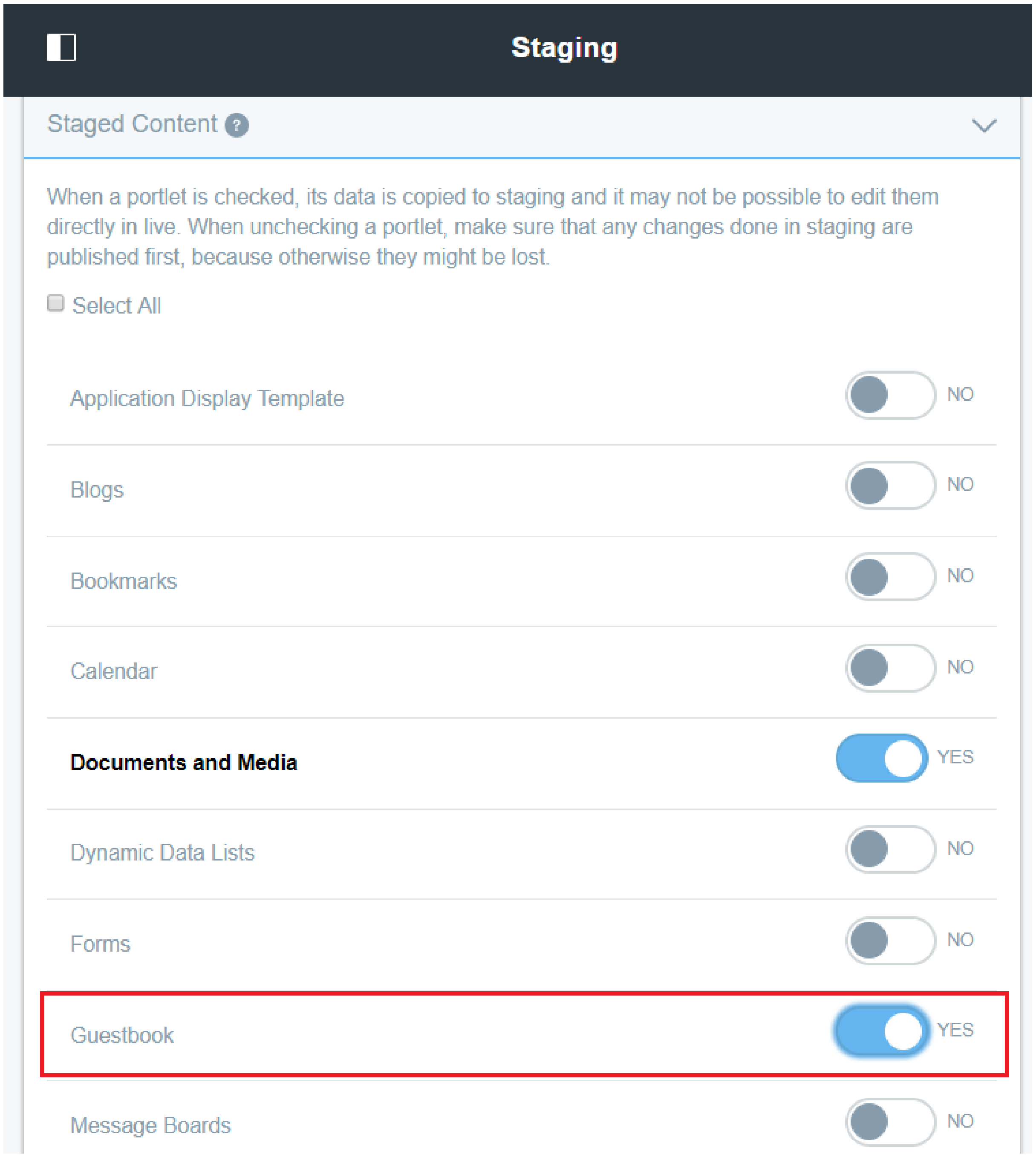
`<p>Enabling Staging and Export/Import<br>Step 1 of 7</p>`

To implement the Staging framework, you must first specify the entities you want to track. For the Guestbook application, there are two: `Guestbooks` and `Entrys`. You can register these entities so they're recognizable to the Staging framework by implementing the `StagedModel` interface in your Guestbook's model classes.

Service Builder generates an app's models as staged models when certain attributes are specified in the app's `service.xml` file. The Guestbook app already defines many of the necessary attributes in its `service.xml` file, so both your `GuestbookModel` and `EntryModel` interfaces already extend the `StagedModel` interface! For example, your Guestbook app's `EntryModel` interface's declaration looks like this:

```
public interface EntryModel extends BaseModel<Entry>, GroupedModel, ShardedModel,
    StagedAuditedModel, WorkflowedModel {
```

The `StagedModel` interface is implemented by the extension of the `StagedAuditedModel` interface. Service Builder chose the `StagedAuditedModel` interface based on the columns you declared. You'll update this later.


Staging

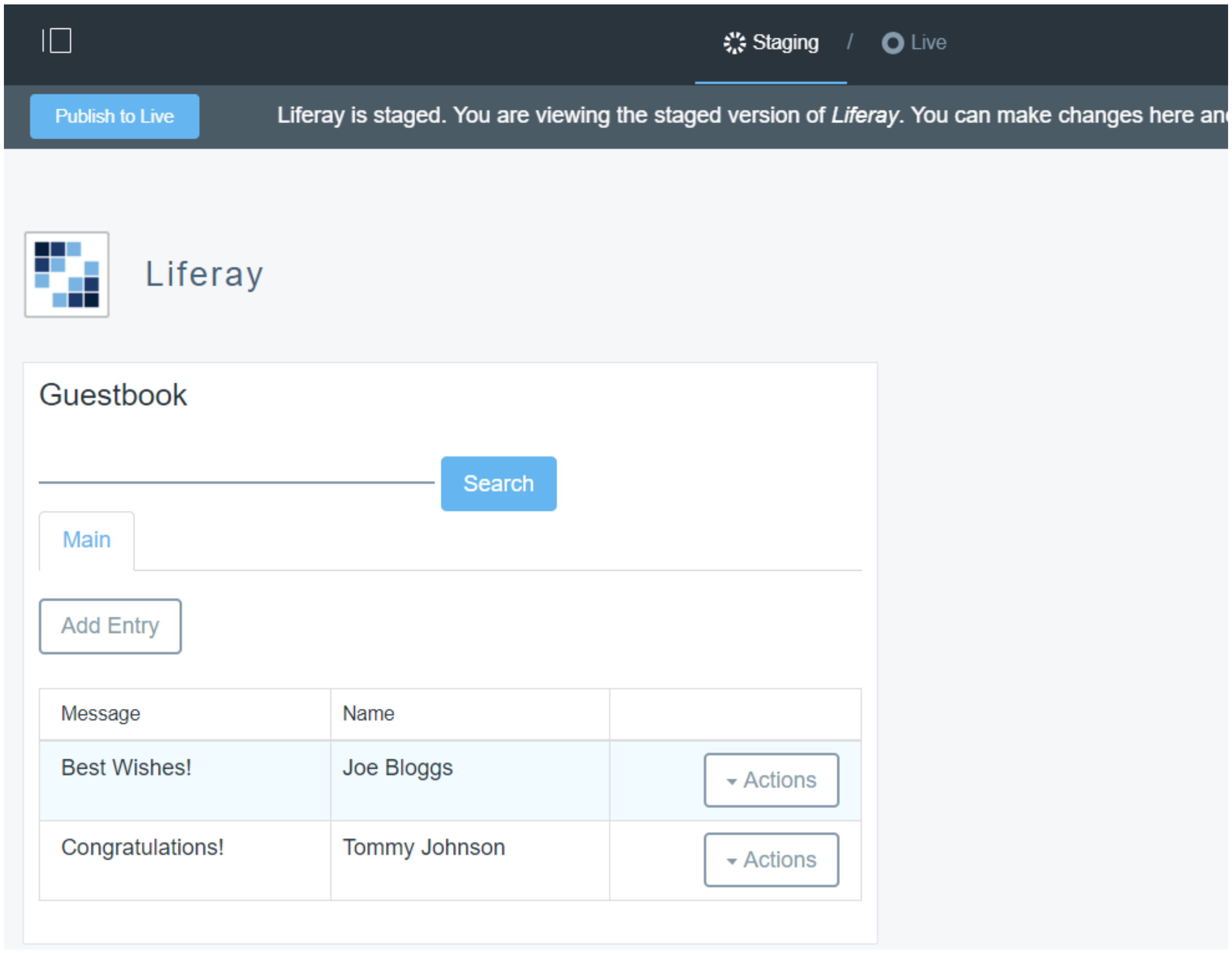
Staged Content

When a portlet is checked, its data is copied to staging and it may not be possible to edit them directly in live. When unchecking a portlet, make sure that any changes done in staging are published first, because otherwise they might be lost.

Select All

Application Display Template	NO
Blogs	NO
Bookmarks	NO
Calendar	NO
Documents and Media	YES
Dynamic Data Lists	NO
Forms	NO
Guestbook	YES
Message Boards	NO

Figure 40.1: Once Staging is implemented in your Guestbook app, you can have its data tracked by the Staging framework.



Liferay is staged. You are viewing the staged version of *Liferay*. You can make changes here an

Message	Name	
Best Wishes!	Joe Bloggs	Actions
Congratulations!	Tommy Johnson	Actions

Figure 40.2: A Staging-enabled Guestbook app can be modified on the staged site first without any users seeing it on the live Site.

The following Staging-specific attributes/columns are currently defined in the Guestbook app's `service.xml` file:

- `uuid` (required)
- `groupId`
- `companyId` (required)
- `userId`
- `userName`
- `createDate` (required)
- `modifiedDate` (required)

One of the most important attributes used by the Staging framework is the UUID (Universally Unique Identifier). This attribute must be set to true in `service.xml` for Service Builder to recognize your model as an eligible staged model. The UUID can differentiate entities between environments, because it's unique across multiple systems.

The `companyId`, `createDate`, and `modifiedDate` columns track the current entity's instance and creation/modification dates.

The others leverage features of the Staging framework like automatic group mapping or entity level Last Publish Date handling. See the Understanding Staged Models tutorial for more information.

Before adding Staging features to your Guestbook app, you must declare some necessary dependencies.

Declaring Staging Dependencies

There are two Staging-specific dependencies used by the Guestbook's Staging functionality.

1. Open the `guestbook-service` module's `build.gradle` file.

2. Add the following dependencies within the `dependencies` block:

   ```
   compileOnly group: "com.liferay", name: "com.liferay.exportimport.api", version: "2.1.0"
   compileOnly group: "com.liferay", name: "com.liferay.xstream.configurator.api", version: "2.0.0"
   ```

3. Save the file, right-click the Guestbook project, and run *Gradle → Refresh Gradle Project*.

Now you're ready to begin implementing staging in your app.

Updating the Extended Staged Model Interface

Staged models that extend the `StagedAuditedModel` interface function independently from the group concept (sometimes referred to as company models). This means that, for example, your guestbook and entry's scope would not be tracked by the Staging framework. You must add one more column to your `service.xml` file to convert your models to `StagedGroupedModels`, so your entities' scope is tracked correctly.

1. Open your `guestbook-service` module's `service.xml` file and add the `lastPublishDate` column for both Guestbook and Entry entities:

   ```
   <column name="lastPublishDate" type="Date" />
   ```

2. Run Service Builder. Do this by navigating to the Gradle Tasks pane on the right side of IDE and selecting your project's *build → buildService* task.

3. Run *Gradle → Refresh Gradle Project* to resolve any remaining errors.

Service Builder has updated your models to extend the `StagedGroupedModel`. For example, your `EntryModel` interface's declaration now looks like this:

```
public interface EntryModel extends BaseModel<Entry>, ShardedModel,
    StagedGroupedModel, WorkflowedModel {
```

For more information on the available staged model interfaces, see this tutorial.

Excellent! Now it's time to create your staged model data handlers.

40.2 Creating the Entry Staged Model Data Handler

```
<p>Enabling Staging and Export/Import<br>Step 2 of 7</p>
```

A Staged Model Data Handler supplies information about a staged model (entity) to the Staging and Export/Import framework. Data handlers replace the need to access the database directly and run queries to export/import data.

You must create a staged model data handler for every entity you want Staging to track. This means you must create a data handler for both your guestbook and entry entities.

First, you'll create a staged model data handler for guestbook entries.

1. In your `guestbook-service` module, create a package named `com.liferay.docs.guestbook.exportimport.data.handler`.

2. In that package, create the `EntryStagedModelDataHandler` class and have it extend the `BaseStagedModelDataHandler<STAGEI` class:

    ```
    public class EntryStagedModelDataHandler
        extends BaseStagedModelDataHandler<Entry> {
    ```

3. Add an `@Component` annotation above the class declaration to declare that the `EntryStagedModelDataHandler` class provides an implementation of the `StagedModelDataHandler` service:

    ```
    @Component(
        immediate = true,
        service = StagedModelDataHandler.class
    )
    ```

4. Set the staged model's local services you need in your data handler:

    ```
    @Reference(unbind = "-")
    protected void setEntryLocalService(EntryLocalService entryLocalService) {

        _entryLocalService = entryLocalService;
    }

    @Reference(unbind = "-")
    protected void setGuestbookLocalService(
        GuestbookLocalService guestbookLocalService) {

        _guestbookLocalService = guestbookLocalService;
    }

    private EntryLocalService _entryLocalService;
    private GuestbookLocalService _guestbookLocalService;
    ```

    This injects the entry and guestbook's local services.

5. You must provide the class names of the models the data handler tracks. You can do this by overriding the `StagedModelDataHandler`'s `getClassNames()` method:

    ```
    public static final String[] CLASS_NAMES = {Entry.class.getName()};

    @Override
    public String[] getClassNames() {
            return CLASS_NAMES;
    }
    ```

As a best practice, you should have one staged model data handler per staged model. It's possible to use multiple class types, but this is not recommended.

6. Add a method that retrieves the staged model's display name:

```
@Override
public String getDisplayName(Entry entry) {
    return entry.getName();
}
```

The display name is presented with the progress bar during the export/import and publication processes.

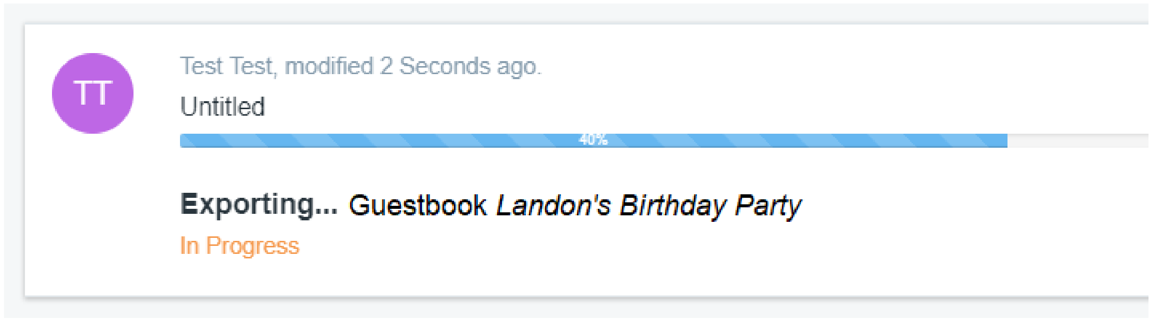

TT
Test Test, modified 2 Seconds ago.
Untitled
40%
Exporting... Guestbook Landon's Birthday Party
In Progress

Figure 40.3: Your staged model data handler provides the display name in the Publish to Live UI.

7. A staged model data handler should ensure everything required for its operation is also imported/exported. For example, an entry requires a guestbook. Therefore, the guestbook should be exported first followed by the entry.

   Add methods that import and export your staged model and its references.

```
@Override
protected void doExportStagedModel(
        PortletDataContext portletDataContext, Entry entry)
    throws Exception {

    Guestbook guestbook =
        _guestbookLocalService.getGuestbook(entry.getGuestbookId());

    StagedModelDataHandlerUtil.exportReferenceStagedModel(
        portletDataContext, entry, guestbook,
        PortletDataContext.REFERENCE_TYPE_PARENT);

    Element entryElement = portletDataContext.getExportDataElement(entry);

    portletDataContext.addClassedModel(
        entryElement, ExportImportPathUtil.getModelPath(entry), entry);
}

@Override
protected void doImportStagedModel(
        PortletDataContext portletDataContext, Entry entry)
    throws Exception {

    long userId = portletDataContext.getUserId(entry.getUserUuid());

    Map<Long, Long> guestbookIds =
        (Map<Long, Long>) portletDataContext.getNewPrimaryKeysMap(
            Guestbook.class);

    long guestbookId = MapUtil.getLong(
        guestbookIds, entry.getGuestbookId(), entry.getGuestbookId());

    Entry importedEntry = null;

    ServiceContext serviceContext =
        portletDataContext.createServiceContext(entry);

    if (portletDataContext.isDataStrategyMirror()) {
        Entry existingEntry = fetchStagedModelByUuidAndGroupId(
            entry.getUuid(), portletDataContext.getScopeGroupId());

        if (existingEntry == null) {
            serviceContext.setUuid(entry.getUuid());

            importedEntry = _entryLocalService.addEntry(
                userId, guestbookId, entry.getName(), entry.getEmail(),
                entry.getMessage(), serviceContext);
        }
        else {
            importedEntry = _entryLocalService.updateEntry(
                userId, guestbookId, existingEntry.getEntryId(),
                entry.getName(), entry.getEmail(), entry.getMessage(),
                serviceContext);
        }
    }
    else {
        importedEntry = _entryLocalService.addEntry(
            userId, guestbookId, entry.getName(), entry.getEmail(),
            entry.getMessage(), serviceContext);
    }

    portletDataContext.importClassedModel(entry, importedEntry);
}
```

The `doExportStagedModel` method retrieves the entry's data element from the `PortletDataContext` and then adds the class model characterized by that data element to the `PortletDataContext`. The `PortletDataContext` populates the LAR file with your application's data during the export process. Note that once an entity has been exported, subsequent calls to the export method don't repeat the export process multiple times, ensuring optimal performance.

An important feature of the import process is that all exported reference elements are automatically imported when needed. The `doImportStagedModel` method does not need to import the reference elements manually; it must only find the new assigned ID for the guestbook before importing the entry.

The `PortletDataContext` keeps this information and a slew of other data up-to-date during the import process. The code snippet shows how to access the old ID and new ID mapping, by using the `portletDataContext.getNewPrimaryKeysMap()` method. The method proceeds with checking the import mode (e.g., Copy As New or Mirror) and depending on the process configuration and existing environment, the entry is either added or updated.

8. When importing a LAR (i.e., publishing to the live Site), the import process expects all of an entity's references to be available and validates their existence.

   For example, if you republish an updated guestbook to the live Site and did not include some of its existing entries in the publication, these entries are considered missing references. A more practical example of this would be an image included in a web content article. If the image included in the web content lives on a different Site (i.e., the image is contained in a different group) or was not included in the publication process, it's considered a missing reference of the web content article.

Since you're dealing with references on two separate Sites that have differing IDs, the system can't easily match them during publication. Consider this scenario for the Guestbook app; suppose you export a guestbook entry as a missing reference with a primary key (ID) of 1. When importing that information, the LAR only provides the ID but not the entry itself. Therefore, during the import process, the Data Handler framework searches for the entry to replace by its UUID, but the entry to replace has a different ID (primary key) of 2. You must provide a way to handle these missing references.

To do this, you must add a method that maps the missing reference's primary key from the export to the existing primary key during import. Since the reference's UUID is constant across systems, it's used to complete the mapping of differing primary keys. Note that a reference can only be missing on the live Site if it has already been published previously. Therefore, when publishing a guestbook for the first time, the system doesn't check for missing references.

Add this method to your class:

```
@Override
protected void doImportMissingReference(
    PortletDataContext portletDataContext, String uuid, long groupId,
    long entryId)
throws Exception {

    Entry existingEntry = fetchMissingReference(uuid, groupId);

    if (existingEntry == null) {
        return;
    }

    Map<Long, Long> entryIds =
        (Map<Long, Long>) portletDataContext.getNewPrimaryKeysMap(
        Entry.class);

    entryIds.put(entryId, existingEntry.getEntryId());
}
```

This method maps the existing staged model to the old ID in the reference element. When a reference is exported as missing, the Data Handler framework calls this method during the import process and updates the new primary key map in the portlet data context.

9. Provide a way for the staged model data handler to fetch your staged models:

```
@Override
public Entry fetchStagedModelByUuidAndGroupId(String uuid, long groupId) {

    return _entryLocalService.fetchEntryByUuidAndGroupId(uuid, groupId);
}

@Override
public List<Entry> fetchStagedModelsByUuidAndCompanyId(
    String uuid, long companyId) {

    return _entryLocalService.getEntriesByUuidAndCompanyId(
        uuid, companyId, QueryUtil.ALL_POS, QueryUtil.ALL_POS,
        new StagedModelModifiedDateComparator<Entry>());
}
```

These methods use the entry's local service to get the entries by UUID and company ID (i.e., portal instance's primary key) or group ID (i.e., Site, Organization, or User Group's primary key).

10. Override the BaseStagedModelDataHandler's delete methods to leverage your newly created fetch method and custom local service:

```
@Override
public void deleteStagedModel(
        String uuid, long groupId, String className, String extraData)
    throws PortalException {

    Entry entry = fetchStagedModelByUuidAndGroupId(uuid, groupId);

    if (entry ≠ null) {
        deleteStagedModel(entry);
    }

}

@Override
public void deleteStagedModel(Entry entry)
    throws PortalException {

    _entryLocalService.deleteEntry(entry);
}
```

    These methods are necessary for the Staging framework to properly delete your entry staged models.

11. Organize your imports (*[CTRL]+[SHIFT]+O*), and save the file.

The entry's staged model data handler is complete! Next you can create the guestbook's staged model data handler.

40.3 Creating the Guestbook Staged Model Data Handler

```
<p>Enabling Staging and Export/Import<br>Step 3 of 7</p>
```

The guestbook's staged model data handler is similar to the entry's data handler. Refer to the previous article for how this works.

1. In the `guestbook-service` module's `com.liferay.docs.guestbook.exportimport.data.handler` package, create the `GuestbookStagedModelDataHandler` class.

2. Declare `BaseStagedModelDataHandler<STAGED_MODEL>` as your extension class and add the `@Component` declaration to declare `StagedModelDataHandler` as your implemented service:

```
@Component(
    immediate = true,
    service = StagedModelDataHandler.class
)
public class GuestbookStagedModelDataHandler
    extends BaseStagedModelDataHandler<Guestbook> {
```

3. Set the staged model's local service you want to leverage in your data handler:

```
@Reference(unbind = "-")
protected void setGuestbookLocalService(
    GuestbookLocalService guestbookLocalService) {

    _guestbookLocalService = guestbookLocalService;
}

private GuestbookLocalService _guestbookLocalService;
```

4. Add the methods to retrieve the guestbook staged model's classes to track and display names:

```
public static final String[] CLASS_NAMES = {
    Guestbook.class.getName()
};

@Override
public String[] getClassNames() {

    return CLASS_NAMES;
}

@Override
public String getDisplayName(Guestbook guestbook) {

    return guestbook.getName();
}
```

5. Add methods to ensure all import/export information is provided to the Staging framework for your guestbook entity:

```
@Override
protected void doExportStagedModel(
        PortletDataContext portletDataContext, Guestbook guestbook)
    throws Exception {

    Element guestbookElement =
        portletDataContext.getExportDataElement(guestbook);

    portletDataContext.addClassedModel(
        guestbookElement, ExportImportPathUtil.getModelPath(guestbook),
        guestbook);
}

@Override
protected void doImportStagedModel(
        PortletDataContext portletDataContext, Guestbook guestbook)
    throws Exception {

    long userId = portletDataContext.getUserId(guestbook.getUserUuid());

    Map<Long, Long> guestbookIds =
        (Map<Long, Long>) portletDataContext.getNewPrimaryKeysMap(
        Guestbook.class);

    long guestbookId = MapUtil.getLong(
        guestbookIds, guestbook.getGuestbookId(),
        guestbook.getGuestbookId());

    Guestbook importedGuestbook = null;

    ServiceContext serviceContext =
        portletDataContext.createServiceContext(guestbook);

    if (portletDataContext.isDataStrategyMirror()) {
        Guestbook existingGuestbook = fetchStagedModelByUuidAndGroupId(
            guestbook.getUuid(), portletDataContext.getScopeGroupId());

        if (existingGuestbook == null) {
            serviceContext.setUuid(guestbook.getUuid());

            importedGuestbook = _guestbookLocalService.addGuestbook(
                userId, guestbook.getName(), serviceContext);
        }
        else {
            importedGuestbook = _guestbookLocalService.updateGuestbook(
                userId, existingGuestbook.getGuestbookId(), guestbook.getName(), serviceContext);

        }
    }
    else {
        importedGuestbook = _guestbookLocalService.addGuestbook(
            userId, guestbook.getName(), serviceContext);
    }

    portletDataContext.importClassedModel(guestbook, importedGuestbook);
}
```

Similar to the guestbook entry, these methods add export/import information to the `PortletDataContext`.

6. Add a method that maps the missing reference ID from the export to the existing ID during import:

```
@Override
protected void doImportMissingReference(
        PortletDataContext portletDataContext, String uuid, long groupId,
        long guestbookId)
    throws Exception {

    Guestbook existingGuestbook = fetchMissingReference(uuid, groupId);

    if (existingGuestbook == null) {
        return;
    }

    Map<Long, Long> guestbookIds =
        (Map<Long, Long>) portletDataContext.getNewPrimaryKeysMap(
        Guestbook.class);

    guestbookIds.put(guestbookId, existingGuestbook.getGuestbookId());
}
```

Remember, this is not called for new entities being published/imported since all of an entity's references are new to the live Site.

7. Provide a way for the staged model data handler to fetch your staged models:

```
@Override
public Guestbook fetchStagedModelByUuidAndGroupId(
    String uuid, long groupId) {

    return _guestbookLocalService.fetchGuestbookByUuidAndGroupId(
        uuid, groupId);
}

@Override
public List<Guestbook> fetchStagedModelsByUuidAndCompanyId(
    String uuid, long companyId) {
        return _guestbookLocalService.getGuestbooksByUuidAndCompanyId(
            uuid, companyId, QueryUtil.ALL_POS, QueryUtil.ALL_POS,
            new StagedModelModifiedDateComparator<Guestbook>());
    }
```

8. Override the `BaseStagedModelDataHandler`'s delete methods to leverage your newly created fetch method and custom local service:

```
@Override
public void deleteStagedModel(
        String uuid, long groupId, String className, String extraData)
    throws PortalException {

    Guestbook guestbook = fetchStagedModelByUuidAndGroupId(uuid, groupId);

    if (guestbook ≠ null) {
        deleteStagedModel(guestbook);
    }
}

@Override
public void deleteStagedModel(Guestbook guestbook)
    throws PortalException {

    _guestbookLocalService.deleteGuestbook(guestbook);
}
```

9. Organize your imports (*[CTRL]+[SHIFT]+O*), and save the file.

Your guestbook staged model data handler is ready to go! Next, you'll begin updating your guestbook's permissions to account for Staging.

40.4 Updating Permissions to Support Staging

```
<p>Enabling Staging and Export/Import<br>Step 4 of 7</p>
```

The guestbook's current permission handlers do not account for Staging. For example, the current configuration would display the *Add Guestbook* and *Add Entry* buttons on the live Site while Staging was enabled. These options should only be available on the staged Site when Staging is enabled.

First, edit the Guestbook app's permissions helper classes to provide permission checks to leverage when Staging is enabled.

1. Open the `GuestbookModelPermission` class residing in the `guestbook-service`'s `com.liferay.docs.guestbook.service.perm` package. Replace the `contains(...)` methods with those below:

```
public static boolean contains(
    PermissionChecker permissionChecker, long groupId, String actionId) {

    return contains(
        permissionChecker, RESOURCE_NAME, GuestbookPortletKeys.GUESTBOOK,
        groupId, actionId);
}

public static boolean contains(
    PermissionChecker permissionChecker, String name, long classPK,
    String actionId) {
```

```
    Group group = GroupLocalServiceUtil.fetchGroup(classPK);

    if ((group != null) && group.isStagingGroup()) {
        classPK = group.getLiveGroupId();
    }

    return permissionChecker.hasPermission(
        classPK, name, classPK, actionId);
}

public static boolean contains(
    PermissionChecker permissionChecker, String name, String portletId,
    long classPK, String actionId) {

    Boolean hasPermission = StagingPermissionUtil.hasPermission(
        permissionChecker, classPK, name, classPK, portletId, actionId);

    if (hasPermission != null) {
        return hasPermission.booleanValue();
    }

    return contains(permissionChecker, name, classPK, actionId);
}
```

This adds two additional contains methods. The original contains method now redirects to a new method that instantiates a `hasPermission` field using the staging permission checker. If it returns as a non-null value (i.e., the app is rendered on the staged Site), the appropriate boolean value is returned based on the user's permissions. If the `hasPermission` field is null (i.e., the app is rendered on the live Site), the third contains method is invoked, which calls the permission checker with group info from the live Site.

Now you'll edit the permissions helper classes for your two entities. These are for the model/resource permissions, so you supply the primary key of the entity you're checking permissions for (e.g., `guestbookId`).

2. Open the `EntryPermission` class residing in the `guestbook-service`'s `com.liferay.docs.guestbook.service.permission` package. In the `contains(PermissionChecker, Entry, String)` method, add this logic:

```
Boolean hasPermission = StagingPermissionUtil.hasPermission(
    permissionChecker, entry.getGroupId(), Entry.class.getName(),
    entry.getEntryId(), GuestbookPortletKeys.GUESTBOOK, actionId);

    if (hasPermission != null) {
        return hasPermission.booleanValue();
    }
```

If the new `hasPermission` field is returned as a non-null value (i.e., the app is rendered on the staged Site), the appropriate boolean value is returned based on the staging context.

3. Open the `GuestbookPermission` class and add the following code in the `contains(PermissionChecker, Guestbook, String)` method:

```
Boolean hasPermission = StagingPermissionUtil.hasPermission(
    permissionChecker, guestbook.getGroupId(),
    Guestbook.class.getName(), guestbook.getGuestbookId(),
    GuestbookPortletKeys.GUESTBOOK, actionId);

if (hasPermission != null) {
    return hasPermission.booleanValue();
}
```

This is similar to the logic added for the entry's permissions.

4. Organize your imports (*[CTRL]+[SHIFT]+O*) for all three classes, and then save them.

Your Guestbook app can now display the proper functionality depending on its staging context (i.e., staged Site or live Site).

The Guestbook's admin portlet requires additional modifications in its JSPs to display options correctly based on staging context.

1. In your `guestbook-web` module, open the `src/main/resources/META-INF/resources/guestbookadminportlet/guestbook_ac` file. Directly after the opening `<liferay-ui:icon-menu>` tag, add the following `if` statement:

```
<c:if test='<%= GuestbookPermission.contains(permissionChecker, guestbook.getGuestbookId(), ActionKeys.UPDATE) %>'>
```

   Close the `if` statement directly after the `<liferay-ui:icon />` tag. When completed, the `if` statement should look like this:

```
<c:if test='<%= GuestbookPermission.contains(permissionChecker, guestbook.getGuestbookId(), ActionKeys.UPDATE) %>'>
    <portlet:renderURL var="editURL">
        <portlet:param name="guestbookId"
            value="<%=String.valueOf(guestbook.getGuestbookId())%>" />
        <portlet:param name="mvcPath"
            value="/guestbookadminportlet/edit_guestbook.jsp" />
    </portlet:renderURL>

    <liferay-ui:icon image="edit" message="Edit"
        url="<%=editURL.toString()%>" />
</c:if>
```

   The new `if` statement hides the Guestbook's editing functionality if the user does not have the permissions to edit a guestbook. Since you added new staging permissions, those are verified too.

2. Below the `</c:if>` statement, add another `if` statement:

```
<c:if test='<%= GuestbookPermission.contains(permissionChecker, guestbook.getGuestbookId(), ActionKeys.DELETE) %>'>
```

   Close the `if` statement directly after the `<liferay-ui:icon-delete />` tag. When completed, the `if` statement should look like this:

```
<c:if test='<%= GuestbookPermission.contains(permissionChecker, guestbook.getGuestbookId(), ActionKeys.DELETE) %>'>
    <portlet:actionURL name="deleteGuestbook" var="deleteURL">
        <portlet:param name="guestbookId"
            value="<%=String.valueOf(guestbook.getGuestbookId())%>" />
    </portlet:actionURL>

    <liferay-ui:icon-delete url="<%=deleteURL.toString()%>" />
</c:if>
```

   Similar to the previous `if` statement, this one hides the Guestbook's delete functionality if the user does not have the permissions to delete a guestbook.

3. Open the `src/main/resources/META-INF/resources/guestbookadminportlet/view.jsp` file and wrap the `<aui:button-row />` tag with the following `if` statement:

```
<c:if test='<%= GuestbookModelPermission.contains(permissionChecker, scopeGroupId, "ADD_GUESTBOOK") %>'>
    ...
</c:if>
```

When finished, it should look like this:

```
<c:if test='<%= GuestbookModelPermission.contains(permissionChecker, scopeGroupId, "ADD_GUESTBOOK") %>'>
    <aui:button-row cssClass="guestbook-admin-buttons">
        <portlet:renderURL var="addGuestbookURL">
            <portlet:param name="mvcPath"
                value="/guestbookadminportlet/edit_guestbook.jsp" />
            <portlet:param name="redirect" value="<%="currentURL"%>" />
        </portlet:renderURL>

        <aui:button onClick="<%= addGuestbookURL.toString() %>"
            value="Add Guestbook" />
    </aui:button-row>
</c:if>
```

This `if` statement hides the Guestbook's *Add Guestbook* button if the user does not have the permissions to create a new guestbook.

Great! You've updated your Guestbook app's permissions to support Staging!

40.5 Configuring XStream

<p>Enabling Staging and Export/Import<br>Step 5 of 7</p>

Configuring XStream for your Guestbook app provides an easy way to customize how entities are serialized to XML and back again. You can use it to enhance the Guestbook's Staging implementation; however, it's not required. There are three ways to leverage Liferay's offering of XStream via the XStreamConfigurator:

- *Allowed Types:* whitelists entities so everything is forbidden except a certain set of items. All staged models are allowed by default; this would be in addition to the default functionality. Liferay DXP defines a default list of allowed types, which are available in the portlet data context.
- *Aliases:* helps with the readability and char length of LAR files by creating an alias for an otherwise long winded entity name.
- *Converters:* converts configured objects to and from XML. This is primarily used to protect sensitive data; when serialized this way, sensitive data cannot be extracted from the generated LAR.

Since Allowed Types don't make sense in this context (there are no additional entities to allow), you don't need them for Guestbook. Converters let you re-write the serialization process from scratch—for example, to add encryption. This is a tutorial on Staging, not serialization, so the default serialization implementation is fine. That leaves aliases.

In the Guestbook, you'll leverage XStream by creating an alias that modifies the XML in the LAR file produced by your app during the staging and export processes.

For example, by default your generated data has this structure:

```
<com.liferay.docs.guestbook.model.impl.GuestBookImpl>
    <field1>...</field1>
    ...
</com.liferay.docs.guestbook.model.impl.GuestBookImpl>
```

With an XStream alias configured in your app, that same LAR content has this structure:

```
<GuestBook>
    <field1>...</field1>
    ...
</GuestBook>
```

Follow the instructions below to create an XStream alias:

1. In the `guestbook-service` module, create a package named `com.liferay.docs.guestbook.xstream.configurator`. In that package, create a class named `GuestbookXStreamConfigurator`.

2. Modify the class to implement the `XStreamConfigurator` interface and create an `@Component` annotation declaring that same class as the implementation service:

   ```
   @Component(
       immediate = true,
       service = XStreamConfigurator.class
   )
   public class GuestbookXStreamConfigurator implements XStreamConfigurator {
   ```

3. Since the Guestbook won't leverage the *Allowed Types* and *Converters* XStream options, override their associated methods and have them return `null`:

   ```
   @Override
   public List<XStreamType> getAllowedXStreamTypes() {

       return null;
   }

   @Override
   public List<XStreamConverter> getXStreamConverters() {

       return null;
   }
   ```

4. Override the `getXStreamAliases()` method to return a list of aliases you want to define. Also, define the list field.

   ```
   @Override
   public List<XStreamAlias> getXStreamAliases() {
       return ListUtil.toList(_xStreamAliases);
   }

   private XStreamAlias[] _xStreamAliases;
   ```

   Next, you'll define the list.

5. Create an `activate()` method that defines the aliases to use for Guestbook's generated LAR file. You'll define an alias for the `GuestbookImpl` and `EntryImpl` classes to convert them from their full package names to simple entity names:

   ```
   @Activate
   protected void activate() {

       _xStreamAliases = new XStreamAlias[] {
           new XStreamAlias(GuestbookImpl.class, "Guestbook"),
           new XStreamAlias(EntryImpl.class, "Entry"),
       };
   }
   ```

6. Organize your imports (*[CTRL]+[SHIFT]+O*), and save the file.

Awesome! You implemented an XStream Configurator for the Guestbook and created an alias for your guestbook and entry entity declarations.

40.6 Defining System Events for Deletions

<p>Enabling Staging and Export/Import<br>Step 6 of 7</p>

The Staging framework tracks entity modifications in a few different ways. Actions like *adding* a guestbook or *editing* an entry are tracked automatically by the framework with the configuration of staged models and their data handlers. Entity deletions are handled slightly differently using system events.

For the Guestbook app, you must define system events for entity deletions so they're appropriately tracked by the Staging framework. If Staging can't track your entity deletions, they can't be managed on the staged Site, which means you can only delete entities from the live Site.

You must define your system events in your local services.

1. Open the `guestbook-service` module's `com.liferay.docs.guestbook.service.impl.GuestbookLocalServiceImpl` class and add the following `deleteGuestbook` methods. These override the default Service Builder generated methods the application has been using:

```
@Indexable(type = IndexableType.DELETE)
@Override
public Guestbook deleteGuestbook(long guestbookId)
    throws PortalException {

    Guestbook guestbook =
        guestbookPersistence.findByPrimaryKey(guestbookId);

    return guestbookLocalService.deleteGuestbook(guestbook);
}

@Indexable(type = IndexableType.DELETE)
@Override
@SystemEvent(type = SystemEventConstants.TYPE_DELETE)
public Guestbook deleteGuestbook(Guestbook guestbook) {

    return guestbookPersistence.remove(guestbook);
}
```

   These methods override the `GuestbookLocalServiceBaseImpl`'s `deleteGuestbook` methods. The major addition is the `@SystemEvent` annotation added to the `deleteGuestbook(Guestbook)` method. This notifies the Staging framework that a deletion system event occurs when the method is called.

   The `deleteGuestbook(long)` method is rerouted to call the `SystemEvent` tracked method, so all deletions are accounted for.

2. Ensure that the remaining `deleteGuestbook` method triggers the system event. Within the `deleteGuestbook(long, ServiceContext)` method, change

```
guestbook = deleteGuestbook(guestbook);
```

   to this

```
guestbook = guestbookLocalService.deleteGuestbook(guestbook);
```

Now all guestbook deletions are tracked via `SystemEvent`.

3. Add the import packages for the new `SystemEvent` annotation (*[CTRL]+[SHIFT]+O*) and then save the file.

4. You must apply the deletion system event to both entities. Open the `com.liferay.docs.guestbook.service.impl.EntryLoc` class and add similar delete methods:

```
@Indexable(type = IndexableType.DELETE)
@Override
public Entry deleteEntry(long entryId)
    throws PortalException {

    Entry entry = entryPersistence.findByPrimaryKey(entryId);

    return entryLocalService.deleteEntry(entry);
}

@Indexable(type = IndexableType.DELETE)
@Override
@SystemEvent(type = SystemEventConstants.TYPE_DELETE)
public Entry deleteEntry(Entry entry) {

    return entryPersistence.remove(entry);
}
```

This is the same logic as before, except the delete system event is now applied for the entry.

5. Make sure the remaining `deleteEntry` method triggers the system event. Within the `deleteEntry(long, ServiceContext)` method, change

```
entry = deleteEntry(entryId);
```

to this

```
entry = entryLocalService.deleteEntry(entry);
```

Now all entry deletions are tracked via `SystemEvent`.

6. Add the import packages for the new `SystemEvent` annotation (*[CTRL]+[SHIFT]+O*) and then save the file.

7. Run Service Builder by navigating to the Gradle Tasks pane on the right side of @ide@ and selecting your project's *build* → *buildService* task.

Your Guestbook app's deletions are now properly tracked by the Staging framework.

40.7 Creating the Portlet Data Handler

<p>Enabling Staging and Export/Import<br>Step 7 of 7</p>

A Portlet Data Handler imports/exports portlet-specific data to a LAR file. Its primary role is to query and coordinate between staged model data handlers. It also configures the Export/Import UI options for the application. For example, the Guestbook application's portlet data handler should

- import/export portlet-specific data pertaining to the Guestbook app
- track actions dealing with guestbook and entry entities (staged models)
- configure export/import UI options for the Guestbook app

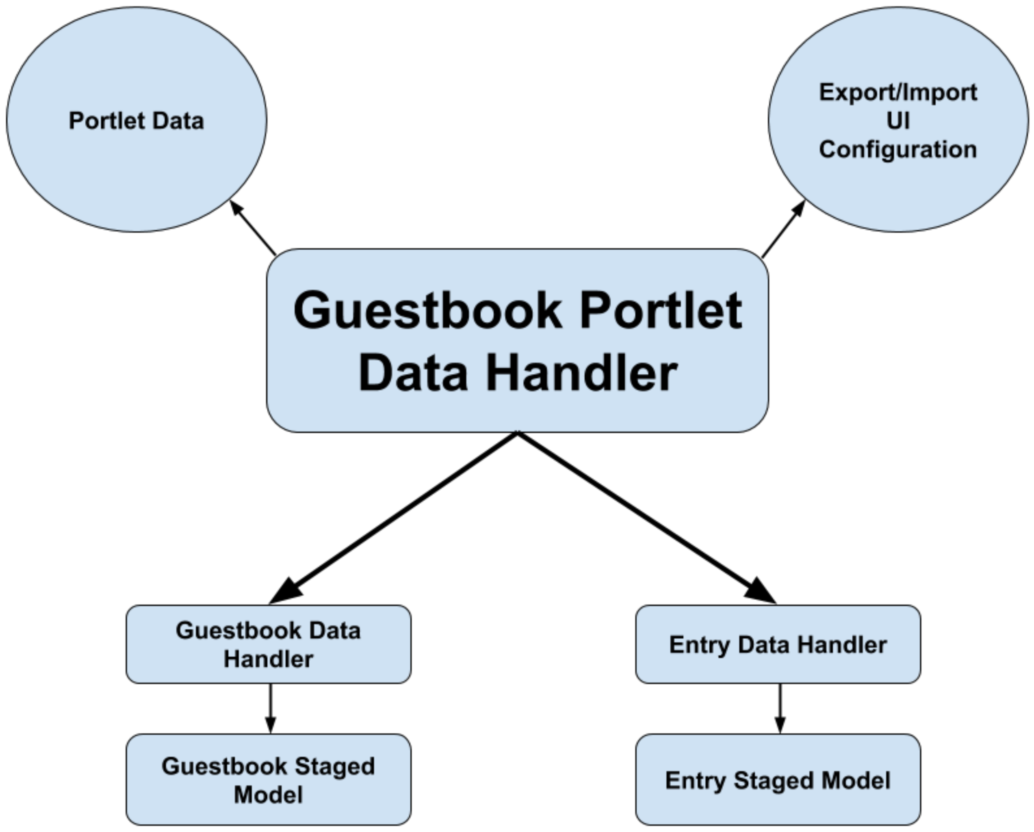


Figure 40.4: The Guestbook's portlet data handler must manage the portlet data, staged model data handlers, and UI configuration.

Follow the instructions below to create the Guestbook's portlet data handler.

1. In your `guestbook-service` module's `com.liferay.docs.guestbook.exportimport.data.handler` package, create the `GuestbookPortletDataHandler` class.

2. Extend the `BasePortletDataHandler` class and add the `@Component` annotation to the class declaration with several configured properties like this:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK,
        "javax.portlet.name=" + GuestbookPortletKeys.GUESTBOOK_ADMIN
    },
    service = PortletDataHandler.class
)
public class GuestbookPortletDataHandler extends BasePortletDataHandler {
```

The three set annotation attributes are described below:

- The `immediate` element tells the container to activate the component immediately once its provided module has started.
- The `property` element sets various properties for the component service. You must associate the portlets you wish to handle with this service so they function properly in the export/import environment. For example, since the Guestbook data handler is used for two portlets, they're both configured using the `javax.portlet.name` property.
- The `service` element should point to the `PortletDataHandler.class` interface.

3. Set what the portlet data handler controls and the portlet's Export/Import UI by adding an activate method:

```
@Activate
protected void activate() {

    setDeletionSystemEventStagedModelTypes(
        new StagedModelType(Entry.class),
        new StagedModelType(Guestbook.class));

    setExportControls(
        new PortletDataHandlerBoolean(
            NAMESPACE, "entries", true, false, null,
            Entry.class.getName()));

    setImportControls(getExportControls());
}
```

This method is called during initialization of the component by using the `@Activate` annotation. It's invoked after dependencies are set and before services are registered.

The three set methods called in the `GuestbookPortletDataHandler`'s `activate` method are described below:

- `setDeletionSystemEventStagedModelTypes`: sets the staged model deletions that the portlet data handler should track. For the Guestbook application, guestbooks and entries are tracked.
- `setExportControls`: adds fine grained controls over export behavior that are rendered in the Export UI. For the Guestbook application, a checkbox is added to select Guestbook content (entries) to export.
- `setImportControls`: adds fine grained controls over import behavior that are rendered in the Import UI. For the Guestbook application, a checkbox is added to select Guestbook content (entries) to import.

4. Set the entity local services you want to leverage in your portlet data handler:

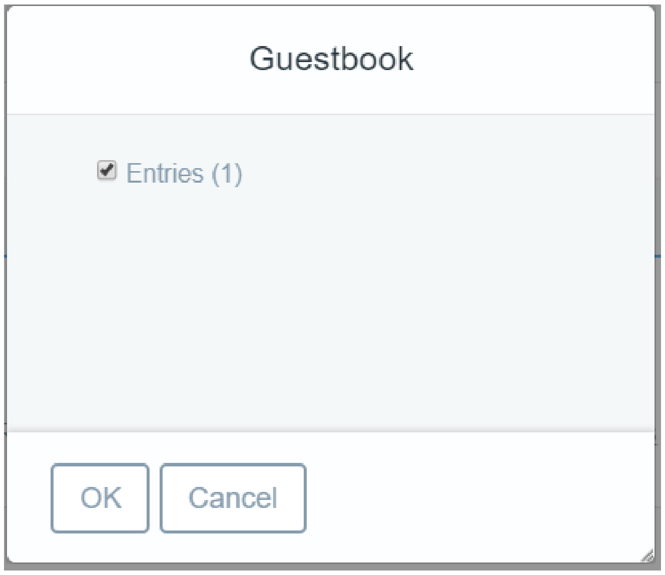


Figure 40.5: You can select the content types you'd like to export/import in the UI.

```
@Reference(unbind = "-")
protected void setGuestbookLocalService(
    GuestbookLocalService guestbookLocalService) {

    _guestbookLocalService = guestbookLocalService;
}

@Reference(unbind = "-")
protected void setEntryLocalService(EntryLocalService entryLocalService) {

    _entryLocalService = entryLocalService;
}

private GuestbookLocalService _guestbookLocalService;
private EntryLocalService _entryLocalService;
```

This provides access to the entry and guestbook's local services.

5. Create a namespace for your entities so the Export/Import framework can tell your application's entities from other entities in Liferay DXP.

```
public static final String NAMESPACE = "guestbook";
```

6. Your portlet data handler should retrieve the data related to its staged model entities so it can properly export/import them. Add this functionality by inserting the following methods:

```
@Override
protected String doExportData(
        final PortletDataContext portletDataContext, String portletId,
        PortletPreferences portletPreferences)
    throws Exception {

    Element rootElement = addExportDataRootElement(portletDataContext);

    if (!portletDataContext.getBooleanParameter(NAMESPACE, "entries")) {
        return getExportDataRootElementString(rootElement);
    }

    portletDataContext.addPortletPermissions(
        GuestbookModelPermission.RESOURCE_NAME);

    rootElement.addAttribute(
        "group-id", String.valueOf(portletDataContext.getScopeGroupId()));

    ActionableDynamicQuery guestbookActionableDynamicQuery =
        _guestbookLocalService.getExportActionableDynamicQuery(
            portletDataContext);
    guestbookActionableDynamicQuery.performActions();

    ActionableDynamicQuery entryActionableDynamicQuery =
        _entryLocalService.getExportActionableDynamicQuery(
            portletDataContext);
        entryActionableDynamicQuery.performActions();

    return getExportDataRootElementString(rootElement);
}

@Override
protected PortletPreferences doImportData(
        PortletDataContext portletDataContext, String portletId,
        PortletPreferences portletPreferences, String data)
    throws Exception {

    if (!portletDataContext.getBooleanParameter(NAMESPACE, "entries")) {
        return null;
    }

    portletDataContext.importPortletPermissions(
        GuestbookModelPermission.RESOURCE_NAME);

    Element guestbooksElement =
        portletDataContext.getImportDataGroupElement(Guestbook.class);

    List<Element> guestbookElements = guestbooksElement.elements();

    for (Element guestbookElement : guestbookElements) {
        StagedModelDataHandlerUtil.importStagedModel(
            portletDataContext, guestbookElement);
    }

    Element entriesElement =
        portletDataContext.getImportDataGroupElement(Entry.class);

    List<Element> entryElements = entriesElement.elements();

    for (Element entryElement : entryElements) {
        StagedModelDataHandlerUtil.importStagedModel(
            portletDataContext, entryElement);
    }

    return null;
}
```

The `doExportData` method first checks if anything should be exported. The `portletDataContext.getBooleanParameter(...`
method checks if the user selected Guestbook entries for export. Later, the `ActionableDynamicQuery` framework runs a query against guestbooks and entries to find ones which should be exported to the LAR file.

The `-ActionableDynamicQuery` classes are automatically generated by Service Builder and are available in your application's local service. It queries the database searching for certain Staging-specific parameters (e.g., `createDate` and `modifiedDate`), and based on those parameters, finds a list of exportable records from the staged model data handler.

The `doImportData` method queries for guestbook and entry data in the imported LAR file that should be added to the database. This is done by extracting XML elements from the LAR file by using utility methods from the `StagedModelDataHandlerUtil` class. The extracted elements tell Liferay DXP what data to import from the LAR file.

7. Add a method that counts the number of affected entities based on the current export or staging process:

```
@Override
protected void doPrepareManifestSummary(
        PortletDataContext portletDataContext,
        PortletPreferences portletPreferences)
    throws Exception {

    ActionableDynamicQuery entryExportActionableDynamicQuery =
        _entryLocalService.getExportActionableDynamicQuery(
            portletDataContext);
    entryExportActionableDynamicQuery.performCount();

    ActionableDynamicQuery guestbookExportActionableDynamicQuery =
        _guestbookLocalService.getExportActionableDynamicQuery(
            portletDataContext);
    guestbookExportActionableDynamicQuery.performCount();
}
```

   This number is displayed in the Export and Staging UI. Note that since the Staging framework traverses the entity graph during export, the built-in components provide an approximate value in some cases.

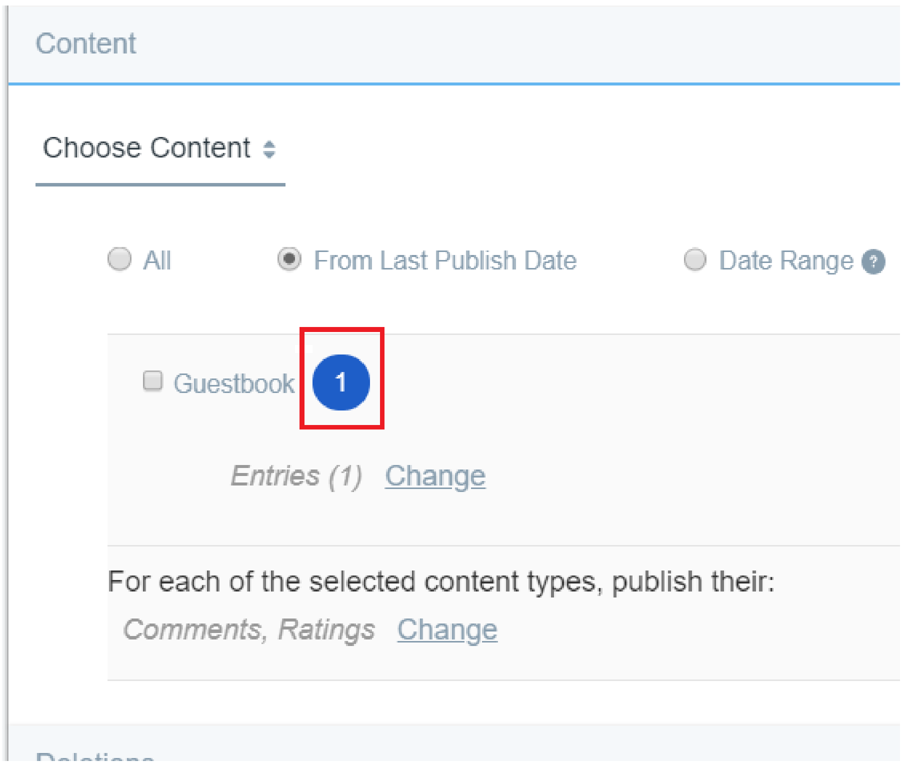
8. Organize your imports (*[CTRL]+[SHIFT]+O*), and save the file. **Hint:** Be sure to choose the `javax.portlet.PortletPreferences` import package.

Excellent! You've set up your Guestbook's portlet data handler and can now handle your portlet's data and control its staged model data handlers.

Your Guestbook app is now leveraging the Staging and Export/Import frameworks! To verify this, when you go to enable Staging, you can now enable it for your Guestbook app.

You can also navigate to the Guestbook Admin portlet and manage Staging from the Options menu. This menu also offers a way to export and import Guestbook LAR files manually.

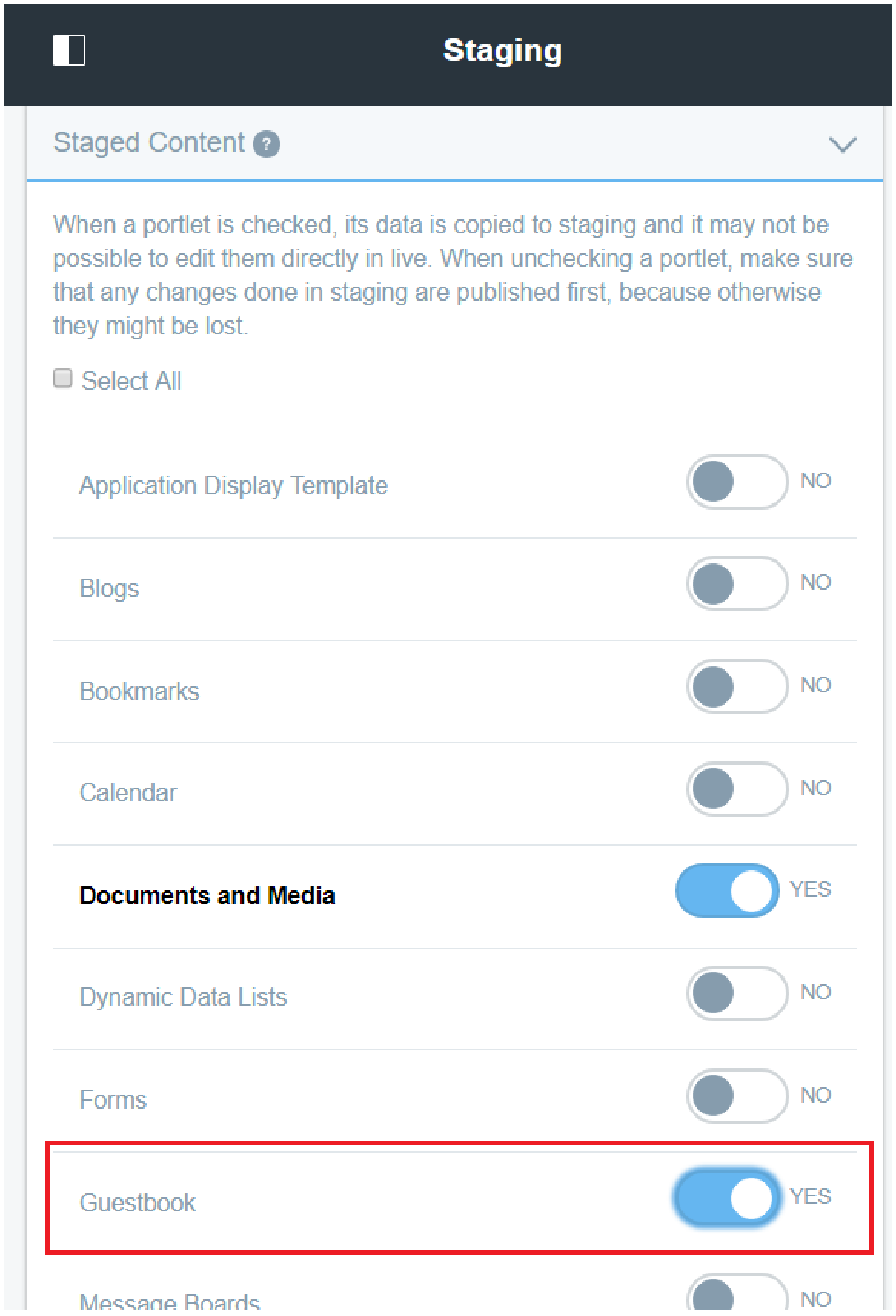
The Guestbook is ready for the staging process!



For each of the selected content types, publish their:

Comments, Ratings Change

Figure 40.6: The number of modified Guestbook entities are displayed in the Export UI.



Staging

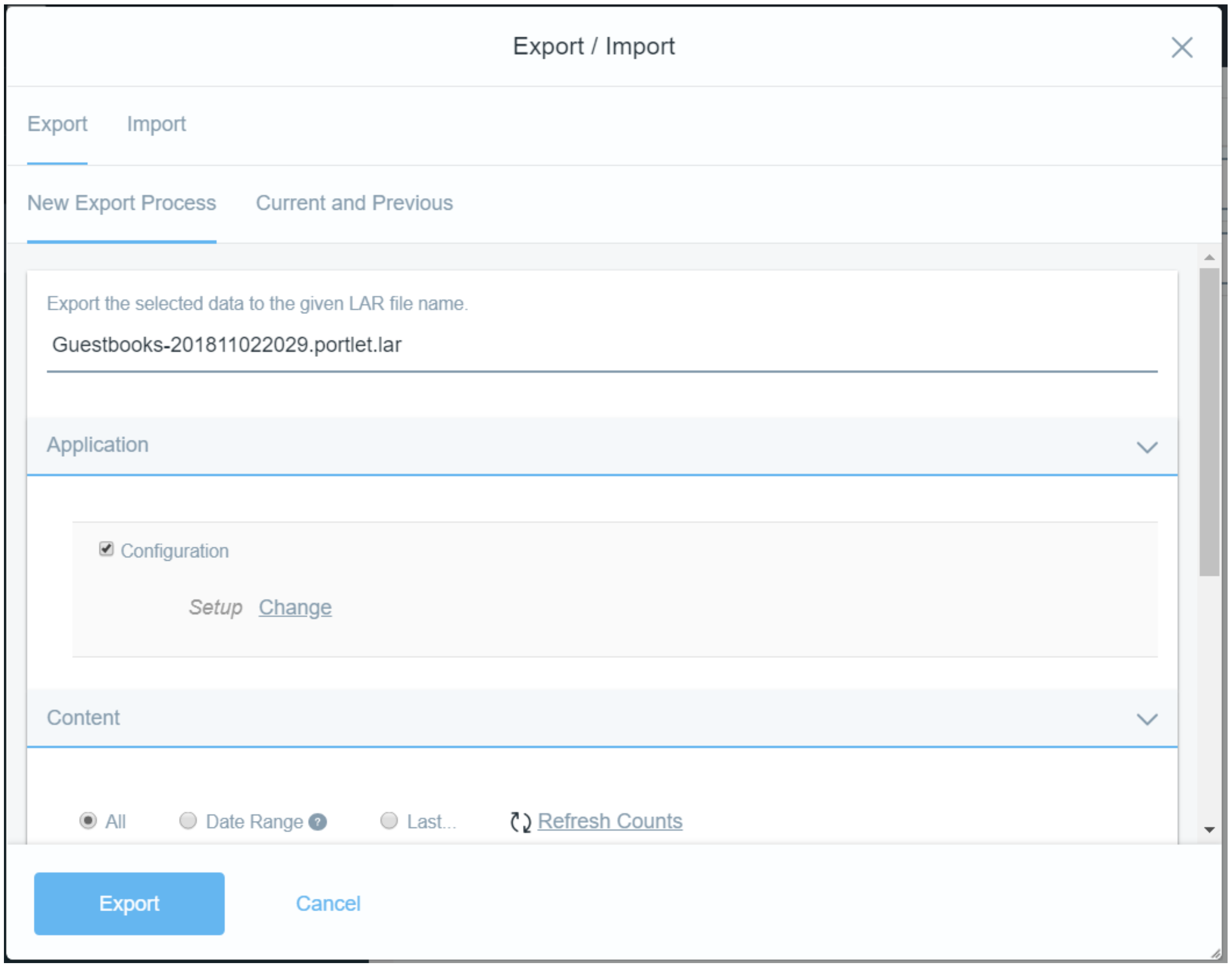
Staged Content

When a portlet is checked, its data is copied to staging and it may not be possible to edit them directly in live. When unchecking a portlet, make sure that any changes done in staging are published first, because otherwise they might be lost.

Select All

Application Display Template	NO
Blogs	NO
Bookmarks	NO
Calendar	NO
Documents and Media	YES
Dynamic Data Lists	NO
Forms	NO
Guestbook	YES
Message Boards	NO

Figure 40.7: Enable the Guestbook Staging functionality.


Export / Import

Export Import

New Export Process Current and Previous

Export the selected data to the given LAR file name.

Guestbooks-201811022029.portlet.lar

Application

☑ Configuration

Setup Change

Content

● All ○ Date Range ○ Last... Refresh Counts

Export Cancel

Figure 40.8: You can manually export and import Guestbook LAR files from the Guestbook Admin portlet.

Chapter 41

Writing an Android App for Liferay DXP

Users expect to access Liferay DXP content from their mobile devices. As an intrepid developer, you naturally want to turn these expectations into reality. Thankfully, Liferay provides a way for your mobile apps to access Liferay DXP content and applications with Liferay Screens! Screens contains native components called *Screenlets* that can call Liferay DXP's remote services and display the results in your app. Each Screenlet comes complete with its own fully pluggable UI that you can customize to your liking. Although the Screenlets included with Screens only work with Liferay DXP's built-in remote services, you can write your own Screenlets that work with your custom portlets' remote services.

If you're an experienced Android developer but need a start-to-finish guide on how to integrate Android apps with Liferay DXP, you're in the right place. This Learning Path walks you through the creation of an Android app that interacts with the Guestbook portlet developed in the Developing a Web Application Learning Path. Since this is a custom portlet, you'll write your own Screenlets that let your app retrieve and display guestbooks and their entries.

You should note that although this Learning Path provides complete code snippets of the app, not every aspect of Android development is explained in detail. Focus is instead placed on the code that leverages Liferay Screens. Therefore, you **must** have significant Android development experience before attempting this Learning Path. Otherwise, you'll likely be confused. Google provides extensive documentation of the Android APIs as well as some basic tutorials on developer.android.com.

Experience in Android development is all you need to start working. You needn't have completed the Liferay MVC Learning Path to obtain a working Guestbook portlet. The complete Guestbook portlet's modules are provided for installation into your local Liferay DXP instance.

Now that you know what you'll be doing here, it's time to move on to the first series of articles: Beginning Android Development for Liferay DXP. These articles walk you through the steps required to get started developing an Android app that interacts with Liferay DXP.

Chapter 42

Beginning Android Development for Your Liferay DXP Instance

Getting started with Liferay Screens for Android is a straightforward process. This series of Learning Path articles walks you through creating an Android app and preparing it to work with the Guestbook portlet developed in the Developing a Web Application Learning Path.

Since Liferay Screens uses the Liferay Mobile SDK to make remote service calls, you'll build a Mobile SDK capable of calling the Guestbook portlet's remote services (the Guestbook Mobile SDK). You'll then install this Mobile SDK and Screens into your Android project. You'll also learn about the Android app's design and implement authentication with Login Screenlet.

This section of the Learning Path covers these topics:

1. Setting up the Guestbook portlet
2. Building the Guestbook Mobile SDK
3. Creating the Android project
4. Installing the Guestbook Mobile SDK and Liferay Screens in the Android project
5. Designing Your App
6. Using Login Screenlet for Authentication

When you finish, you'll be ready to start developing your first Screenlet.

42.1 Setting up the Guestbook Portlet

Before you begin developing the Guestbook app for Android, you must set up the Guestbook portlet in a Liferay DXP instance. To do this, follow these steps:

1. Install JDK 8
2. Install and Configure a Local Liferay DXP bundle
3. Deploy the Guestbook Portlet to the Local Liferay DXP Instance

Installing the JDK

To get started, you must have JDK 8 installed. You can download and install the Java SE JDK from the Java downloads page. This page also has links to the JDK installation instructions.

Installing and Configuring a Local Liferay DXP Bundle

First, download a Liferay DXP Tomcat bundle from liferay.com. Then click here and follow the instructions to install the bundle. To follow Liferay DXP best practices, you should create a `bundles` folder and unzip it there. The bundle's root folder is referred to as *Liferay Home* and is named according to the version, edition, and specific Liferay DXP release. For example, if you downloaded Liferay Portal 7.0 CE GA4 and unzipped it to a `bundles` folder on your machine, that bundle's Liferay Home folder path is:

```
bundles/liferay-ce-portal-7.0-ga4
```

Now you're ready to start Liferay DXP! Start the bundle as described in the link above. If you're on Mac or Linux you should also run `tail -f ../logs/catalina.out` immediately after the `./startup.sh` command executes. This ensures that the server log prints to the terminal. This step isn't necessary on Windows because the server log automatically opens in another window.

After a minute or two, Liferay DXP starts up and automatically takes you to its initial setup page at http://localhost:8080. On this page, you need to provide some basic information about how to set up your Liferay DXP instance. Enter a name for your instance, select the default language, and then uncheck the *Add Sample Data* box. Then enter the first name, last name, and email address of the default administrative user. For the purposes of this Learning Path, these don't have to be real. If you want to connect Liferay DXP to a separate database such as MySQL or PostgreSQL, you can configure that connection here. Note that although the default embedded database is fine for development on your local machine, it isn't optimized for production. Click *Finish Configuration* when you're done. Then accept the terms of use and set a password and a password reminder query for your administrative user. Your Liferay DXP instance then takes you to its default site.

Great! Next, you'll deploy the Guestbook portlet to your Liferay DXP instance.

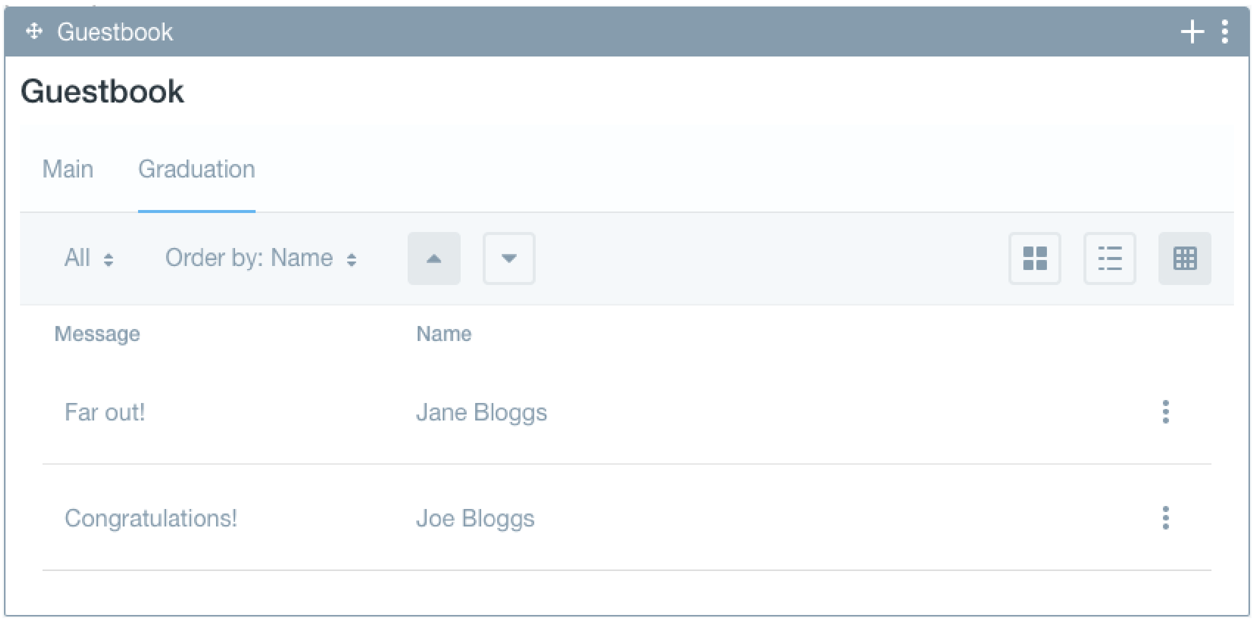
Deploying the Guestbook Portlet

Now that your Liferay DXP instance is set up, you can deploy the Guestbook portlet to it. First, click here to download the Guestbook portlet's modules:

- `com.liferay.docs.guestbook.api-1.0.0.jar`
- `com.liferay.docs.guestbook.portlet-1.0.0.jar`
- `com.liferay.docs.guestbook.service-1.0.0.jar`
- `com.liferay.docs.guestbook.service-wsdd-1.0.0.jar`

Place these modules in your Liferay DXP instance's `[Liferay Home]/deploy` folder. You should then see console messages indicating that the modules have successfully deployed and started. On your Liferay DXP instance's default site, click the *Add* button (➕) on the upper-right corner of the screen. Then click the *Applications → Sample* category and drag *Guestbook* onto the page. The Guestbook portlet should now appear with the default guestbook (Main). In the portlet, add a new guestbook and an entry or two each from the *Add* menu (⊕) that appears in the top right of the portlet's border when you mouse over the portlet. When you create the Guestbook Android app, this ensures there's some content to display in it. The Guestbook portlet on your site should now look like this:

Stupendous! You've successfully set up a Liferay DXP instance and added the Guestbook portlet to it. Now you're ready to get started with the Liferay Mobile SDK.



Message	Name
Far out!	Jane Bloggs
Congratulations!	Joe Bloggs

Figure 42.1: The Guestbook portlet, with a new guestbook and some entries.

42.2 Building the Guestbook Mobile SDK

Once you've deployed the Guestbook portlet, you're ready to build the Guestbook Mobile SDK. You might be asking yourself, "Why do I have to build a separate Mobile SDK? Can't I just use the pre-built Mobile SDK that Liferay already provides?" Fantastic question! The reason is that Liferay's pre-built Mobile SDK doesn't have the classes it needs to call the Guestbook portlet's remote services. The pre-built Mobile SDK includes only the framework necessary to make server calls to the remote services of Liferay DXP's *core* apps. Core apps (also referred to as *out-of-the-box* apps) are those included with every Liferay DXP instance. Since you're calling services of an app the default Mobile SDK doesn't know about (the Guestbook portlet), you must build a Mobile SDK that can call its services. Now put on your hard hat, because it's time to get building!

Building the Mobile SDK

In the Mobile SDK source code, Liferay provides a Mobile SDK Builder that you can use to build your own Mobile SDKs. For the builder to generate the classes that can call a non-core app's remote services, those services must be available and accompanied by a Web Service Deployment Descriptor (WSDD). To learn how the Guestbook portlet's remote services and WSDD were generated, see the section Generating Web Services in the web application Learning Path. Since the Guestbook portlet's web services already exist, you don't need to generate them. Just remember that you must generate web services when developing your own portlets.

To build the Guestbook Mobile SDK, first download the Mobile SDK's source code by clicking here. Unzip the file to a location on your machine where you want the Mobile SDK to reside. This location is purely personal preference; the builder works the same no matter where you put the Mobile SDK's source code. Once unzipped, the Mobile SDK's source code is in the `liferay-mobile-sdk-android-7.0.6` folder.

Now you're ready to build the Guestbook Mobile SDK. The builder contains a convenient command line wizard to assist you in building Mobile SDKs. To start it, navigate to the `liferay-mobile-sdk-android-7.0.6` folder and run the following command:

```
./gradlew createModule
```

The wizard launches and asks you to enter your project's properties. You must first provide the `Context` property. This is the context path of the remote services the builder will generate classes and methods for. To view your Liferay DXP instance's remote service context paths, go to http://localhost:8080/api/jsonws. On the page's upper left, there's a menu for selecting the context path. Select *gb*, which is the Guestbook portlet's context path. The UI updates to show only the remote services available in the selected context path. Return to the terminal and enter `gb` for the `Context` property.

Next, the wizard needs the `Package Name` property. This is the package path for the classes the builder generates. Accept the default value of `com.liferay.mobile.android`. The wizard then asks for the `POM Description` property. Technically, you only need this if you want to publish your Mobile SDK to Maven. Since the builder requires it, however, enter `Guestbook SDK`. The following screenshot shows these properties entered in the wizard:

Once you enter the final property, the builder runs and generates a `BUILD SUCCESSFUL` message. Now that the builder contains a `gb` module, you must generate that module's remote services. To do this, first navigate to the following folder:

```
liferay-mobile-sdk-android-7.0.6/modules/gb
```

Then run the following command:

```
../../gradlew generate
```

As before, the builder runs and generates a `BUILD SUCCESSFUL` message. Great! You're probably wondering what just happened, though. The builder generated the source classes you'll use in your Android app to interact with the Guestbook portlet. You can find these source classes in the following folder of the Mobile SDK's source code:

```
modules/gb/android/src/gen/java
```

Also note that the source classes are in the package path you specified when generating the module, with an additional folder that denotes the Liferay DXP version they work with. The full path to the generated source classes is therefore:

```
modules/gb/android/src/gen/java/com/liferay/mobile/android/v7
```

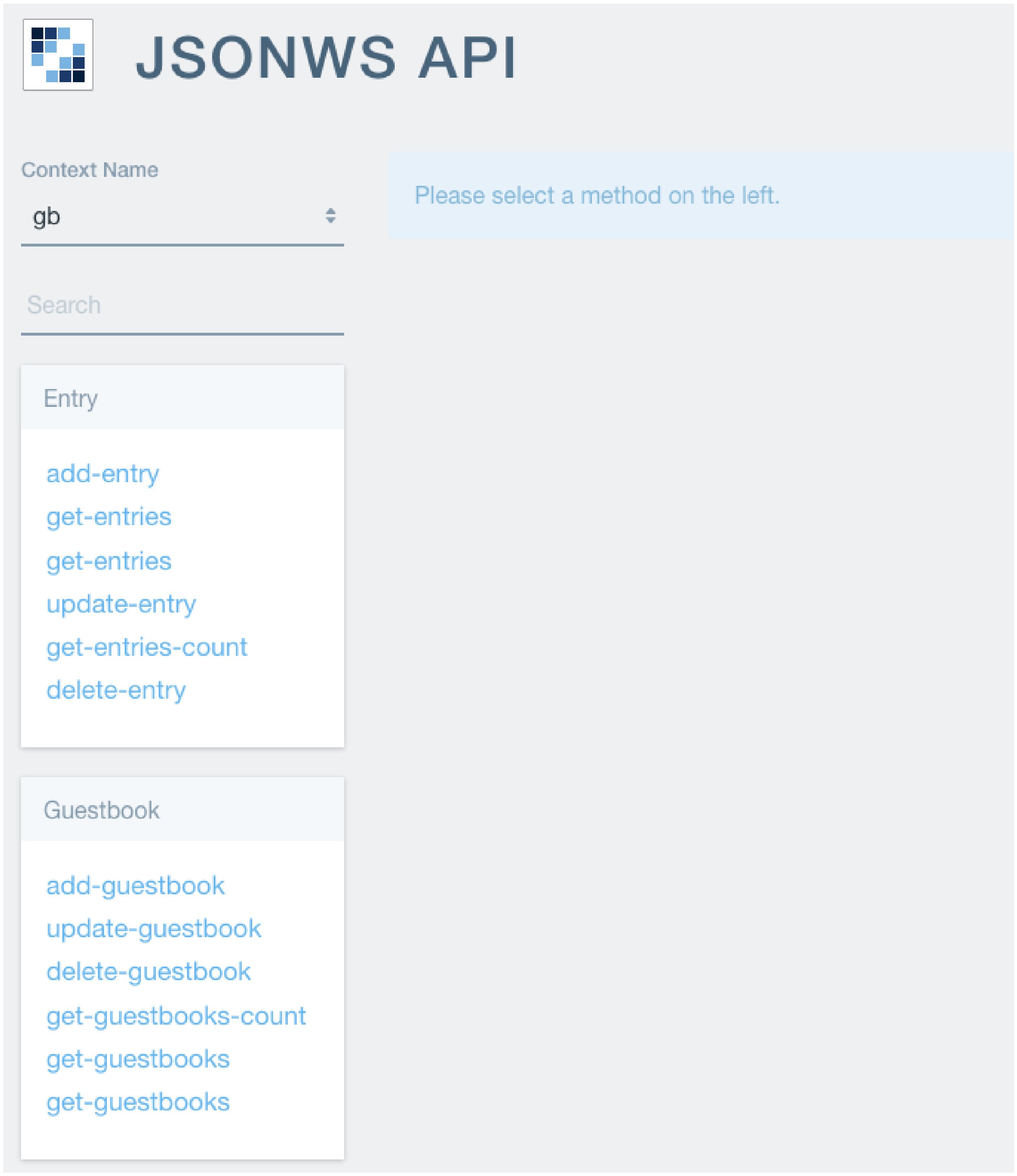
This folder has two subfolders that correspond to each entity in the Guestbook portlet: `guestbook` and `entry`. Each subfolder contains that entity's source class, `GuestbookService` and `EntryService`, respectively.

There's one last thing to do before you can use these classes in your Android app: put them in a JAR file. To do this, make sure you're still in the `modules/gb` folder on the command line and run `../../gradlew jar`. This command does two things:

1. Generates a JAR file in `modules/gb/build/libs` that contains the Guestbook portlet's service classes. This JAR file is the Guestbook Mobile SDK.

2. Generates a custom-built version of Liferay's pre-built Mobile SDK in `liferay-mobile-sdk-android-7.0.6/android/build/libs`.

Congratulations! You just built the Guestbook Mobile SDK. Now that's an accomplishment worth writing in a guestbook. All you need now is an Android app in which to install this Mobile SDK. The next article shows you how to create this.


JSONWS API

Context Name

gb

Search

Entry

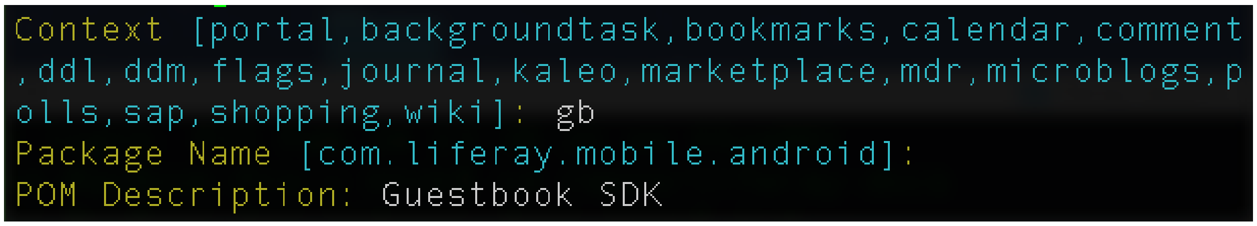
- add-entry
- get-entries
- get-entries
- update-entry
- get-entries-count
- delete-entry

Guestbook

- add-guestbook
- update-guestbook
- delete-guestbook
- get-guestbooks-count
- get-guestbooks
- get-guestbooks

Please select a method on the left.

Figure 42.2: The Guestbook Portlet's context path (gb) on the server.



```
Context [portal,backgroundtask,bookmarks,calendar,comment
,ddl,ddm,flags,journal,kaleo,marketplace,mdr,microblogs,p
olls,sap,shopping,wiki]: gb
Package Name [com.liferay.mobile.android]:
POM Description: Guestbook SDK
```

Figure 42.3: To build your Mobile SDK, you must enter values for the `Context`, `Package Name`, and `POM Description` properties. The blue values in square brackets are defaults.

42.3 Creating the Android Project

Now that you've built the Guestbook Mobile SDK, you're ready to create the Guestbook Android app. This article walks you through the steps required to create the app's project in Android Studio. After this, you'll be ready to install the Guestbook Mobile SDK and Liferay Screens. First though, you should make sure you've installed Android's development tools.

Installing Android Studio

This Learning Path uses Android Studio–Android's official IDE–to develop the Guestbook app. As an Android developer, you're likely very familiar with Android Studio. If you need help setting up and using Android Studio, see the following topics in Android Studio's documentation:

- Android Studio Installation Instructions
- Meet Android Studio

Once Android Studio is up and running, you're ready to create the Guestbook app!

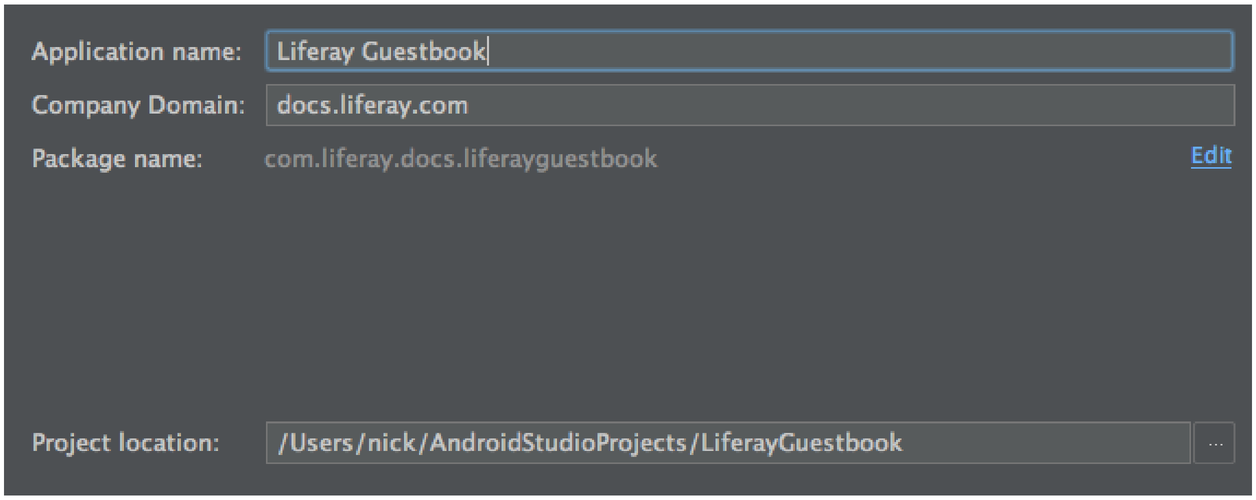
Creating the Guestbook App

When you start Android Studio, it presents a welcome screen containing a Quick Start menu. Click *Start a new Android Studio project* in this menu. This launches the *Create New Project* wizard, which asks you to enter the app name, company domain, and project location. Enter *Liferay Guestbook* as the app name and *docs.liferay.com* as the company domain. Android Studio uses these values to autofill your app's package name and project location. Accept the package name `com.liferay.docs.liferayguestbook`, and choose a project location that's convenient. Click *Next*.

The next screen asks you to specify your app's supported form factors and minimum Android SDK. Make sure that only the *Phone and Tablet* checkbox is selected. In the *Minimum SDK* menu, select *API 15: Android 4.0.3 (IceCreamSandwich)*. Android Studio gives you an estimate of the percentage of devices active on the Google Play store that can run the selected API level. You can view a graphical representation of these estimates by clicking the *Help me choose* link in the text below the Minimum SDK menu. Click *Next* when you're finished.

You must now specify your app's first activity. Although you'll use this activity to authenticate users to your Liferay DXP instance, don't select Login Activity. Select *Empty Activity* instead. Later, you'll insert Login Screenlet in this activity. Login Screenlet contains everything your users need to authenticate to a Liferay DXP instance, including the UI. Click *Next*.

The final screen of the New Project Wizard asks you to enter the activity's name and the name of its layout file. Accept the defaults and click *Finish*.

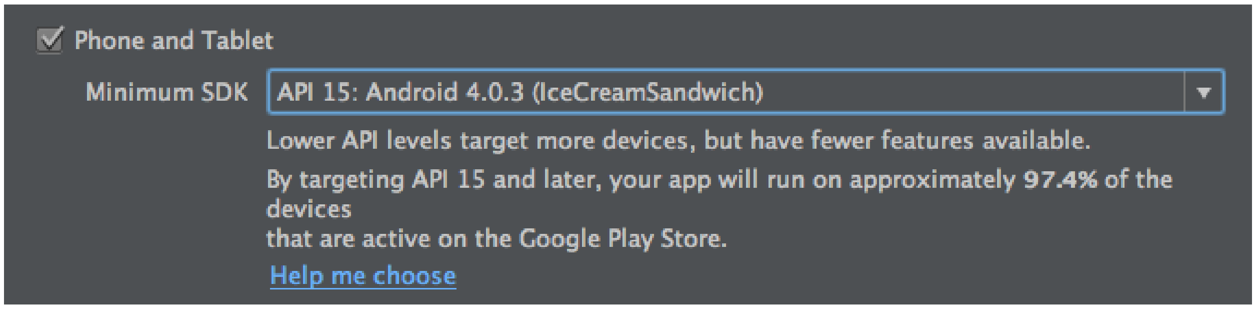

Application name: Liferay Guestbook

Company Domain: docs.liferay.com

Package name: com.liferay.docs.liferayguestbook

Project location: /Users/nick/AndroidStudioProjects/LiferayGuestbook

Figure 42.4: The first screen of Android Studio's Create New Project wizard asks you to enter your app's name and company domain.


☑ Phone and Tablet

Minimum SDK: API 15: Android 4.0.3 (IceCreamSandwich)

Lower API levels target more devices, but have fewer features available.

By targeting API 15 and later, your app will run on approximately **97.4%** of the devices that are active on the Google Play Store.

Figure 42.5: The second screen of Android Studio's Create New Project wizard lets you select your app's form factors and minimum Android API level.

A progress indicator appears that indicates your project is building. Android Studio then opens the project with the activity's class and layout ready to edit. The project's structure appears on the left side of the screen.

Well done! You successfully created the Guestbook app's project. Now it's time to put the Guestbook Mobile SDK and Liferay Screens to work!

42.4 Installing the Guestbook Mobile SDK and Liferay Screens for Android

For your Android app to interact with the Guestbook portlet, you must install the following libraries in your Android project:

- **Liferay's pre-built Mobile SDK:** This Mobile SDK contains the classes that call Liferay DXP's core remotes services. It also contains the framework necessary for any Mobile SDK to make server calls.
- **The Guestbook Mobile SDK:** This Mobile SDK contains only the classes that call the Guestbook portlet's remote services.

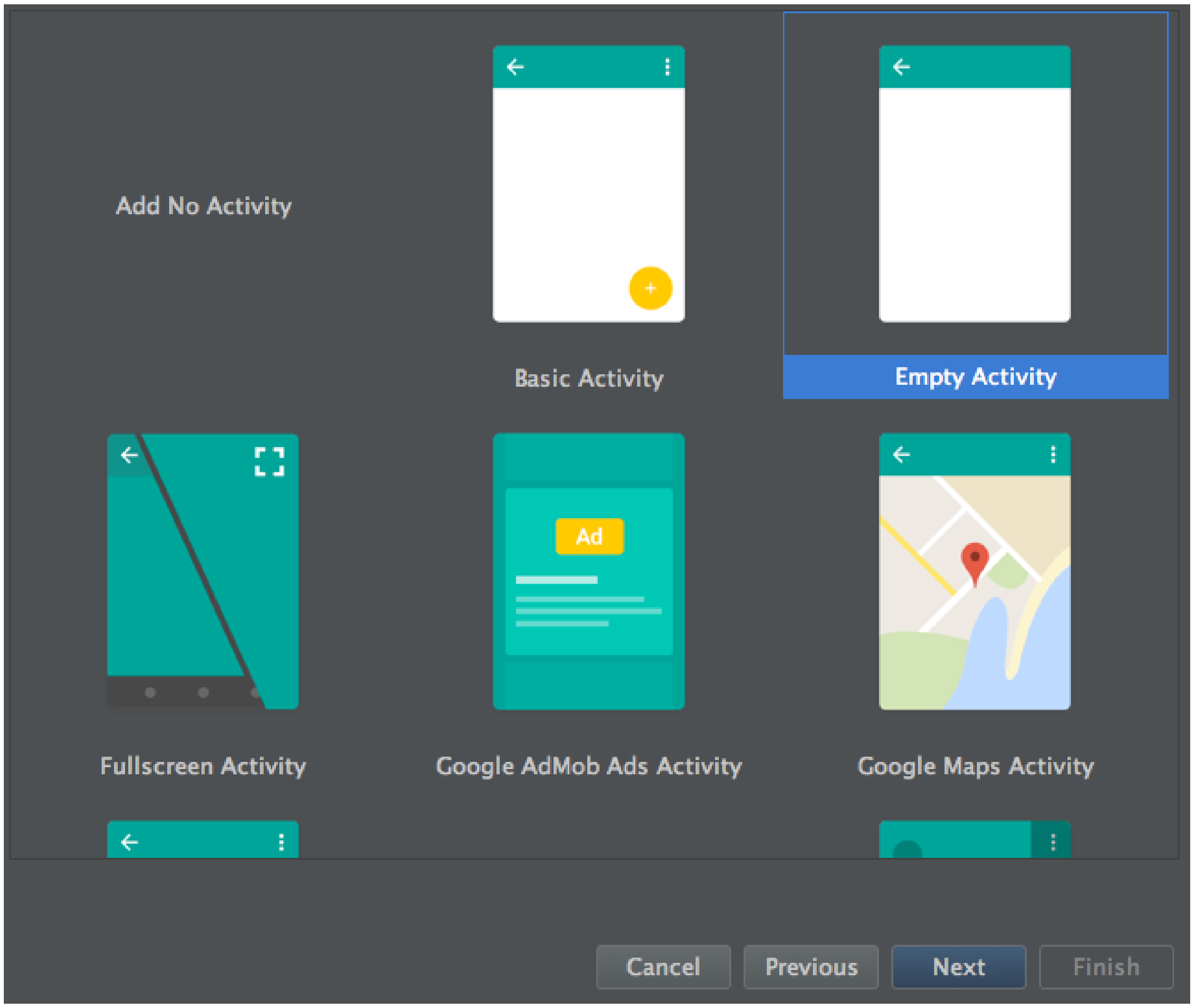
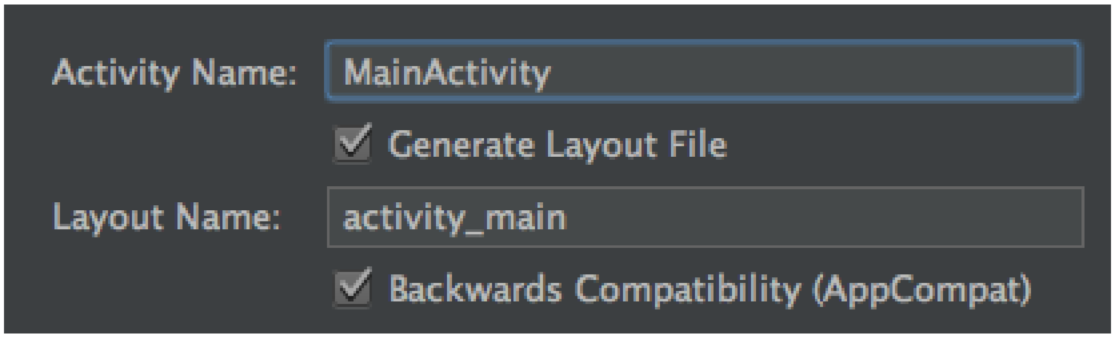


Figure 42.6: The third screen of Android Studio's Create New Project wizard lets you specify an activity for your app.



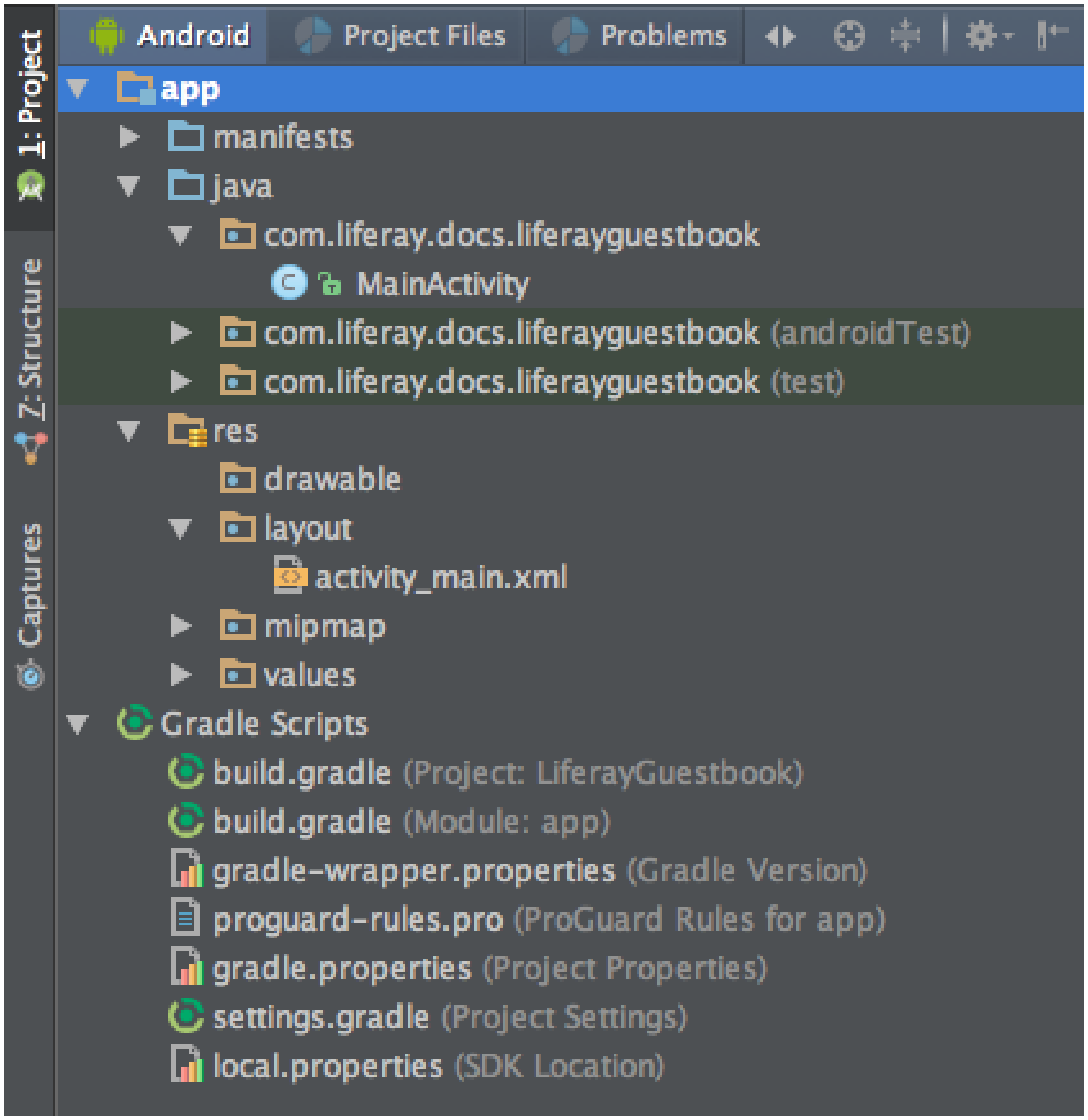
Activity Name: MainActivity

☑ Generate Layout File

Layout Name: activity_main

☑ Backwards Compatibility (AppCompat)

Figure 42.7: In the final screen of Android Studio's Create New Project wizard, accept the default values for the activity and layout name.



```
app
    manifests
    java
        com.liferay.docs.liferayguestbook
            MainActivity
        com.liferay.docs.liferayguestbook (androidTest)
        com.liferay.docs.liferayguestbook (test)
    res
        drawable
        layout
            activity_main.xml
        mipmap
        values
Gradle Scripts
    build.gradle (Project: LiferayGuestbook)
    build.gradle (Module: app)
    gradle-wrapper.properties (Gradle Version)
    proguard-rules.pro (ProGuard Rules for app)
    gradle.properties (Project Properties)
    settings.gradle (Project Settings)
    local.properties (SDK Location)
```

Figure 42.8: Android Studio shows your project's structure.

- **Liferay Screens:** Screens contains the Screenlet framework and several built-in Screenlets like Login Screenlet. Because these built-in Screenlets interact with Liferay DXP's core apps, they make their server calls with Liferay's pre-built Mobile SDK. Note that all Screenlets, including those that make server calls with a custom-built Mobile SDK, must use the framework in Liferay's pre-built Mobile SDK to issue their calls.

Since Liferay's pre-built Mobile SDK is a dependency of Liferay Screens, installing Screens automatically installs this Mobile SDK. You must, however, install the Guestbook Mobile SDK manually.

This article walks you through the installation of the Guestbook Mobile SDK and Liferay Screens. When you finish, you'll be ready to start developing the app.

Installing the Guestbook Mobile SDK

The Mobile SDK Builder generated two separate JAR files in your `liferay-mobile-sdk-android-7.0.6` folder:

1. `modules/gb/build/libs/liferay-gb-android-sdk-1.0.jar`
2. `android/build/libs/liferay-android-sdk-7.0.6.jar`

The first JAR file is the Guestbook Mobile SDK. The second JAR file is a custom built version of Liferay's pre-built Mobile SDK. Because Screens includes the pre-built Mobile SDK, you don't need to install the second JAR file. You must, however, install the first JAR file. To do so, copy `liferay-gb-android-sdk-1.0.jar` into your app's `app/libs` folder (the default location for your Android app is `AndroidStudioProjects/LiferayGuestbook`). That's it! Next, you'll install Liferay Screens.

Installing Liferay Screens

To install Liferay Screens, you must edit your app's `build.gradle` file. Note that your project has two `build.gradle` files: one for the project, and another for the app module. You can find them under *Gradle Scripts* in your Android Studio project. This screenshot highlights the app module's `build.gradle` file:

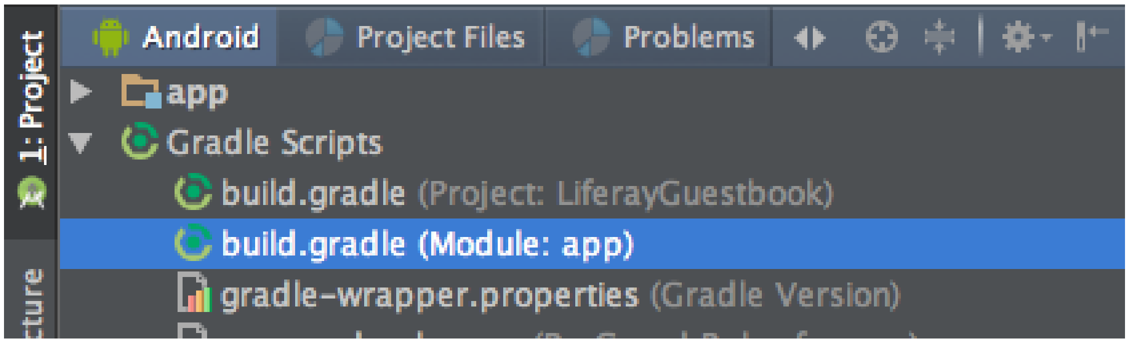

1: Project
Android
Project Files
Problems
app
Gradle Scripts
build.gradle (Project: LiferayGuestbook)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)

Figure 42.9: The app module's `build.gradle` file.

In the app module's `build.gradle` file, add the following lines of code inside the `dependencies` element:

```
compile 'com.liferay.mobile:liferay-screens:2.1.1'
compile 'com.liferay.mobile:liferay-material-viewset:2.1.1'
```

This adds the `liferay-screens` and `liferay-material-viewset` dependencies. Although only the `liferay-screens` dependency is necessary to install Screens, adding other View Sets, like the Material View Set, gives you flexibility when designing your app's look and feel. Click here for more information on Views in Liferay Screens.

Once you edit build.gradle, a message appears at the top of the file that asks you to sync your app with its Gradle files. Syncing with the Gradle files is required to incorporate any changes you make to them. Syncing also downloads and installs any new dependencies, like those you just added. Sync the Gradle files now by clicking the *Sync Now* link in the message.

Note that after syncing, your build.gradle may show an error similar to this:

```
All com.android.support libraries must use the exact same version specification...
```

If this occurs, you must manually add the correct version of the com.android.support dependencies. For example, the app in this Learning Path currently uses version 25.3.1 of the com.android.support libraries. This requires that you manually add the following dependencies to the app's build.gradle:

```
compile 'com.android.support:support-v4:25.3.1'
compile 'com.android.support:recyclerview-v7:25.3.1'
compile 'com.android.support:transition:25.3.1'
compile 'com.android.support:design:25.3.1'
compile 'com.android.support:exifinterface:25.3.1'
```

After adding these inside the dependencies element, click *Sync Now* again. The error message should be gone once the sync completes.

Great! Now you're ready to test your Screens and Mobile SDK installations.

Verifying the Installations

To check your Screens and Mobile SDK installations, first open your project's MainActivity class in Android Studio. It's in the java folder's com.liferay.docs.liferayguestbook package. Then add the following imports to this file:

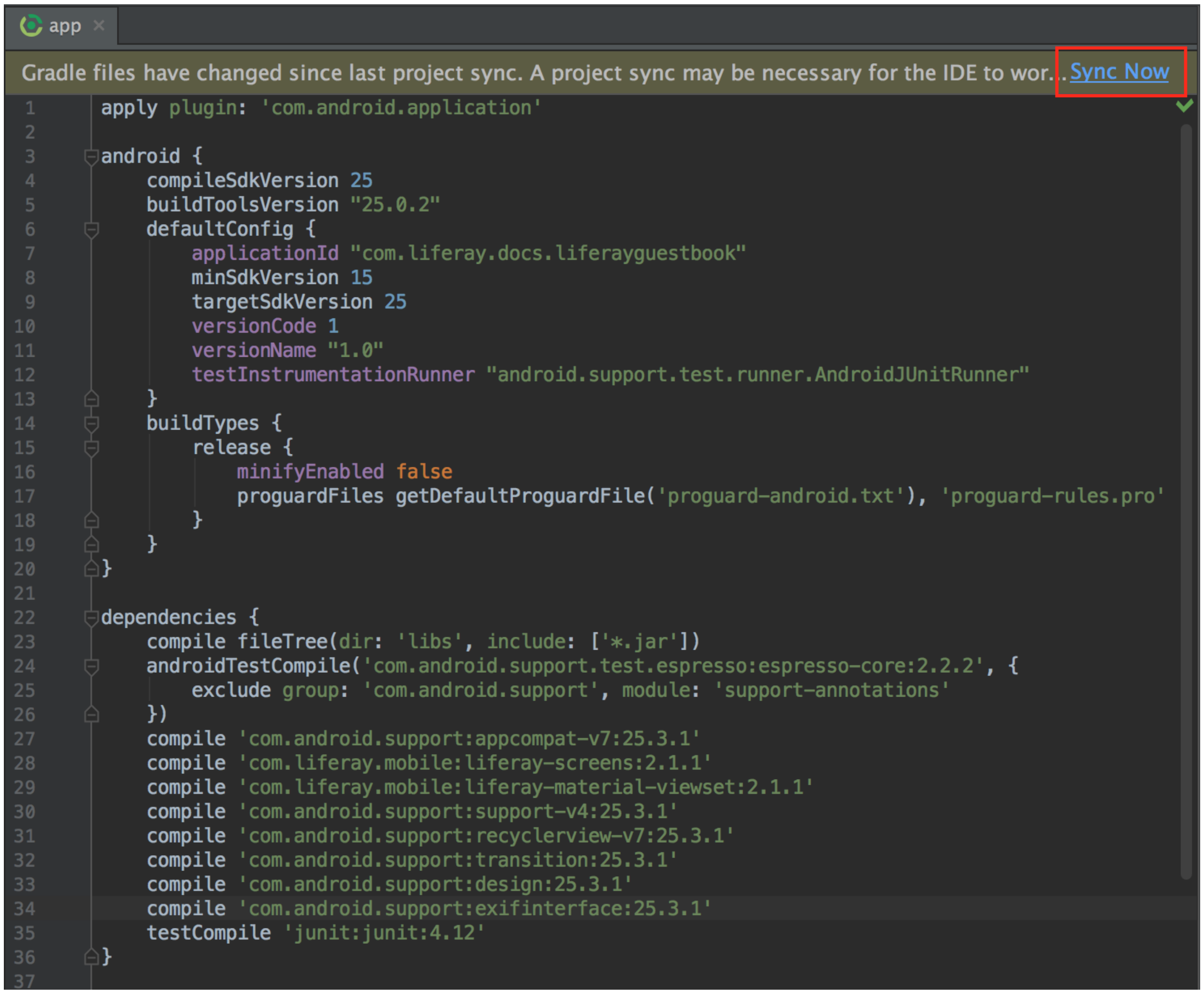
```
import com.liferay.mobile.android.service.Session;
import com.liferay.mobile.android.v7.entry.EntryService;
import com.liferay.mobile.android.v7.guestbook.GuestbookService;
import com.liferay.mobile.screens.auth.login.LoginScreenlet;
```

If Android Studio recognizes these imports, then you're good to go! Remove them once you've verified that they're valid. Next, there's one final small but important task to complete: point your app at the correct Liferay DXP instance.

Configuring Communication with Liferay DXP

For Screens to work properly with your app, you must point it to your Liferay DXP instance. You do this by adding a server_context.xml file in your project's res/values directory. Create this file and add the following code to it:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <!-- Change these values for your portal instance -->

    <string name="liferay_server">http://10.0.2.2:8080</string>

    <integer name="liferay_company_id">20116</integer>
    <integer name="liferay_group_id">20147</integer>

    <integer name="liferay_portal_version">70</integer>

</resources>
```


app
Gradle files have changed since last project sync. A project sync may be necessary for the IDE to wor...
Sync Now

```
apply plugin: 'com.android.application'

android {
    compileSdkVersion 25
    buildToolsVersion "25.0.2"
    defaultConfig {
        applicationId "com.liferay.docs.liferayguestbook"
        minSdkVersion 15
        targetSdkVersion 25
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {
        exclude group: 'com.android.support', module: 'support-annotations'
    })
    compile 'com.android.support:appcompat-v7:25.3.1'
    compile 'com.liferay.mobile:liferay-screens:2.1.1'
    compile 'com.liferay.mobile:liferay-material-viewset:2.1.1'
    compile 'com.android.support:support-v4:25.3.1'
    compile 'com.android.support:recyclerview-v7:25.3.1'
    compile 'com.android.support:transition:25.3.1'
    compile 'com.android.support:design:25.3.1'
    compile 'com.android.support:exifinterface:25.3.1'
    testCompile 'junit:junit:4.12'
}

```

Figure 42.10: After editing the app module's `build.gradle` file, click *Sync Now* to incorporate the changes in your app.

As the comment indicates, change the values to match those of your Liferay DXP instance. The server address `http://10.0.2.2:8080` is suitable for testing with Android Studio's emulator, because it corresponds to `localhost:8080` through the emulator. The Liferay DXP instance you set up earlier should be running on `localhost:8080`.

The `liferay_company_id` value is your Liferay DXP instance's ID. You can find it in your Liferay DXP instance at *Control Panel* → *Configuration* → *Virtual Instances*. The instance's ID is in the *Instance ID* column. Copy and paste this value into the `liferay_company_id` value in `server_context.xml`.

The `liferay_group_id` value is the ID of the site your app needs to communicate with. Since the app needs to communicate with the Guestbook portlet, navigate to the site you put the Guestbook portlet on. In the *Site Administration* menu, select *Configuration* → *Site Settings*. The site ID is listed at the top of the *General* tab. Copy and paste this value from your portal to the `liferay_group_id` value in `server_context.xml`.

Awesome! Next, you'll learn the app's basic design.

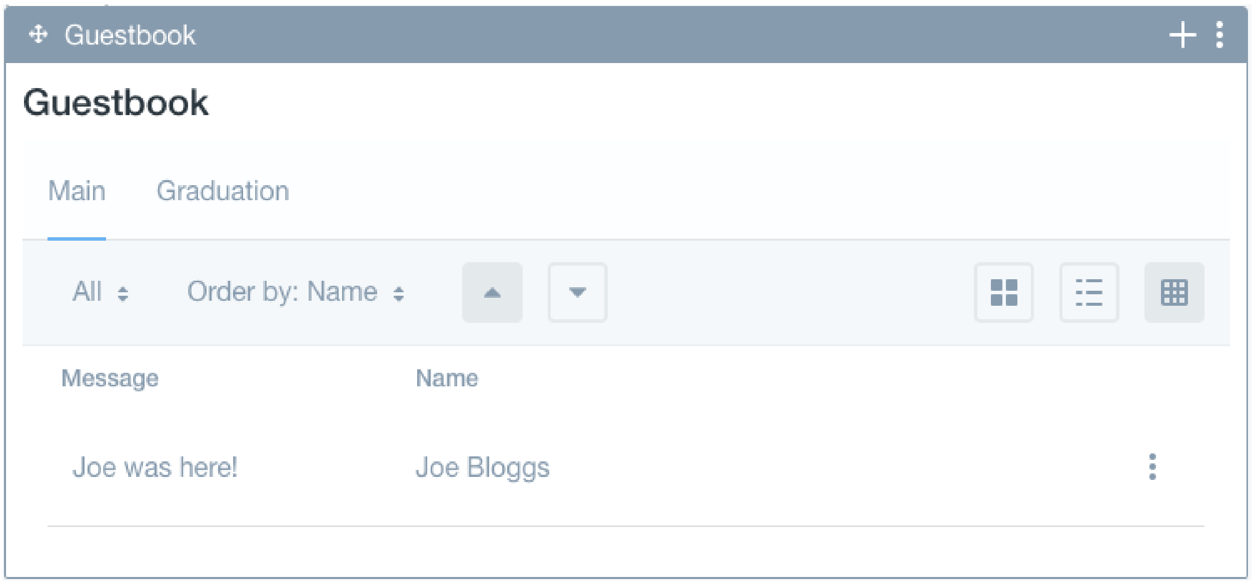
42.5 Designing Your App

As a developer, you know that developing any kind of app without an overall design goal and plan to implement it is a recipe for disaster. To avoid this, you need to decide some things upfront. The Liferay Guestbook app needs a straightforward way to do three things:

1. Authenticate users
2. Display guestbooks
3. Display entries

To authenticate users, all you need to do is insert and configure Login Screenlet in your app. Login Screenlet comes complete with its own UI. The design for authentication, therefore, like with Liferay DXP itself, is done for you.

You must, however, create the UI for displaying guestbooks and entries. What sort of UI would be best for this? Although the *best* UI for any purpose is a matter of opinion, displaying guestbooks and entries in a list is a good choice. Lists are common, compact design elements familiar to mobile users. Since most mobile devices don't have room to display a list of guestbooks and a list of entries at the same time, you also need a user-friendly way to display and manage these lists. It makes sense to show the first guestbook's entries automatically after the user authenticates. This is similar to the Guestbook portlet's design: it shows a list of the first guestbook's entries by default. When the user selects a different guestbook, you can then use the same UI to show the selected guestbook's entries instead.



Message	Name
Joe was here!	Joe Bloggs

Figure 42.11: By default, the first guestbook in the portlet is selected.

You must also decide how the users can select different guestbooks. Showing the list of guestbooks in a navigation drawer that slides out from the left side of the screen is a good choice. A navigation drawer is easily hidden and is a common Android UI element.

To display these lists of guestbooks and entries, you'll create your own Screenlets: Guestbook List Screenlet and Entry List Screenlet. Guestbook List Screenlet needs to retrieve guestbooks from the portlet and display them in a simple list. Once written, using this Screenlet is a simple matter of inserting it in the navigation drawer. Entry List Screenlet needs to retrieve and display a guestbook's entries in a similar list. You'll display the entries by inserting Entry List Screenlet in the UI element where you want it.

Also note that these Screenlets are *list Screenlets*. You develop list Screenlets by using the list Screenlet framework, which sits on top of the core Screenlet framework. The list Screenlet framework makes it easy for developers to write Screenlets that display lists of entities from a Liferay DXP instance.

Awesome! Now you have a basic UI design and know the Screenlets you'll create to implement it. But where in the app can you use these Screenlets? The app only contains one empty activity, `MainActivity`, which you'll use for authentication with Login Screenlet. To use your custom list Screenlets, you'll need to create an additional activity and a fragment: `GuestbooksActivity` and `EntriesFragment`. You'll create the activity in a moment.

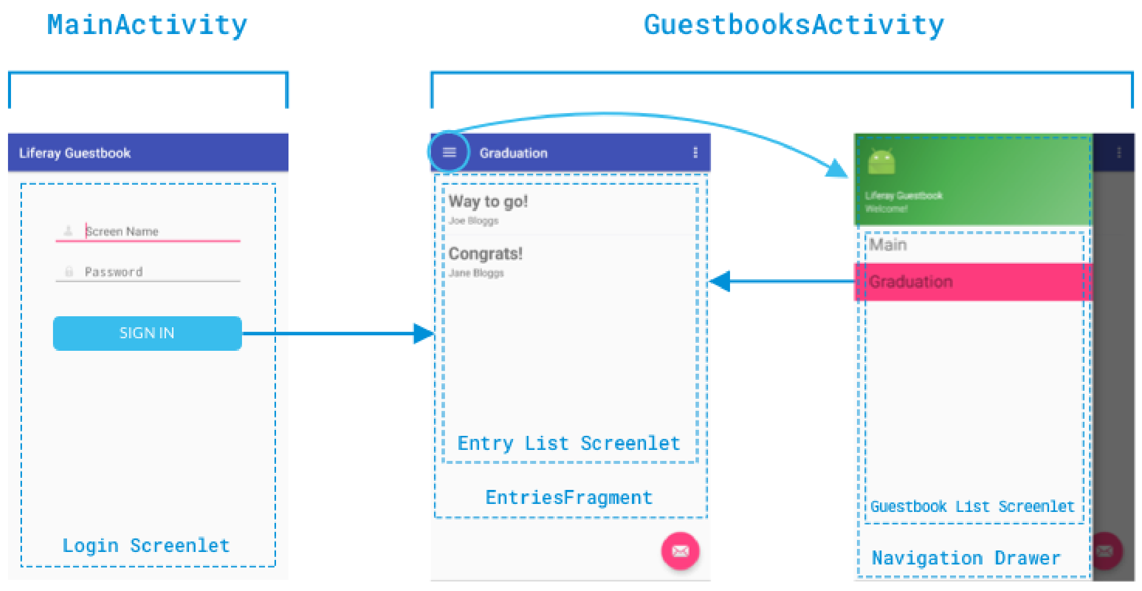


Figure 42.12: The Liferay Guestbook app's design uses two activities and a fragment. In this diagram, each activity and fragment is labeled, along with the Screenlets and the navigation drawer.

In addition to showing the app's components, this diagram shows how the user navigates through the app. After sign in, the user transitions to `GuestbooksActivity`. This activity uses Entry List Screenlet in `EntriesFragment` to display the selected guestbook's entries (the first guestbook is selected by default). Pressing the hamburger button at the top-left of this screen opens the navigation drawer, where Guestbook List Screenlet displays the list of guestbooks. Selecting a guestbook closes the drawer to reveal that guestbook's entries. Note that you only need one activity, `GuestbooksActivity`, to display guestbooks and entries. The navigation drawer and `EntriesFragment` are part of this activity.

Now you're ready to create `GuestbooksActivity`. Fortunately, Android Studio has a template for creating an activity that contains a navigation drawer. Follow these steps to create `GuestbooksActivity` with the navigation drawer template:

1. Right click the package `com.liferay.docs.liferayguestbook` and select *New* → *Activity* → *Navigation Drawer Activity* to launch the New Android Activity wizard.

2. Name the activity `GuestbooksActivity`, accept the defaults for the remaining fields, and click *Finish*.

3. After Android Studio creates the activity, the GuestbooksActivity class and content_guestbooks.xml layout open in the editor. Close them. You don't need to edit these files yet.

Great! Now you understand the Liferay Guestbook app's design. You also have the app structure in place. Next, you'll authenticate users by adding Login Screenlet to MainActivity.

42.6 Using Login Screenlet for Authentication

For the app to retrieve data from the Guestbook portlet, the user must first authenticate to the Liferay DXP instance. You can implement authentication using the Liferay Mobile SDK, but it takes time to write. Using Liferay Screens to authenticate takes about ten minutes. In this article, you'll use Login Screenlet to implement authentication in your app.

Adding Login Screenlet to the App

To use any Screenlet, you must follow two steps:

1. Insert the Screenlet's XML in the layout of the activity or fragment where you want the Screenlet to appear.

2. Implement the Screenlet's listener in the activity or fragment class.

In this app, you'll use Login Screenlet in MainActivity. This means you must insert the Screenlet's XML in MainActivity's layout, activity_main.xml. You'll then implement Login Screenlet's listener, LoginListener, in the MainActivity class.

Insert the Screenlet's XML

Follow these steps to insert Login Screenlet's XML in activity_main.xml:

1. Open activity_main.xml from the res/layout folder and delete the TextView generated by Android Studio when you created the project. Insert Login Screenlet's XML in its place:

   ```
   <com.liferay.mobile.screens.auth.login.LoginScreenlet
       android:id="@+id/login_screenlet"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       app:basicAuthMethod="screen_name"
       app:layoutId="@layout/login_default"
       />
   ```

   Note the two app attributes in the Login Screenlet's XML. The app:basicAuthMethod attribute tells the Screenlet to use basic authentication instead of OAuth. The screen_name value tells the Screenlet to authenticate with the user's screen name. You can alternatively set this to email or userId. This Learning Path uses screen_name only because it's much faster to type a screen name than a full email address in the emulator. Also, this value must match the authentication setting in the Liferay DXP instance. By default, Liferay DXP instances use email address for authentication. For this Learning Path, you need to set your Liferay DXP instance to authenticate by screen name instead. Click here for instructions on changing your Liferay DXP instance's authentication setting.

   The second app attribute in Login Screenlet's XML is app:layoutId. This attribute sets the *View* to display the Screenlet with. Views in Liferay Screens set a Screenlet's look and feel independent of the Screenlet's core functionality. You can think of them as a sort of skin or theme for a Screenlet. The value `@layout/login_default` specifies Login Screenlet's Default View, which is part of the Default *View Set*. A View Set is a collection of Views for several Screenlets. Using a View Set lets you apply a consistent look and feel across multiple Screenlets. To use a View that is part of a View Set, like Login Screenlet's Default View, the theme of the app or activity must inherit the theme that sets the View Set's styles. For the Default View Set, this is `default_theme`.

2. To set the app's theme to inherit from `default_theme`, open `res/values/styles.xml` and set the base app theme's parent to `default_theme`. In this app, the base app theme is `AppTheme`. The theme declaration should now look like this:

```
<style name="AppTheme" parent="default_theme">
    ...
```

Click here for more information on using Views in Liferay Screens. For more information on Login Screenlet's available attributes, click here.

Next, you'll implement `LoginListener` in the `MainActivity` class.

Implement the Screenlet's Listener

To use a Screenlet in an activity or fragment, you must also implement the Screenlet's listener in that activity or fragment's class. You'll do this now to use Login Screenlet in `MainActivity`:

1. Open `MainActivity` and change its declaration to implement `LoginListener`. The class declaration should now look like this:

```
public class MainActivity extends AppCompatActivity implements LoginListener {...
```

   You must also import `com.liferay.mobile.screens.auth.login.LoginListener`.

2. Implementing `LoginListener` requires you to implement the `onLoginSuccess` and `onLoginFailure` methods. Add them to the class as follows:

```
@Override
public void onLoginSuccess(User user) {
    Toast.makeText(this, "Login successful!", Toast.LENGTH_SHORT).show();
}

@Override
public void onLoginFailure(Exception e) {
    Toast.makeText(this, "Couldn't log in " + e.getMessage(), Toast.LENGTH_LONG).show();
}
```

   When you add these methods, you must import `android.widget.Toast` and `com.liferay.mobile.screens.context.User`.

   These are listener methods called when login succeeds or fails, respectively. Using them lets your app respond the Screenlet's actions. For the moment, they each only do one thing: display a success or failure message to the user. You'll change this shortly. Note that each Screenlet has different listener methods; they're listed in the Screenlet reference documentation.

3. Now you must get a reference to the Screenlet and set the `MainActivity` class as its listener. To do so, add the following code to the end of the `onCreate` method:

```
LoginScreenlet loginScreenlet = (LoginScreenlet) findViewById(R.id.login_screenlet);
loginScreenlet.setListener(this);
```

This requires you to import `com.liferay.mobile.screens.auth.login.LoginScreenlet`.

The `findViewById` method uses the Screenlet's ID from the layout to create the reference. The `setListener` method then sets `MainActivity` as Login Screenlet's listener.

Now run the app by clicking the green *play* button in the toolbar, or by selecting *Run 'app'* from the *Run* menu. If you've never run the emulator, you must first create and choose an Android Virtual Device (AVD) to run your app. For more information on this and running the emulator in general, click here. Once the emulator launches, unlock it if necessary. Your app automatically opens to Login Screenlet. Enter your credentials and click *SIGN IN*. The toast message pops up saying that the login succeeded.

The toast message goes away and you remain on the login screen. Nothing else happens. Don't worry, this is supposed to happen; you haven't added any other functionality yet. You'll fix this next.

Navigating from Login Screenlet

When login succeeds, the app should open `GuestbooksActivity`. You'll do this by using an Android intent in `MainActivity`'s `onLoginSuccess` method:

1. Replace the contents of `onLoginSuccess` with this code:

   ```
   Intent intent = new Intent(this, GuestbooksActivity.class);
   startActivity(intent);
   ```

   When login succeeds, this code creates an `Intent` and uses it to start `GuestbooksActivity`. If you haven't already, make sure to import `android.content.Intent` in `MainActivity`.

2. Now you're ready to see the intent in action! Run the app in the emulator and log in when prompted. When login succeeds, `GuestbooksActivity` opens.

Nice work! You successfully implemented Liferay DXP authentication in your Android app. It didn't take you that long, either. So far, however, that's all your app does; it doesn't display any content. Next, you'll rectify this by developing Guestbook List Screenlet.

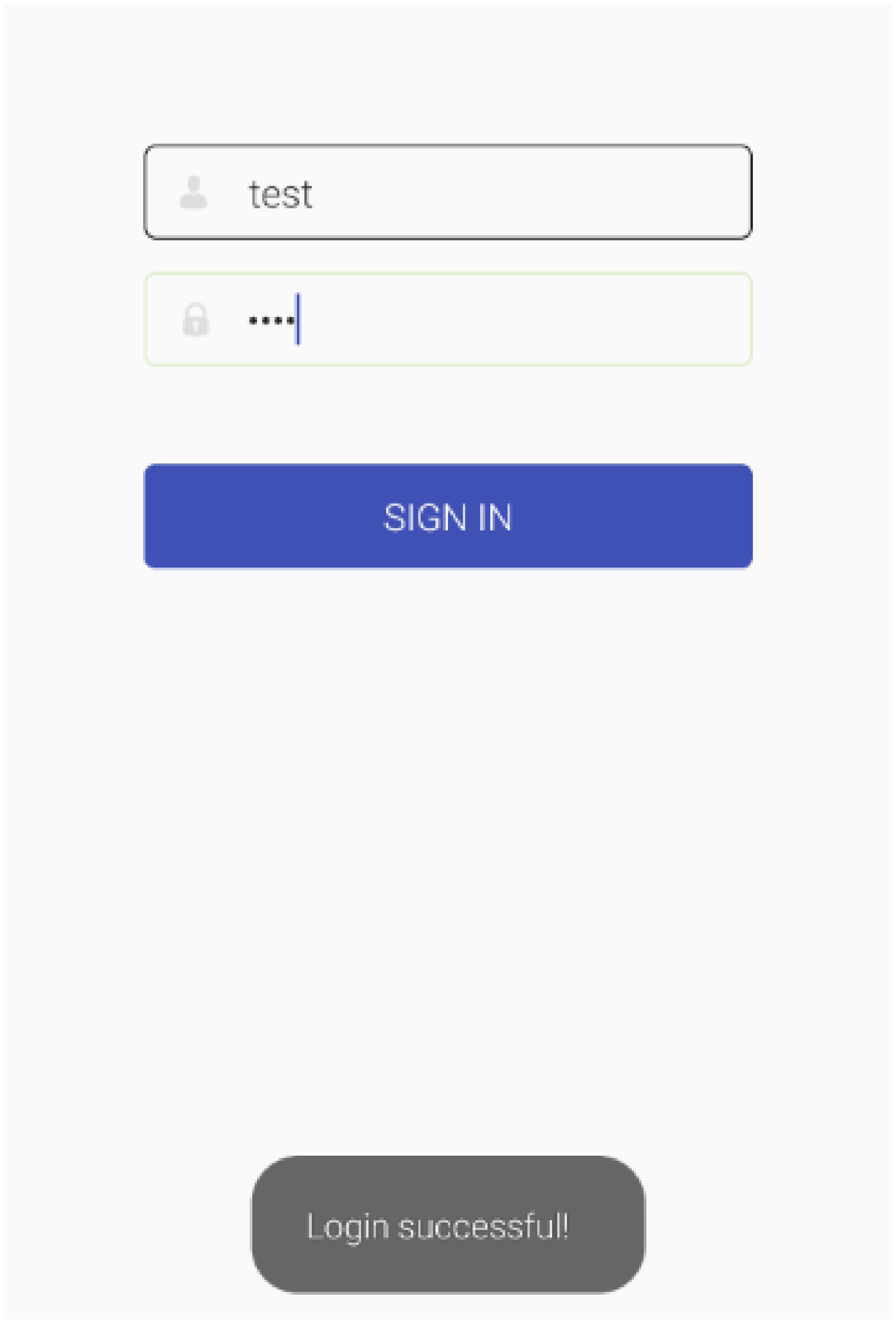


Figure 42.13: Login Screenlet successfully authenticated you with the Liferay DXP instance.

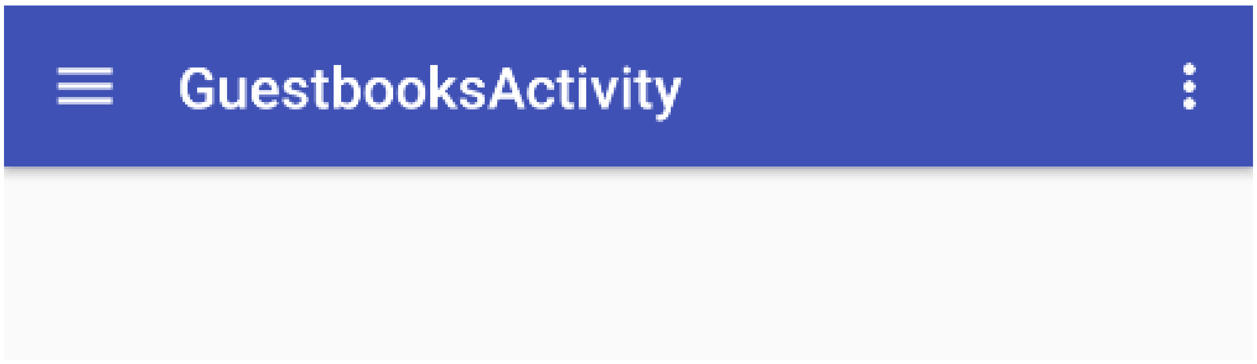


Figure 42.14: Upon login, the app takes you to the new activity.

Chapter 43

Creating Guestbook List Screenlet

In the previous section, you created an Android app that contains the Guestbook Mobile SDK and Liferay Screens. You also used Login Screenlet to implement authentication to Liferay DXP. That's all your app does though. It doesn't display any Guestbook portlet content. In this section of the Learning Path, you'll create Guestbook List Screenlet to retrieve and display the portlet's guestbooks in your app's navigation drawer.

Creating your own Screenlets brings additional benefits. Since you use a consistent, repeatable development model to create them, you can often reuse code when creating other Screenlets. You can also package and reuse Screenlets in other apps. What's more, Screenlet UIs are fully pluggable. This lets you change a Screenlet's appearance quickly without affecting its functionality. In summary, Screenlets are pretty much the greatest thing since sliced bread. Now it's time to make a sandwich.

As background material, the following materials are helpful:

1. Getting started: creating the Screenlet's package, and model class.

2. Creating the Screenlet's UI (its View).

3. Creating the Screenlet's Interactor. Interactors are Screenlet components that make server calls.

4. Creating the Screenlet class. The Screenlet class governs the Screenlet's behavior.

Before beginning, you should read the following tutorials:

- Architecture of Liferay Screens for Android: Explains the components that constitute a Screenlet, and how they relate to one another.

- Creating Android Screenlets: Explains the general steps for creating a Screenlet.

- Creating Android List Screenlets: Explains the general steps for creating a list Screenlet. This section of the Learning Path follows this tutorial.

Note that these tutorials explain Screenlet and list Screenlet concepts that this Learning Path doesn't cover in depth. Although it's possible to complete this Learning Path without reading these tutorials, they explain how Screenlets work in more detail. By reading them you'll be better able to apply the Learning Path material to your own Screenlets.

If you get confused or stuck while creating Guestbook List Screenlet, refer to the finished app that contains the Screenlet code here in GitHub.

43.1 Getting Started with Guestbook List Screenlet

Before creating a Screenlet, you should know how you'll use it. If you plan to use it in only one app, then you can create it in that app's project. If you need to use it in several apps, however, then it's best to create it in a separate project for redistribution. For information on creating Screenlets for redistribution, see the tutorial Packaging Your Android Screenlets. Since you'll use Guestbook List Screenlet in only this app, you can create it in a new package inside the app's project. Create this package now:

1. In Android Studio, right click the *java* folder and select *New → Package*.
2. Select *.../app/src/main/java* as the destination directory, and click *OK*.
3. Enter `com.liferay.docs.guestbooklistscreenlet` as the package's name and click *OK*. Android Studio lists the new package alongside the package that contains the app's activity and fragment (`liferayguestbook`). If it doesn't appear at first, you may need to collapse and reopen the *java* folder.

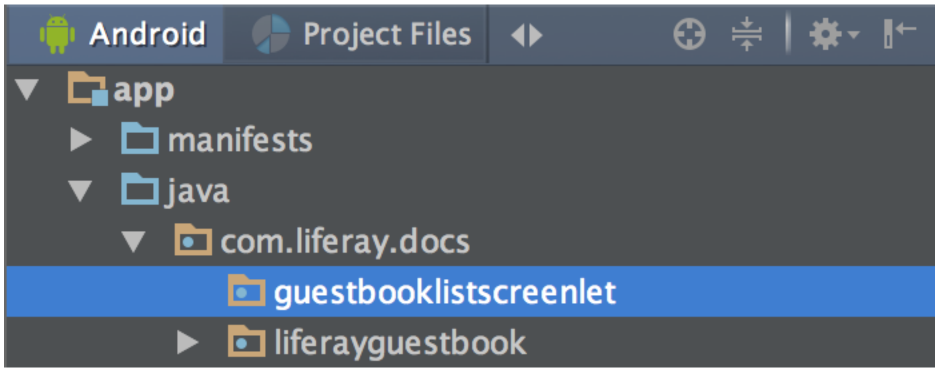


Figure 43.1: Guestbook List Screenlet's new package is highlighted.

Before getting started, you should understand how pagination works in list Screenlets.

Pagination

To ensure that users can scroll smoothly through large lists of items, list Screenlets support fluent pagination. Support for this is built into the list Screenlet framework. You'll see this as you construct your list Screenlet. For example, several methods have parameters for the start row and end row of a page in the list.

Now you're ready to begin!

Creating the Model Class for Guestbooks

Entities come back from Liferay Portal in JSON. To work with these results efficiently in your app, you must convert them to model objects that represent the entity in the portal. Although Screens's `BaseListInteractor` transforms the JSON entities into `Map` objects for you, you still must convert these into proper entity objects for use in your app. You'll do this via a model class.

The model class you'll create for Guestbook List Screenlet, `GuestbookModel`, creates `GuestbookModel` objects that represent guestbooks retrieved from the Guestbook portlet. You'll create this model class in a separate package outside of the `guestbooklistscreenlet` package. In this case, it makes sense to organize your code this way because other Screenlets may also use the model class. For example, if a Screenlet that edits guestbooks existed, it would also need `GuestbookModel` objects. Putting the model class in a separate package makes it clear that this class doesn't belong exclusively to a single Screenlet.

Follow these steps to create `GuestbookModel`:

1. Create a new package called `model` inside the `com.liferay.docs` package.

2. Inside this new `model` package, create a new class called `GuestbookModel`.

3. Replace `GuestbookModel`'s contents with this code:

```
package com.liferay.docs.model;

import android.os.Parcel;
import android.os.Parcelable;

import java.util.Date;
import java.util.Map;

public class GuestbookModel implements Parcelable {

    private Map values;
    private long guestbookId;
    private long groupId;
    private long companyId;
    private long userId;
    private String userName;
    private long createDate;
    private long modifiedDate;
    private String name;

    public static final Creator<GuestbookModel> CREATOR = new Creator<GuestbookModel>() {
        @Override
        public GuestbookModel createFromParcel(Parcel in) {
            return new GuestbookModel(in);
        }

        @Override
        public GuestbookModel[] newArray(int size) {
            return new GuestbookModel[size];
        }
    };

    public GuestbookModel() {
        super();
    }

    protected GuestbookModel(Parcel in) {
        guestbookId = in.readLong();
        groupId = in.readLong();
        companyId = in.readLong();
        userId = in.readLong();
        userName = in.readString();
        createDate = in.readLong();
        modifiedDate = in.readLong();
        name = in.readString();
    }

    public GuestbookModel(Map<String, Object> stringObjectMap) {
        values = stringObjectMap;
        guestbookId = Long.parseLong((String) stringObjectMap.get("guestbookId"));
        groupId = Long.parseLong((String) stringObjectMap.get("groupId"));
        companyId = Long.parseLong((String) stringObjectMap.get("companyId"));
        userId = Long.parseLong((String) stringObjectMap.get("userId"));
        userName = (String) stringObjectMap.get("userName");
        createDate = (long) stringObjectMap.get("createDate");
```

```
            modifiedDate = (long) stringObjectMap.get("modifiedDate");
            name = (String) stringObjectMap.get("name");
        }

        @Override
        public void writeToParcel(Parcel dest, int flags) {
            dest.writeLong(guestbookId);
            dest.writeLong(groupId);
            dest.writeLong(companyId);
            dest.writeLong(userId);
            dest.writeString(userName);
            dest.writeLong(createDate);
            dest.writeLong(modifiedDate);
            dest.writeString(name);
        }

        @Override
        public int describeContents() {
            return 0;
        }

        public long getGuestbookId() {
            return guestbookId;
        }

        public long getGroupId() {
            return groupId;
        }

        public long getCompanyId() {
            return companyId;
        }

        public long getUserId() {
            return userId;
        }

        public String getUserName() {
            return userName;
        }

        public Date getCreateDate() {
            return new Date(createDate);
        }

        public Date getModifiedDate() {
            return new Date(modifiedDate);
        }

        public String getName() {
            return name;
        }

        public Map getValues() {
            return values;
        }

        public void setValues(Map values) {
            this.values = values;
        }
    }
```

This class creates `GuestbookModel` objects that represent the Guestbook portlet's `Guestbook` objects. The constructor with the `Map<String, Object>` argument does the heavy lifting. Following a successful Mobile SDK call, the list Screenlet framework's `BaseListInteractor` class returns this `Map`, which contains the data of a guestbook retrieved from the portlet. To get the guestbook's data from the `Map`, the constructor uses the `get` method with each parameter of the portlet's `Guestbook` entity. To see how the portlet defines these parameters, see the Liferay MVC Learning Path article on Service Builder. For now, the only parameters you really need in `GuestbookModel` are `guestbookId` and `name`. Because you might need the rest later, however, it's best to add all of them now.

Besides the getters and setter, the remaining code in `GuestbookModel` implements Android's `Parcelable` interface. For more information on this, see Android's documentation on `Parcelable`.

Great! Now you have a model class for guestbooks. Next, you'll create the Screenlet's UI.

43.2 Creating Guestbook List Screenlet's UI

Recall that in Liferay Screens for Android, Screenlet UIs are called Views, and every Screenlet must have at least one View. In this article, you'll use the following steps to create a View for Guestbook List Screenlet:

1. Create the row layout. This layout defines the UI for each row instance in the list.

2. Create the adapter class. This is an Android adapter that fills a row layout instance with the data for one list item. This repeats for each list item until the list is full.

3. Create the View class. This class renders the UI, handles user interactions, and communicates with the Screenlet class.

4. Create the View's layout. This layout defines the Screenlet's UI as a whole. For a list Screenlet, this is a scrollable list.

Note that these are the same steps for creating a View as those in the list Screenlet tutorial.

You'll create Guestbook List Screenlet's View in its own package inside the `guestbooklistscreenlet` package. Create a new package named `view` inside the `guestbooklistscreenlet` package. Now you're ready to create the row layout.

Creating the Row Layout

First, you must create the layout that defines the UI for each row instance in the list. Since each row in Guestbook List Screenlet displays only a single guestbook's name, the row layout only needs a single `TextView`. Create the layout file `res/layout/guestbook_row.xml` and paste in this content:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:id="@+id/guestbook_name"
        android:textSize="25sp"
        android:padding="10dp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

Note that the `textSize` and `padding` values result in clean, readable text for this example. When you develop your own list Screenlets, you can style your row layouts however you like.

Creating the Adapter Class

Android adapters fill a layout with content. In Guestbook List Screenlet, the layout is `guestbook_row.xml` (the row layout) and the content is a guestbook's name. To make list scrolling smooth, the adapter class should use an Android view holder. To make this easier, you can extend the list Screenlet framework's `BaseListAdapter` class with your model class and view holder as type arguments. By extending `BaseListAdapter`, your adapter needs only two methods:

- `createViewHolder`: instantiates the view holder
- `fillHolder`: fills in the view holder for each row

Your view holder should also contain variables for any data each row needs to display. The view holder must assign these variables to the corresponding row layout elements, and set the appropriate data to them.

Inside the Screenlet's `view` package, create the following `GuestbookAdapter` class:

```
package com.liferay.docs.guestbooklistscreenlet.view;

import android.support.annotation.NonNull;
import android.view.View;
import android.widget.TextView;

import com.liferay.docs.liferayguestbook.R;
import com.liferay.docs.model.GuestbookModel;
import com.liferay.mobile.screens.base.list.BaseListAdapter;
import com.liferay.mobile.screens.base.list.BaseListAdapterListener;

public class GuestbookAdapter extends
    BaseListAdapter<GuestbookModel, GuestbookAdapter.GuestbookViewHolder> {

    public GuestbookAdapter(int layoutId, int progressLayoutId, BaseListAdapterListener listener) {
        super(layoutId, progressLayoutId, listener);
    }

    @NonNull
    @Override
    public GuestbookViewHolder createViewHolder(View view, BaseListAdapterListener listener) {
        return new GuestbookAdapter.GuestbookViewHolder(view, listener);
    }

    @Override
    protected void fillHolder(GuestbookModel entry, GuestbookViewHolder holder) {
        holder.bind(entry);
    }

    public class GuestbookViewHolder extends BaseListAdapter.ViewHolder {

        private final TextView name;

        public GuestbookViewHolder(View view, BaseListAdapterListener listener) {
            super(view, listener);

            name = (TextView) view.findViewById(R.id.guestbook_name);
        }

        public void bind(GuestbookModel entry) {
            name.setText(entry.getName());
        }

    }
}
```

This adapter class extends `BaseListAdapter` with `GuestbookModel` and `GuestbookAdapter.GuestbookViewHolder` as type arguments. The view holder is an inner class that extends `BaseListAdapter`'s view holder. Since

Guestbook List Screenlet only needs to display a guestbook's name in each row, the view holder only needs one variable: `name`. The view holder's constructor assigns the `TextView` from `guestbook_row.xml` to this variable. The `bind` method then sets the guestbook's name as the `TextView`'s text. The other methods in `GuestbookAdapter` leverage the view holder. The `createViewHolder` method instantiates `GuestbookViewHolder`. The `fillHolder` method calls the view holder's `bind` method to set the guestbook's name as the `name` variable's text.

Next, you'll create the View class.

Creating the View Class

Recall that the View class controls a Screenlet's UI. It renders the UI, handles user interactions, and communicates with the Screenlet class. The list Screenlet framework provides most of this functionality for you via the `BaseListScreenletView` class. Your View class must extend this class to provide your row layout ID and an instance of your adapter. You'll do this by overriding `BaseListScreenletView`'s `getItemLayoutId` and `createListAdapter` methods. Also, when you extend `BaseListScreenletView` you must do so with your model class, view holder, and adapter as type arguments. This is required for your View class to represent your model objects in a view holder, inside an adapter.

Create the `GuestbookListView` class inside the `view` package, and replace its contents with this code:

```
package com.liferay.docs.guestbooklistscreenlet.view;

import android.content.Context;
import android.util.AttributeSet;

import com.liferay.docs.liferayguestbook.R;
import com.liferay.docs.model.GuestbookModel;
import com.liferay.mobile.screens.base.list.BaseListScreenletView;

public class GuestbookListView extends BaseListScreenletView<GuestbookModel,
    GuestbookAdapter.GuestbookViewHolder, GuestbookAdapter> {

    public GuestbookListView(Context context) {
        super(context);
    }

    public GuestbookListView(Context context, AttributeSet attributes) {
        super(context, attributes);
    }

    public GuestbookListView(Context context, AttributeSet attributes, int defaultStyle) {
        super(context, attributes, defaultStyle);
    }

    @Override
    protected GuestbookAdapter createListAdapter(int itemLayoutId, int itemProgressLayoutId) {
        return new GuestbookAdapter(itemLayoutId, itemProgressLayoutId, this);
    }

    @Override
    protected int getItemLayoutId() {
        return R.layout.guestbook_row;
    }
}
```

This View class represents `GuestbookModel` instances in a `GuestbookViewHolder` inside a `GuestbookAdapter`. This class therefore extends `BaseListScreenletView` parameterized with `GuestbookModel`, `GuestbookAdapter.GuestbookViewHold` and `GuestbookAdapter`. Besides overriding `createListAdapter` to return a `GuestbookAdapter` instance, the only other functionality that this View class needs to support is to get the layout for each row in the list. The overridden `getItemLayoutId` method does this by returning the row layout `guestbook_row`.

Now you're ready to create your View's main layout.

Creating the View's Layout

Although you already created a layout for your list rows, you must still create a layout to define the list as a whole. This layout must contain:

- The View class's fully qualified name as the layout's first element.
- An Android `RecyclerView` to let your app efficiently scroll through a potentially large list of items.
- An Android `ProgressBar` to indicate progress when loading the list.

Apart from the View class and styling, this layout's code is the same for all list Screenlets.

Create the layout file `res/layout/list_guestbooks.xml` and replace its contents with this code:

```
<com.liferay.docs.guestbooklistscreenlet.view.GuestbookListView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/liferay_list_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ProgressBar
        android:id="@+id/liferay_progress"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:visibility="gone"/>

    <android.support.v7.widget.RecyclerView
        android:id="@+id/liferay_recycler_list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:visibility="gone"/>
</com.liferay.docs.guestbooklistscreenlet.view.GuestbookListView>
```

Note that the `android:id` values in this layout XML are hardcoded into the Screens framework and changing them will cause your app to crash.

Great! You're done with Guestbook List Screenlet's View. Next, you'll create the Screenlet's Interactor.

43.3 Creating Guestbook List Screenlet's Interactor

Interactors are Screenlet components that make server calls and process the results. Interactors themselves are made up of several components:

1. **The event class:** creates event objects that contain the server call's results. Liferay Screens uses these event objects via the EventBus library to communicate the results between the Screenlet's components.

2. **The listener interface:** defines the methods the app developer needs to respond to the Screenlet's behavior. For example, Login Screenlet's listener defines the `onLoginSuccess` and `onLoginFailure` methods. Screens calls these methods when login succeeds or fails, respectively. By implementing these methods in the activity or fragment class that contains the Screenlet, the app developer can respond to login success and failure.

3. **The Interactor class:** makes the server calls, processes the results in the event objects, and notifies the listener of those results.

Since the list Screenlet framework already contains two listener interfaces, you only need to create the event and Interactor classes. You'll create the event class first.

Creating the Event Class

1. First, create a new package called `interactor` in the `com.liferay.docs.guestbooklistscreenlet` package. You'll create your Interactor's components in this new package.

2. A list Screenlet's event class must extend the `ListEvent` class with the Screenlet's model class as a type argument. This lets the event class contain the server call's results as model objects. Guestbook List Screenlet's event class, `GuestbookEvent`, must therefore extend `ListEvent` with `GuestbookModel` as a type argument. Create this class now in the `interactor` package. The class declaration should look like this:

   ```
   public class GuestbookEvent extends ListEvent<GuestbookModel> {...
   ```

   This requires you to import `com.liferay.docs.model.GuestbookModel` and `com.liferay.mobile.screens.base.list.intera`

3. The event class should also contain a private instance variable for the model class, a constructor that sets this variable, and a no-argument constructor that calls the superclass constructor. Add this code now:

   ```
   private GuestbookModel guestbook;

   public GuestbookEvent() {
       super();
   }

   public GuestbookEvent(GuestbookModel guestbook) {
       this.guestbook = guestbook;
   }
   ```

4. You must also implement `ListEvent`'s abstract methods in your event class. Note that these methods support offline mode. Even though Guestbook List Screenlet doesn't support offline mode, you must still implement these methods. Add these methods to `GuestbookEvent` now:

   - `getListKey`: returns the ID for the cache. This ID is typically the data each list row displays. For example, the `getListKey` method in `GuestbookEvent` returns the guestbook's name:

     ```
     @Override
     public String getListKey() {
         return guestbook.getName();
     }
     ```

   - `getModel`: unwraps the model entity to the cache by returning the model class instance. For example, the `getModel` method in `GuestbookEvent` method returns the guestbook:

     ```
     @Override
     public GuestbookModel getModel() {
         return guestbook;
     }
     ```

Note that this code is almost identical to the example event class in the list Screenlet tutorial. The only difference is that `GuestbookEvent` handles `GuestbookModel` objects.

Nice work! Your event class is done. You're almost ready to write the Screenlet's server call. First, however, you should understand the basics of how server calls work in Interactors.

Understanding Screenlet Server Calls

Recall that Interactor classes use the Liferay Mobile SDK to make server calls and process the results. An Interactor class does this with the following sequence:

1. Get the Mobile SDK session and use it to create the Mobile SDK service you want to call.
2. Invoke the Mobile SDK service method that makes the server call.
3. Create an event object from the JSON that the server call returns. If your Screenlet has a model class, create a model object from this JSON, then use the model object to create the event object.

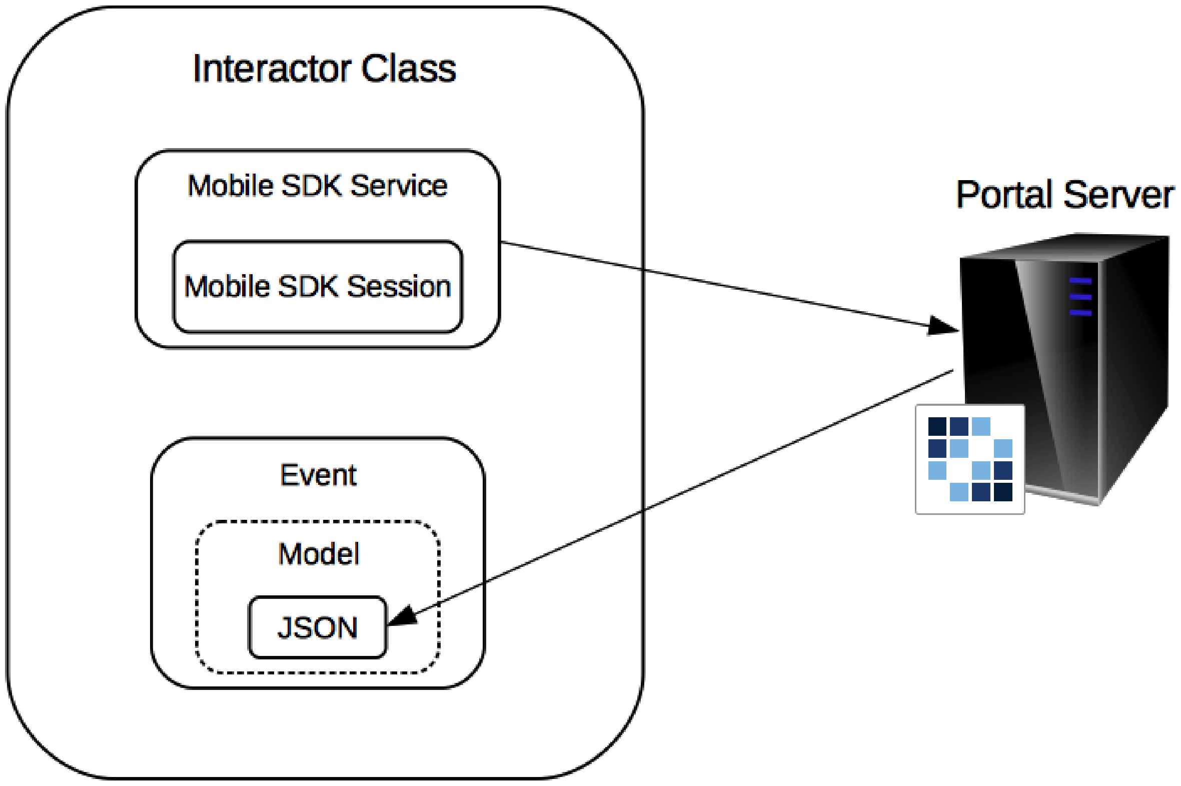


Figure 43.2: This diagram shows a typical server call made by a Screenlet's Interactor. The dashed line around the model class indicates that it's optional. Although list Screenlets require model classes, non-list Screenlets don't.

To call the Guestbook portlet's remote services, you'll use the Guestbook Mobile SDK you built and installed earlier. This Mobile SDK contains the services required to call the Guestbook portlet's remote services. Next, you'll create Guestbook List Screenlet's Interactor class.

Creating the Interactor Class

A Screenlet's Interactor class is the central component of the Interactor. It makes the server calls, processes the results in the event objects, and notifies the listener of those results. The list Screenlet framework's `BaseListInteractor` class provides most of the functionality that Interactor classes in list Screenlets require.

You must, however, extend `BaseListInteractor` to make your service calls and handle their results via your model and event classes.

Follow these steps to create Guestbook List Screenlet's Interactor class, `GuestbookListInteractor`:

1. Create the `GuestbookListInteractor` class in the package `com.liferay.docs.guestbooklistscreenlet.interactor`. A list Screenlet's Interactor class must extend `BaseListInteractor` with `BaseListInteractorListener<YourModelClass>` and your event class as type arguments. You must therefore change `GuestbookListInteractor` to extend `BaseListInteractor` with `BaseListInteractorListener<GuestbookModel>` and `GuestbookEvent` as type arguments:

   ```
   public class GuestbookListInteractor extends
       BaseListInteractor<BaseListInteractorListener<GuestbookModel>, GuestbookEvent> {...
   ```

   This requires that you add the following imports:

   ```
   import com.liferay.docs.model.GuestbookModel;
   import com.liferay.mobile.screens.base.list.interactor.BaseListInteractor;
   import com.liferay.mobile.screens.base.list.interactor.BaseListInteractorListener;
   ```

2. Override the `getPageRowsRequest` method to retrieve a page of entities. In this method, you can use the `getSession()` method to retrieve the session created by authentication with Login Screenlet. Then make the server call by creating a service instance from the session and calling the service method that retrieves the entities. Guestbook List Screenlet must retrieve a page of guestbooks, so you must create a `GuestbookService` instance from the session. Then call the service's `getGuestbooks` method with the `groupId`, start row, and end row. The `groupId` specifies the site to retrieve guestbooks from, while the start row and end row define the list rows that mark the start and end of the page of guestbooks, respectively. Add this `getPageRowsRequest` method to `GuestbookListInteractor`:

   ```
   @Override
   protected JSONArray getPageRowsRequest(Query query, Object... args) throws Exception {

       return new GuestbookService(getSession()).getGuestbooks(groupId, query.getStartRow(),
           query.getEndRow());
   }
   ```

   Note that the `groupId` variable isn't set anywhere. Interactors that extend `BaseListInteractor`, like `GuestbookListInteractor`, inherit this variable via the Screens framework. You'll set it when you create the Screenlet class.

   This `getPageRowsRequest` method requires that you add the following imports:

   ```
   import com.liferay.mobile.android.v7.guestbook.GuestbookService;
   import com.liferay.mobile.screens.base.list.interactor.Query;
   import org.json.JSONArray;
   ```

3. Override the `getPageRowCountRequest` method to retrieve the total number of entities. This enables pagination. In `GuestbookListInteractor`, you retrieve the total number of guestbooks from a site by creating a `GuestbookService` instance from the session and then calling the service's `getGuestbooksCount` method with the `groupId`. Add this `getPageRowCountRequest` method to `GuestbookListInteractor`:

```
@Override
protected Integer getPageRowCountRequest(Object... args) throws Exception {

    return new GuestbookService(getSession()).getGuestbooksCount(groupId);
}
```

4. Override the `createEntity` method to create and return a new event object containing the server call's results. The `BaseListInteractor` class converts the JSON that results from a successful server call into a `Map<String, Object>`. The `createEntity` method's only argument is this `Map`, which you use to create a `GuestbookModel` object. Then use the model object to create and return a new `GuestbookEvent` object. Add this `createEntity` method to `GuestbookListInteractor`:

   ```
   @Override
   protected GuestbookEvent createEntity(Map<String, Object> stringObjectMap) {
       GuestbookModel guestbook = new GuestbookModel(stringObjectMap);
       return new GuestbookEvent(guestbook);
   }
   ```

   This requires you to import `java.util.Map`.

5. Override the `getIdFromArgs` method to return the value of the first object argument as a string. Add this method to `GuestbookListInteractor`:

   ```
   @Override
   protected String getIdFromArgs(Object... args) {
       return String.valueOf(args[0]);
   }
   ```

   This is a boilerplate method that returns a cache key for offline mode. Even though you won't add offline mode support to Guestbook List Screenlet, this method makes it easier if you decide to do so later.

Nice work! Your Interactor class is finished. Note that this class is very similar to the Interactor class in the list Screenlet creation tutorial.

Your Interactor is finished too. Next, you'll create the Screenlet class.

43.4 Creating Guestbook List Screenlet's Screenlet Class

When using a Screenlet, app developers primarily interact with its Screenlet class. The Screenlet class contains attributes for configuring the Screenlet's behavior, a reference to the Screenlet's View, methods for invoking Interactor operations, and more.

You'll use these steps to create the Screenlet class:

1. Define the Screenlet's attributes. These are the XML attributes the app developer can set when inserting the Screenlet's XML. These attributes control aspects of the Screenlet's behavior.

2. Create the Screenlet class. This class implements the Screenlet's functionality defined in the View and Interactor. It also reads the attribute values and configures the Screenlet accordingly.

First, you'll define Guestbook List Screenlet's attributes.

Defining Screenlet Attributes

Before creating the Screenlet class, you should define its attributes. These are the attributes the app developer can set when inserting the Screenlet's XML in an activity or fragment layout. For example, to use Login Screenlet, the app developer could insert the following Login Screenlet XML in an activity or fragment layout:

```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:basicAuthMethod="email"
    app:layoutId="@layout/login_default"
    />
```

The app developer can set the app attributes `basicAuthMethod` and `layoutId` to set Login Screenlet's authentication method and View, respectively. The Screenlet class reads these settings to enable the appropriate functionality.

When creating a Screenlet, you can define the attributes you want to make available to app developers. You do this in an XML file inside your Android project's `res/values` directory. Guestbook List Screenlet only needs one attribute. You'll define it now. Create the file `guestbook_attrs.xml` in your app's `res/values` directory. Replace the file's contents with the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="GuestbookListScreenlet">
        <attr name="groupId"/>
    </declare-styleable>
</resources>
```

This defines the `groupId` attribute, which the app developer can use to set the portal site to communicate with if they don't want to use the default `groupId` setting in `server_context.xml`. You'll account for this attribute's functionality in the Screenlet class.

Now that you've defined this attribute and know what it needs to do, you're ready to write the Screenlet class.

Extending BaseListScreenlet

Because the `BaseListScreenlet` class provides the basic functionality for all Screenlet classes in list Screenlets, including methods for pagination and other default behavior, your Screenlet class must extend `BaseListScreenlet` with your model class and Interactor as type arguments.

Use the following steps to create the Screenlet class for Guestbook List Screenlet, `GuestbookListScreenlet`:

1. Create the `GuestbookListScreenlet` class in the package `com.liferay.docs.guestbooklistscreenlet`. This class must extend `BaseListScreenlet` with the model and Interactor as type arguments:

   ```
   public class GuestbookListScreenlet extends
       BaseListScreenlet<GuestbookModel, GuestbookListInteractor> {...
   ```

   This requires you to add the following imports:

   ```
   import com.liferay.docs.guestbooklistscreenlet.interactor.GuestbookListInteractor;
   import com.liferay.docs.model.GuestbookModel;
   import com.liferay.mobile.screens.base.list.BaseListScreenlet;
   ```

2. For constructors, leverage the superclass constructors:

   ```
   public GuestbookListScreenlet(Context context) {
       super(context);
   }

   public GuestbookListScreenlet(Context context, AttributeSet attrs) {
       super(context, attrs);
   }

   public GuestbookListScreenlet(Context context, AttributeSet attrs, int defStyleAttr) {
       super(context, attrs, defStyleAttr);
   }

   public GuestbookListScreenlet(Context context, AttributeSet attrs, int defStyleAttr,
       int defStyleRes) {
           super(context, attrs, defStyleAttr, defStyleRes);
   }
   ```

   This requires you to add the following imports:

   ```
   import android.content.Context;
   import android.util.AttributeSet;
   ```

3. Implement the error method. This is a boilerplate method that uses a listener in the Screenlet framework to propagate any exception, and the user action that produced it, that occurs during the service call. This method does this by checking for a listener and then calling its error method with the Exception and userAction:

   ```
   @Override
   public void error(Exception e, String userAction) {
       if (getListener() != null) {
           getListener().error(e, userAction);
       }
   }
   ```

4. Override the createScreenletView method. This method reads the Screenlet's attribute values via an Android TypedArray, and instantiates the View. In Guestbook List Screenlet, you only need to read the value of the groupId attribute and set it to the groupId variable. Recall that the Screens framework propagates this variable to your Interactor. Finish the createScreenletView method by calling the superclass's createScreenletView method. This instantiates the View for you:

   ```
   @Override
   protected View createScreenletView(Context context, AttributeSet attributes) {
       TypedArray typedArray = context.getTheme().obtainStyledAttributes(attributes,
           R.styleable.GuestbookListScreenlet, 0, 0);
       groupId = typedArray.getInt(R.styleable.GuestbookListScreenlet_groupId,
           (int) LiferayServerContext.getGroupId());
       typedArray.recycle();

       return super.createScreenletView(context, attributes);
   }
   ```

   Note that if the app developer doesn't set the groupId attribute, LiferayServerContext.getGroupId() is called to retrieve the app's default liferay_group_id setting from res/values/server_context.xml.

   This createScreenletView method requires you to add the following imports:

```
import android.content.res.TypedArray;
import android.view.View;
import com.liferay.docs.liferayguestbook.R;
import com.liferay.mobile.screens.context.LiferayServerContext;
```

5. Override the `loadRows` method to start your Interactor and thereby retrieve the list rows from the server. This method takes an instance of your Interactor as an argument, which you use to call the Interactor's `start` method. The `loadRows` method in `GuestbookListScreenlet` therefore starts a `GuestbookListInteractor` instance. Note that the Interactor inherits `start` from `BaseListInteractor`. Also, because you don't need to pass any data to `GuestbookListInteractor`, you can call the `start` method with `0` as an argument:

   ```
   @Override
   protected void loadRows(GuestbookListInteractor interactor) {
       interactor.start(0);
   }
   ```

6. Override the `createInteractor` method to instantiate your Interactor. Since that's all this method needs to do, call your Interactor's constructor and return the new instance:

   ```
   @Override
   protected GuestbookListInteractor createInteractor(String actionName) {
       return new GuestbookListInteractor();
   }
   ```

Awesome! Your Screenlet class is finished. Note that this Screenlet class is very similar to the one in the list Screenlet creation tutorial.

Your Screenlet is finished, too! Before using Guestbook List Screenlet, however, you'll create Entry List Screenlet to show a list of each guestbook's entries. After all, viewing guestbooks without their entries doesn't make much sense. It isn't very exciting either. What's really exciting is that you can create Entry List Screenlet with the same set of steps you used to create Guestbook List Screenlet. The next series of articles in this Learning Path walks you through this.

Chapter 44

Creating Entry List Screenlet

In the previous section, you created Guestbook List Screenlet to retrieve and display guestbooks from the Guestbook portlet. You still need a way to retrieve and display each guestbook's entries, though. You'll do this by creating another list Screenlet: Entry List Screenlet. This section walks you through the steps required to create it.

Because you use a consistent development model to create Screenlets, similar Screenlets have similar code. As with guestbooks, it makes sense to display entries in a list using a list Screenlet. This means you can reuse most of Guestbook List Screenlet's code in Entry List Screenlet. You'll therefore create Entry List Screenlet using the same sequence of steps you used to create Guestbook List Screenlet:

1. Getting started: creating the Screenlet's package and model class.
2. Creating the Screenlet's UI (its View).
3. Creating the Screenlet's Interactor.
4. Creating the Screenlet class.

Although this Learning Path section presents complete code snippets, it only discusses the code unique to Entry List Screenlet. Refer back to the previous section for detailed explanations of the code shared with Guestbook List Screenlet. If you get confused or stuck, refer to the finished app that contains the Screenlet code here in GitHub.

44.1 Getting Started with Entry List Screenlet

Like Guestbook List Screenlet, you'll create Entry List Screenlet in a new package inside your app's project. Get started by creating the package `com.liferay.docs.entrylistscreenlet`. Once you have this package, you're ready to start writing the Screenlet.

Creating the Model Class for Entries

Recall that you need a model class to represent entities retrieved from Liferay DXP. The model class you'll create for guestbook entries, `EntryModel`, creates `EntryModel` objects that serve as guestbook entries retrieved from the Guestbook portlet.

Create the following `EntryModel` class alongside the `GuestbookModel` class in the `com.liferay.docs.model` package:

```
package com.liferay.docs.model;

import android.os.Parcel;
import android.os.Parcelable;

import java.util.Date;
import java.util.Map;

public class EntryModel implements Parcelable {

    private Map values;
    private long entryId;
    private long groupId;
    private long companyId;
    private long userId;
    private String userName;
    private long createDate;
    private long modifiedDate;
    private String name;
    private String email;
    private String message;
    private long guestbookId;

    public static final Creator<EntryModel> CREATOR = new Creator<EntryModel>() {
        @Override
        public EntryModel createFromParcel(Parcel in) {
            return new EntryModel(in);
        }

        @Override
        public EntryModel[] newArray(int size) {
            return new EntryModel[size];
        }
    };

    public EntryModel() {
        super();
    }

    protected EntryModel(Parcel in) {
        entryId = in.readLong();
        groupId = in.readLong();
        companyId = in.readLong();
        userId = in.readLong();
        userName = in.readString();
        createDate = in.readLong();
        modifiedDate = in.readLong();
        name = in.readString();
        email = in.readString();
        message = in.readString();
        guestbookId = in.readLong();
    }

    public EntryModel(Map<String, Object> stringObjectMap) {
        values = stringObjectMap;
        entryId = Long.parseLong((String) stringObjectMap.get("entryId"));
        groupId = Long.parseLong((String) stringObjectMap.get("groupId"));
        companyId = Long.parseLong((String) stringObjectMap.get("companyId"));
        userId = Long.parseLong((String) stringObjectMap.get("userId"));
        userName = (String) stringObjectMap.get("userName");
        createDate = (long) stringObjectMap.get("createDate");
        modifiedDate = (long) stringObjectMap.get("modifiedDate");
        name = (String) stringObjectMap.get("name");
        email = (String) stringObjectMap.get("email");
        message = (String) stringObjectMap.get("message");
        guestbookId = Long.parseLong((String) stringObjectMap.get("guestbookId"));
    }
```

```
@Override
public void writeToParcel(Parcel dest, int flags) {
    dest.writeLong(entryId);
    dest.writeLong(groupId);
    dest.writeLong(companyId);
    dest.writeLong(userId);
    dest.writeString(userName);
    dest.writeLong(createDate);
    dest.writeLong(modifiedDate);
    dest.writeString(name);
    dest.writeString(email);
    dest.writeString(message);
    dest.writeLong(guestbookId);
}

@Override
public int describeContents() {
    return 0;
}

public Map getValues() {
    return values;
}

public void setValues(Map values) {
    this.values = values;
}

public long getEntryId() {
    return entryId;
}

public long getGroupId() {
    return groupId;
}

public long getCompanyId() {
    return companyId;
}

public long getUserId() {
    return userId;
}

public String getUserName() {
    return userName;
}

public Date getCreateDate() {
    return new Date(createDate);
}

public Date getModifiedDate() {
    return new Date(modifiedDate);
}

public String getName() {
    return name;
}

public String getEmail() {
    return email;
}

public String getMessage() {
    return message;
}
```

```
    public long getGuestbookId() {
        return guestbookId;
    }

}
```

Besides working with entries instead of guestbooks, this class works the same as `GuestbookModel`. For an explanation of the code, see the article on getting started with Guestbook List Screenlet.

Next, you'll create the Screenlet's UI.

44.2 Creating Entry List Screenlet's UI

Once you have the model class for entries, you can create the Screenlet's UI. Recall that in Liferay Screens for Android, you create a Screenlet's UI by implementing a View. In this article, you'll create Entry List Screenlet's View by using the same sequence of steps you used to create Guestbook List Screenlet's View:

1. Create the row layout. This layout defines the UI for each row in the list.
2. Create the adapter class. The adapter fills a row layout instance with the data for one list item. This repeats for each list item.
3. Create the View class. This class renders the UI, handles user interactions, and communicates with the Screenlet class.
4. Create the View's layout. This layout defines the Screenlet's UI as a whole. For a list Screenlet, this is a scrollable list.

As you follow these steps, you'll see that Entry List Screenlet's View shares a great deal of code with Guestbook List Screenlet's View. The biggest difference between these Views is that one displays guestbooks and the other displays entries. The mechanisms they use to display data, however, are almost identical.

To get started, create a new package named `view` inside the `entrylistscreenlet` package. You'll create the row layout first.

Creating the Row Layout

You must create the layout that defines the Screenlet's UI for each list row instance. Recall that in Guestbook List Screenlet, `guestbook_row.xml` serves this purpose with a single `TextView` it uses to display a guestbook's name. You'll create a similar layout here for Entry List Screenlet, but you'll use two `TextView` elements: one for the entry and one for the name of the person that left it. Create `entry_row.xml` in your app's `res/layout` directory and replace its contents with this code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:id="@+id/entry_message"
        android:textSize="25sp"
        android:textStyle="bold"
        android:paddingLeft="10dp"
        android:paddingRight="10dp"
        android:paddingBottom="1dp"
        android:paddingTop="10dp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <TextView
        android:id="@+id/entry_name"
        android:textSize="15sp"
        android:paddingLeft="10dp"
        android:paddingRight="10dp"
        android:paddingBottom="10dp"
        android:paddingTop="1dp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

You'll use the first `TextView` (`entry_message`) to display the entry, and the second `TextView` (`entry_name`) to display the name of the person that left it. The `padding` settings in each `TextView` element group the text closer together and create extra space at the top and bottom of the row. This makes it clear that the text in each row belongs together as a single list item. Of course, this is only one of many possible representations. You can style each row as you wish.

Next, you'll create your Screenlet's adapter class.

Creating the Adapter Class

Recall that an adapter class is required to fill each row with data. Entry List Screenlet's adapter class is almost identical to that of Guestbook List Screenlet. The only difference, besides working with `EntryModel` instead of `GuestbookModel`, is that it needs two variables: one for the entry and one for the name of the person who left it. In contrast, Guestbook List Screenlet's adapter class needed only one variable for the guestbook's name.

Inside the Entry List Screenlet's `view` package, create the following `EntryAdapter` class:

```
package com.liferay.docs.entrylistscreenlet.view;

import android.support.annotation.NonNull;
import android.view.View;
import android.widget.TextView;

import com.liferay.docs.liferayguestbook.R;
import com.liferay.docs.model.EntryModel;
import com.liferay.mobile.screens.base.list.BaseListAdapter;
import com.liferay.mobile.screens.base.list.BaseListAdapterListener;


public class EntryAdapter extends BaseListAdapter<EntryModel, EntryAdapter.EntryViewHolder> {

    public EntryAdapter(int layoutId, int progressLayoutId, BaseListAdapterListener listener) {
        super(layoutId, progressLayoutId, listener);
    }

    @NonNull
    @Override
    public EntryViewHolder createViewHolder(View view, BaseListAdapterListener listener) {
        return new EntryAdapter.EntryViewHolder(view, listener);
    }

    @Override
    protected void fillHolder(EntryModel entry, EntryViewHolder holder) {
        holder.bind(entry);
    }

    public class EntryViewHolder extends BaseListAdapter.ViewHolder {

        private final TextView message;
        private final TextView name;

        public EntryViewHolder(View view, BaseListAdapterListener listener) {
            super(view, listener);

            message = (TextView) view.findViewById(R.id.entry_message);
            name = (TextView) view.findViewById(R.id.entry_name);
        }

        public void bind(EntryModel entry) {
            message.setText(entry.getMessage());
            name.setText(entry.getName());
        }

    }
}
```

For an explanation of how this code works, see the section on the adapter class in the article on Creating Guestbook List Screenlet's UI.

Now you're ready to create the View class.

Creating the View Class

Recall that the View class controls a Screenlet's UI. Because Entry List Screenlet is so similar to Guestbook List Screenlet, their View classes are almost identical. The only difference is–you guessed it–one uses entries and the other uses guestbooks. For a full explanation of the View class, see the section on the View class in the article on Creating Guestbook List Screenlet's UI.

Create the `EntryListView` class inside Entry List Screenlet's `view` package, and replace its contents with this code:

```
package com.liferay.docs.entrylistscreenlet.view;

import android.content.Context;
import android.util.AttributeSet;

import com.liferay.docs.liferayguestbook.R;
import com.liferay.docs.model.EntryModel;
import com.liferay.mobile.screens.base.list.BaseListScreenletView;


public class EntryListView extends BaseListScreenletView<EntryModel,
    EntryAdapter.EntryViewHolder, EntryAdapter> {

    public EntryListView(Context context) {
        super(context);
    }

    public EntryListView(Context context, AttributeSet attributes) {
        super(context, attributes);
    }

    public EntryListView(Context context, AttributeSet attributes, int defaultStyle) {
        super(context, attributes, defaultStyle);
    }

    @Override
    protected EntryAdapter createListAdapter(int itemLayoutId, int itemProgressLayoutId) {
        return new EntryAdapter(itemLayoutId, itemProgressLayoutId, this);
    }

    @Override
    protected int getItemLayoutId() {
        return R.layout.entry_row;
    }
}
```

Fabulous work! Next, you'll create your View's main layout.

Creating the View's Layout

In the first step, you created a layout for each list row. Recall that you must also create a layout for the list as a whole. Although you may be getting tired of hearing this, it's saving you a great deal of work: Entry List Screenlet's layout is almost identical to that of Guestbook List Screenlet. The only difference is that Entry List Screenlet's layout uses `EntryListView` instead of `GuestbookListView`.

Create the file `list_entries.xml` in the `res/layout` directory, and replace its contents with this code:

```
<com.liferay.docs.entrylistscreenlet.view.EntryListView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/liferay_list_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ProgressBar
        android:id="@+id/liferay_progress"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:visibility="gone"/>

    <android.support.v7.widget.RecyclerView
        android:id="@+id/liferay_recycler_list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:visibility="gone"/>
</com.liferay.docs.entrylistscreenlet.view.EntryListView>
```

For a full explanation of this layout, see the section on creating the layout in the article on creating Guestbook List Screenlet's UI. You must also be sure not to change the layout's `android:id` values. They're hard-coded into the list Screenlet framework and changing them will cause your app to crash.

Nice job! You're done creating Entry List Screenlet's View. Next, you'll create the Screenlet's Interactor.

44.3 Creating Entry List Screenlet's Interactor

Recall that Interactors are Screenlet components that make server calls and process the results. Also recall that Interactors themselves are made up of several components:

1. The event class
2. The listener interface
3. The Interactor class

Since the list Screenlet framework already contains two listeners, you only need to create the event and Interactor classes. This article walks you through the steps required do this. Because Entry List Screenlet's Interactor is so similar to that of Guestbook List Screenlet, these steps aren't explained in detail. Focus is instead placed on the few places in the code where the Interactors diverge. For a full explanation of the code, see the article on creating Guestbook List Screenlet's Interactor.

You'll create the event class first.

Creating the Event Class

Recall that you must create an event class to communicate the server call's results via EventBus. First, create a new package called `interactor` in the `com.liferay.docs.entrylistscreenlet` package. Then create the `EntryEvent` class in the `interactor` package. Replace this class's contents with this code:

```
package com.liferay.docs.entrylistscreenlet.interactor;

import com.liferay.docs.model.EntryModel;
import com.liferay.mobile.screens.base.list.interactor.ListEvent;

public class EntryEvent extends ListEvent<EntryModel> {

    private EntryModel entry;

    public EntryEvent() {
        super();
    }

    public EntryEvent(EntryModel entry) {
        this.entry = entry;
    }

    @Override
    public String getListKey() {
        return entry.getMessage();
    }

    @Override
    public EntryModel getModel() {
        return entry;
    }
}
```

This code is almost identical to `GuestbookEvent`. The only difference is that it works with entries instead of guestbooks.

Next, you'll create the Interactor class.

Creating the Interactor Class

Recall that an Interactor class issues the server call and processes the results via the event. In the `interactor` package, create a new class called `EntryListInteractor`. Replace this class's content with this code:

```
package com.liferay.docs.entrylistscreenlet.interactor;

import com.liferay.docs.model.EntryModel;
import com.liferay.mobile.android.v7.entry.EntryService;
import com.liferay.mobile.screens.base.list.interactor.BaseListInteractor;
import com.liferay.mobile.screens.base.list.interactor.BaseListInteractorListener;
import com.liferay.mobile.screens.base.list.interactor.Query;

import org.json.JSONArray;
import java.util.Map;

public class EntryListInteractor extends
    BaseListInteractor<BaseListInteractorListener<EntryModel>, EntryEvent> {

    @Override
    protected JSONArray getPageRowsRequest(Query query, Object... args) throws Exception {

        long guestbookId = (long) args[0];
        return new EntryService(getSession()).getEntries(groupId, guestbookId,
            query.getStartRow(), query.getEndRow());
    }

    @Override
    protected Integer getPageRowCountRequest(Object... args) throws Exception {

        long guestbookId = (long) args[0];
        return new EntryService(getSession()).getEntriesCount(groupId, guestbookId);
    }

    @Override
    protected EntryEvent createEntity(Map<String, Object> stringObjectMap) {
        EntryModel entry = new EntryModel(stringObjectMap);
        return new EntryEvent(entry);
    }

    @Override
    protected String getIdFromArgs(Object... args) {
        return String.valueOf(args[0]);
    }
}
```

Besides getting entries instead of guestbooks, this class is almost identical to `GuestbookListInteractor`. The only other differences are due to the service calls that retrieve the entries and number of entries from a guestbook in the Guestbook portlet. These service calls, made in `getPageRowsRequest` and `getPageRowCountRequest`, require an `EntryService` instance. The `getEntries` method retrieves a guestbook's entries, and the `getEntriesCount` method retrieves the number of entries in a guestbook. Note that these calls require a guestbook ID (`guestbookId`) in addition to the group ID (`groupId`). The `getPageRowsRequest` and `getPageRowCountRequest` methods get the `guestbookId` from the args argument, and then use it along with `groupId` make their service calls. You'll see how the `guestbookId` gets into the args argument when you create the Screenlet class.

Nicely done! Now that Entry List Screenlet has an Interactor, you must create the Screenlet class. The next article shows you how to do this.

44.4 Creating Entry List Screenlet's Screenlet Class

Recall that when using a Screenlet, the app developer primarily interacts with its Screenlet class. The Screenlet class contains attributes for configuring the Screenlet's behavior, a reference to the Screenlet's View, methods for invoking Interactor operations, and more. This article shows you how to create Entry List Screenlet's Screenlet class.

As with most of Entry List Screenlet, its Screenlet class is almost identical to that of Guestbook List Screenlet. Besides working with entries instead of guestbooks, the only difference is that it must know its entries' guestbook ID. Even the Screenlets' attributes are the same.

Therefore, this article doesn't explain all the code in detail. Focus is instead placed on the few parts that differ from Guestbook List Screenlet. For a full explanation of the code, click here to see the article on creating Guestbook List Screenlet's Screenlet class.

You'll create Entry List Screenlet's class with the same steps you used to create Guestbook List Screenlet's class:

1. Define the Screenlet's attributes. These are the XML attributes the app developer can set when inserting the Screenlet's XML. These attributes control aspects of the Screenlet's behavior.

2. Create the Screenlet class. This class implements the Screenlet's functionality defined in the View and Interactor. It also reads the attribute values and configures the Screenlet accordingly.

First, you'll define Entry List Screenlet's attributes.

Defining Screenlet Attributes

Recall that before creating the Screenlet class, you must define the attributes the app developer needs to control the Screenlet's behavior. Entry List Screenlet, like Guestbook List Screenlet, only needs a `groupId` attribute. In your app's `res/values` directory, create `entry_attrs.xml` and replace its contents with this code:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="EntryListScreenlet">
        <attr name="groupId"/>
    </declare-styleable>
</resources>
```

Next, you'll create the Screenlet class.

Creating the Screenlet Class

Entry List Screenlet's class must contain an instance variable for the ID of the guestbook the Screenlet retrieves entries from. This is required to start the Interactor. This is the only significant difference between the Screenlet classes of Entry List Screenlet and Guestbook List Screenlet. The remaining differences exist only because they handle different entities.

Create the `EntryListScreenlet` class in the `entrylistscreenlet` package. Replace the class's content with this code:

```
package com.liferay.docs.entrylistscreenlet;

import android.content.Context;
import android.content.res.TypedArray;
import android.util.AttributeSet;
import android.view.View;

import com.liferay.docs.entrylistscreenlet.interactor.EntryListInteractor;
import com.liferay.docs.liferayguestbook.R;
import com.liferay.docs.model.EntryModel;
import com.liferay.mobile.screens.base.list.BaseListScreenlet;
import com.liferay.mobile.screens.context.LiferayServerContext;

public class EntryListScreenlet extends BaseListScreenlet<EntryModel, EntryListInteractor> {

    private long guestbookId;

    public EntryListScreenlet(Context context) {
        super(context);
    }

    public EntryListScreenlet(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public EntryListScreenlet(Context context, AttributeSet attrs, int defStyleAttr) {
        super(context, attrs, defStyleAttr);
    }

    public EntryListScreenlet(Context context, AttributeSet attrs, int defStyleAttr, int defStyleRes) {
        super(context, attrs, defStyleAttr, defStyleRes);
    }

    @Override
    public void error(Exception e, String userAction) {
        if (getListener() != null) {
            getListener().error(e, userAction);
        }
    }
    @Override
    protected View createScreenletView(Context context, AttributeSet attributes) {
        TypedArray typedArray = context.getTheme().obtainStyledAttributes(attributes,
            R.styleable.GuestbookListScreenlet, 0, 0);
        groupId = typedArray.getInt(R.styleable.GuestbookListScreenlet_groupId,
            (int) LiferayServerContext.getGroupId());
        typedArray.recycle();

        return super.createScreenletView(context, attributes);
    }

    @Override
    protected void loadRows(EntryListInteractor interactor) {
        interactor.start(guestbookId);
    }

    @Override
    protected EntryListInteractor createInteractor(String actionName) {
        return new EntryListInteractor();
    }

    public long getGuestbookId() {
        return guestbookId;
    }

    public void setGuestbookId(long guestbookId) {
        this.guestbookId = guestbookId;
    }
}
```

The instance variable for the guestbook ID is `guestbookId`. The getter and setter methods `getGuestbookId` and `setGuestbookId` let the app developer get and set this variable, respectively. The `loadRows` method starts the Interactor by calling the `start` method with `guestbookId` as an argument. Behind the scenes, the list Screenlet framework passes `guestbookId` to the Interactor's `getPageRowsRequest` and `getPageRowCountRequest` methods via the args argument. This is why you were able to extract `guestbookId` from the args argument in these methods. For an explanation of how the rest of this Screenlet class works, click here to see the article on creating Guestbook List Screenlet's Screenlet class.

That's it! Now you're ready to use Entry List Screenlet alongside Guestbook List Screenlet. The following section of this Learning Path concludes with both Screenlets working together in harmony.

Chapter 45

Using the Guestbook List and Entry List Screenlets

Now that you have the Guestbook List and Entry List Screenlets, you're ready to put them to work. As you'll see, using these Screenlets isn't much more difficult than using Login Screenlet. This is an advantage of Screenlets; it typically takes only a few minutes to get them up and running. They also integrate with the rest of your app's UI.

To add your Screenlets to the app, you'll follow these steps:

1. Understand how `GuestbooksActivity`'s UI works. Since your Screenlets augment this UI instead of replacing it, you should first understand how it works.

2. Prepare `GuestbooksActivity` for Guestbook List Screenlet.

3. Use Guestbook List Screenlet by inserting it in `GuestbooksActivity`.

4. Create `EntriesFragment` for Entry List Screenlet. You'll also set `GuestbooksActivity` to display this fragment when a guestbook is selected in Guestbook List Screenlet.

5. Use Entry List Screenlet by inserting it in `EntriesFragment`.

If you get confused or stuck at any point in this section of the Learning Path, refer to the finished app's code here in GitHub.

First, you'll see how `GuestbooksActivity`'s UI works.

45.1 Understanding GuestbooksActivity's UI

Recall that you used Android Studio's Navigation Drawer Activity template to create `GuestbooksActivity`. Any activity this template creates contains a navigation drawer and all the components the activity needs. This includes layout files that display content. Currently, these files contain simple placeholder content. You'll replace this content shortly with content from your Guestbook portlet. Before doing so, however, you should know where the placeholder content exists in the project's structure and how the app displays it.

The app's UI is defined by three layout files that combine to display the app's content:

1. activity_guestbooks.xml: The activity's main layout file. This layout defines the navigation drawer and includes the app_bar_guestbooks layout. The latter appears when the navigation drawer is closed.

2. app_bar_guestbooks.xml: Defines the action bar (app bar) and includes the content_guestbooks layout.

3. content_guestbooks.xml: Defines the activity's main content, which appears below the action bar.

First, you'll learn how activity_guestbooks.xml works.

Understanding the Activity's Main Layout File

First, open GuestbooksActivity's main layout file, activity_guestbooks.xml. This file should look similar to this one:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start">

    <include layout="@layout/app_bar_guestbooks"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header_guestbooks"
        app:menu="@menu/activity_guestbooks_drawer" />

</android.support.v4.widget.DrawerLayout>
```

This isn't much code considering everything that's in GuestbooksActivity. The NavigationView and its parent DrawerLayout define the navigation drawer. Two attributes in NavigationView define the drawer's contents: app:headerLayout and app:menu. The app:headerLayout value "@layout/nav_header_guestbooks" specifies that the layout res/layout/nav_header_guestbooks.xml renders the drawer's header section. The app:menu value "@menu/activity_guestbooks_drawer" specifies that the menu in res/menu/activity_guestbooks_drawer.xml creates the drawer's items. Above the NavigationView, the include statement adds the layout app_bar_guestbooks.xml as the content shown when the navigation drawer is closed. The following diagram illustrates how activity_guestbooks.xml maps to the UI.

The activity's main content also contains a toolbar (the action bar), some text, and a floating action button. Next, you'll see how these are defined.

Understanding the app_bar_guestbooks and content_guestbooks Layouts

Open app_bar_guestbooks.xml. It should look similar to this:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
```

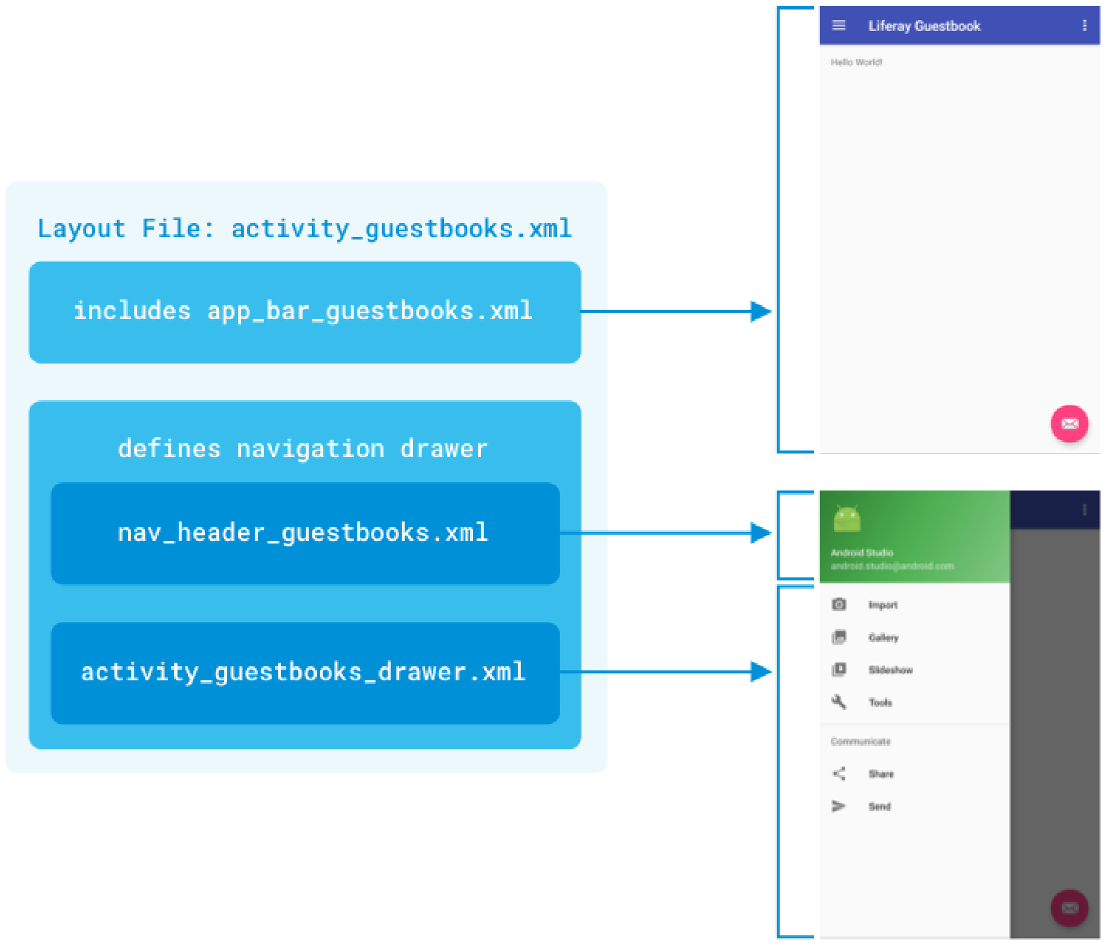

Layout File: activity_guestbooks.xml
includes app_bar_guestbooks.xml
defines navigation drawer
nav_header_guestbooks.xml
activity_guestbooks_drawer.xml
Liferay Guestbook
Hello World!
Android Studio
android.studio@android.com
Import
Gallery
Slideshow
Tools
Communicate
Share
Send

Figure 45.1: The `activity_guestbooks.xml` layout defines the app's main UI components.

```
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:fitsSystemWindows="true"
tools:context="com.liferay.docs.liferayguestbook.GuestbooksActivity">

<android.support.design.widget.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/AppTheme.AppBarOverlay">

    <android.support.v7.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="?attr/actionBarSize"
        android:background="?attr/colorPrimary"
        app:popupTheme="@style/AppTheme.PopupOverlay" />
    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_guestbooks" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:src="@android:drawable/ic_dialog_email" />

</android.support.design.widget.CoordinatorLayout>
```

The AppBarLayout and Toolbar elements define the toolbar at the top of the activity. Following the toolbar definition, the include statement adds the content_guestbooks layout to app_bar_guestbooks. The content_guestbooks layout defines the content displayed in the activity's body (below the toolbar). Right now, this layout only contains an empty ConstraintLayout element. Now open content_guestbooks.xml. Its contents should look similar to this:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="@string/appbar_scrolling_view_behavior"
    tools:context="com.liferay.docs.liferayguestbook.GuestbooksActivity"
    tools:showIn="@layout/app_bar_guestbooks">

</android.support.constraint.ConstraintLayout>
```

Anything you define in this layout becomes the activity's main body content. Later, you'll return to content_guestbooks.xml to display the guestbook entries retrieved from the Guestbook portlet.

Now return to app_bar_guestbooks.xml. This layout concludes by using FloatingActionButton to define the floating action button. Pressing this button in the app slides a snackbar containing placeholder content up from the bottom of the screen. Although you won't do anything with the floating action button in this Learning Path, you'll leave it in place. When you finish this Learning Path, you may want to test your Liferay mobile development chops by adding functionality to this button.

The following figure illustrates how the app_bar_guestbooks layout maps to the activity's UI. On the left, this figure lists each UI component in app_bar_guestbooks. Each arrow points to the component's rendering on the right.

Awesome! Now you know which layout files in the project define the app's UI. You also know the exact UI components these files define. Next, you'll prepare GuestbooksActivity for Guestbook List Screenlet.

45.2 Preparing GuestbooksActivity for Guestbook List Screenlet

Recall that you want GuestbooksActivity to display Guestbook List Screenlet and Entry List Screenlet. Before using these Screenlets, however, you must prepare GuestbooksActivity as follows:

1. Refactor the action bar so you can later set its title to the selected guestbook's name.

2. Refactor the navigation drawer so you can later close it when a guestbook is selected in Guestbook List Screenlet.

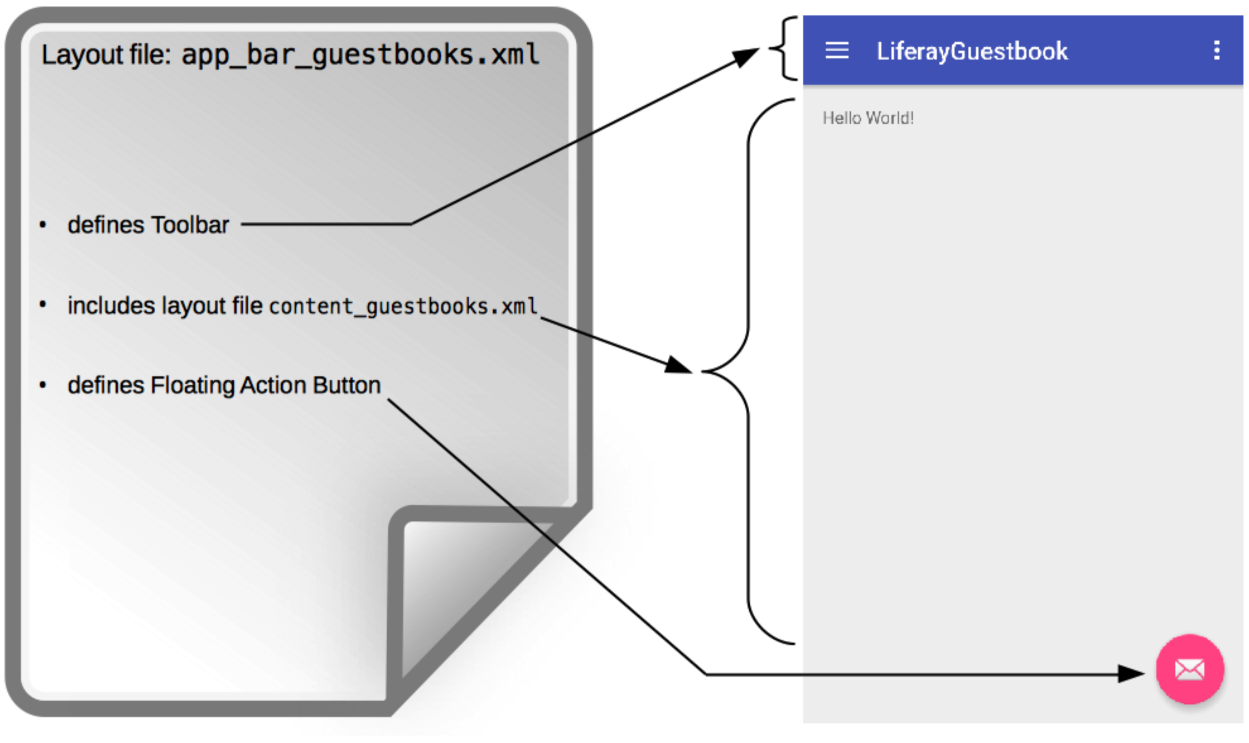

Layout file: app_bar_guestbooks.xml
defines Toolbar
includes layout file content_guestbooks.xml
defines Floating Action Button
LiferayGuestbook
Hello World!

Figure 45.2: The layout `app_bar_guestbooks.xml` defines the activity's main content.

3. Delete the `NavigationView.OnNavigationItemSelectedListener` implementation. Since Guestbook List Screenlet handles guestbook selections, you don't need `NavigationView`.

When you finish, you'll be ready to use Guestbook List Screenlet. Note that you won't always have to take steps like these before using Screenlets. You only do so here to fit this particular app's design.

First, you'll refactor the action bar.

Refactoring the Action Bar

By default, the action bar displays the activity's name. When you use Guestbook List Screenlet, you want the action bar to display the selected guestbook's name instead. You'll enable this by modifying the code that creates the action bar. Android Studio created this code for you in the `GuestbooksActivity` class's `onCreate` method:

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

1. Remove this code. Although you could edit it, you'll instead create a separate method that creates the action bar. Note that you don't need to worry about the now missing `toolbar` variable in `onCreate`; you'll fix it shortly.

2. Create `ActionBar` and `Toolbar` instance variables. This lets you refer to them anywhere in the activity. Add these variables to the `GuestbooksActivity` class:

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
```

This requires that you import `android.support.v7.app.ActionBar`.

3. Add the following `initActionBar()` method to `GuestbooksActivity`:

```
private void initActionBar() {
    toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);
    actionBar = getSupportActionBar();
    actionBar.setTitle("");
}
```

   Like the code you removed from `onCreate`, this method also creates a `Toolbar` and sets it as the action bar. It also sets the action bar's title to an empty string. This prevents the activity's title from showing in the action bar before the app can retrieve guestbooks from the portlet.

4. Call `initActionBar()` in `onCreate`. Place the call immediately below the `setContentView` call. The first few lines of `onCreate` should now look like this:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_guestbooks);

    initActionBar();

    ...
}
```

Next, you'll modify the code that controls the navigation drawer.

Refactoring the Navigation Drawer

Before you can use Guestbook List Screenlet in the navigation drawer, you must refactor the drawer's existing code. Do so now by following these steps:

1. Currently, the navigation drawer initialization code is in the `onCreate` method. Android Studio created this code for you when you used the Navigation Drawer Activity template to create `GuestbooksActivity`. Delete this code from `onCreate`:

```
DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(
    this, drawer, toolbar, R.string.navigation_drawer_open, R.string.navigation_drawer_close);
drawer.setDrawerListener(toggle);
toggle.syncState();
```

   Instead, you'll initialize the navigation drawer in a separate method that you'll call in `onCreate`. You'll create this method shortly.

2. You'll also change the `drawer` variable to be an instance variable that you can refer to throughout the class. This lets you use this variable to close the drawer when a guestbook is selected in Guestbook List Screenlet. Add this variable to `GuestbooksActivity`:

```
private DrawerLayout drawer;
```

3. Add the following `initDrawer` method. This method's contents match the drawer initialization code you deleted in `onCreate`, except that `drawer` is now an instance variable:

```
private void initDrawer() {
    // drawer initialization
    drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
    ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(
        this, drawer, toolbar, R.string.navigation_drawer_open,
        R.string.navigation_drawer_close);
    drawer.setDrawerListener(toggle);
    toggle.syncState();
}
```

4. In the `onCreate` method, place the call to `initDrawer()` immediately below the `initActionBar` call. The first few lines of `onCreate` should now look like this:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_guestbooks);

    initActionBar();
    initDrawer();

    ...
}
```

5. Also, because you want to use the same `DrawerLayout` instance throughout the class, delete the line of code that creates a new `DrawerLayout` in the `onBackPressed` method. Your `onBackPressed` method should now look like this:

```
@Override
public void onBackPressed() {
    if (drawer.isDrawerOpen(GravityCompat.START)) {
        drawer.closeDrawer(GravityCompat.START);
    } else {
        super.onBackPressed();
    }
}
```

Now you're ready to delete the `NavigationView.OnNavigationItemSelectedListener` implementation. The next section walks you through this.

Deleting the NavigationView.OnNavigationItemSelectedListener Implementation

Since Guestbook List Screenlet handles navigation drawer item selections, you don't need to implement `NavigationView.OnNavigationItemSelectedListener` in `GuestbooksActivity`. Follow these steps to remove this implementation:

1. Delete the implementation from the class declaration. The class declaration should now look like this:

```
public class GuestbooksActivity extends AppCompatActivity {...
```

2. Remove the code in `GuestbooksActivity` that implements `NavigationView.OnNavigationItemSelectedListener`. To do this, first delete the following code at the end of the `onCreate` method:

```
NavigationView navigationView = (NavigationView) findViewById(R.id.nav_view);
navigationView.setNavigationItemSelectedListener(this);
```

Your onCreate method should now look like this:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_guestbooks);

    initActionBar();
    initDrawer();

    FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action", Snackbar.LENGTH_LONG)
                .setAction("Action", null).show();
        }
    });
}
```

3. Delete the `onNavigationItemSelected` method, along with its `@Override` and `@SuppressWarnings("StatementWithEmptyBody`
   statements.

4. Finally, remove the `android.support.design.widget.NavigationView` import.

Great job! Now you're ready to insert Guestbook List Screenlet in `GuestbooksActivity`.

45.3 Using Guestbook List Screenlet

The steps for using Guestbook List Screenlet are the same as those for using any Screenlet:

1. Insert the Screenlet's XML in the activity or fragment layout you want the Screenlet to appear in.

2. Implement the Screenlet's listener in the activity or fragment class.

Recall that you used these steps to insert Login Screenlet in `MainActivity`. First, you'll insert Guestbook List Screenlet's XML in `GuestbooksActivity`'s layout.

Inserting the Screenlet XML in the Layout

Recall that `activity_guestbooks.xml` defines `GuestbooksActivity`'s UI. Also recall that the `NavigationView` in `activity_guestbooks.xml` defines the navigation drawer.

To put Guestbook List Screenlet in the drawer, you must insert the Screenlet's XML in the `NavigationView`. You must also remove the placeholder content from the `NavigationView`. To do these things, replace the `NavigationView` in `activity_guestbooks.xml` with this code:

```
<android.support.design.widget.NavigationView
    android:id="@+id/nav_view"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_gravity="start"
    android:fitsSystemWindows="true"
    app:headerLayout="@layout/nav_header_guestbooks">
    <com.liferay.docs.guestbooklistscreenlet.GuestbookListScreenlet
        android:id="@+id/guestbooklist_screenlet"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:paddingTop="@dimen/nav_header_height"
        app:layoutId="@layout/list_guestbooks"/>

</android.support.design.widget.NavigationView>
```

Compared to the `NavigationView` it replaced, this `NavigationView` contains Guestbook List Screenlet's XML and lacks the `app:menu` attribute. Recall that this attribute pointed to the menu resource file that creates the drawer's items. Since the Screenlet now handles the drawer's items (the guestbooks), you don't need `app:menu` or the menu resource file. Delete the menu resource file `res/menu/activity_guestbooks_drawer.xml`. You also don't need the drawable resources that Android Studio created for the navigation drawer's placeholder content. Delete `res/values/drawables.xml`, and each of the `ic_menu_*.xml` files in `res/drawable`.

Returning your attention to `activity_guestbooks.xml`, note that Guestbook List Screenlet's XML strongly resembles Login Screenlet's XML. Both contain an `android:id` value that you can use in the activity to get a reference to the Screenlet. Both also use a `layoutId` attribute to specify the Screenlet's View. Guestbook List Screenlet's XML, however, differs by using the `android:paddingTop` attribute. This attribute's value, `@dimen/nav_header_height`, pads the top of the Screenlet by the height of the navigation drawer's header section. This prevents the Screenlet and drawer header from overlapping.

Great! Next, you'll implement the Screenlet's listener interface in `GuestbooksActivity`.

Implementing the Screenlet's Listener

To use a Screenlet, you must implement its listener methods in the class of the activity or fragment where you want the Screenlet to appear. How you implement these methods depends on how you want the Screenlet to function in your app. For example, when you used Login Screenlet you implemented `LoginListener` in `MainActivity`. You implemented this listener's `onLoginSuccess` and `onLoginFailure` methods to display a message to the user. You then changed the `onLoginSuccess` implementation to navigate from `MainActivity` to `GuestbooksActivity`. Since these methods are `void`, however, you could have left them empty. Obviously this wouldn't have made for a very useful app, but it highlights an important point: Screenlet listener methods let the app developer choose how to respond to the Screenlet's events. By implementing these methods, app developers can therefore control how the Screenlet functions with their app.

Before implementing Guestbook List Screenlet's listener, however, you should add a method that the listener methods can use to help display a guestbook and its entries. You might now be thinking, "I thought you said Screenlets contain their own UIs? Why does the activity need special methods for displaying the Screenlets' entities?" Although a list Screenlet's UI displays the list of entities, the rest of the app's UI must still account for that list. Consider the action bar, for example. List Screenlets don't include an action bar, but you should still change the action bar's contents to reflect what's on the screen. When a guestbook is selected in Guestbook List Screenlet, the action bar should display that guestbook's name. You can accomplish this by calling a method that takes a `GuestbookModel` and sets that guestbook's name as the action bar's title.

Follow these steps to add this method and implement the listener:

1. Add this `showEntries` method to `GuestbooksActivity`:

```
    public void showEntries(GuestbookModel guestbook) {

        actionBar.setTitle(guestbook.getName());
    }
```

This requires you to import `com.liferay.docs.model.GuestbookModel`. This method is called `showEntries` because you'll also use it to display the guestbook's entries via Entry List Screenlet (you'll add this code later). You'll call this method in the listener methods you'll implement to process a guestbook selection.

2. Recall that Guestbook List Screenlet doesn't need any custom listener methods. It can use the listener methods defined in the list Screenlet framework's `BaseListListener` interface. Change `GuestbooksActivity`'s class declaration to implement `BaseListListener<GuestbookModel>`. The class declaration should now look like this:

```
public class GuestbooksActivity extends AppCompatActivity implements
    BaseListListener<GuestbookModel> {...
```

   This requires you to import `com.liferay.mobile.screens.base.list.BaseListListener`.

3. To implement `BaseListListener`, you must implement the following methods:

   - `onListPageFailed(int startRow, Exception e)`: Called when the server call to retrieve a page of items fails. This method's arguments include the `Exception` generated when the server call failed. Implement this method to show the user a toast message containing an error:

```
@Override
public void onListPageFailed(int startRow, Exception e) {

    Toast.makeText(this, "Page request failed", Toast.LENGTH_LONG).show();
}
```

     This requires you to import `android.widget.Toast`.

   - `onListPageReceived(int startRow, int endRow, List<E> entries, int rowCount)`: Called when the server call to retrieve a page of items succeeds. Note that this method's arguments include the list of objects retrieved from the server (`entries`), and the page's start row (`startRow`), and end row (`endRow`). Recall that by default, you want the activity to display the first guestbook's entries. You'll use this method to do so because it receives the guestbooks from the server. Note that because `startRow` and `endRow` change for each page, a `startRow` of `0` corresponds to the first guestbook on the first page. Use an `if` statement to select this guestbook, and then call `showEntries`:

```
@Override
public void onListPageReceived(int startRow, int endRow, List<GuestbookModel> guestbooks,
    int rowCount) {

    if (startRow == 0) {
        showEntries(guestbooks.get(0));
    }
}
```

     This requires you to import `java.util.List`.

   - `onListItemSelected(E element, View view)`: Called when the user selects an item in the list. This method's arguments include the selected list item (`element`). To process the guestbook's selection, call `showEntries` in this method. Also, close the navigation drawer following the `showEntries` call:

```
    @Override
    public void onListItemSelected(GuestbookModel guestbook, View view) {

        showEntries(guestbook);
        drawer.closeDrawers();
    }
```

4. Because BaseListListener extends the BaseCacheListener interface, the activity must also implement BaseCacheListener's error method. This method lets you respond to an error alongside the user action that caused it. In this app, you don't need to do anything in this method, so you can leave its contents empty:

```
@Override
public void error(Exception e, String userAction) {

}
```

5. Now that you've implemented the listener methods, you must set GuestbooksActivity as the listener. This is where the guestbooklist_screenlet ID that you set in the Screenlet's XML comes in handy. Add the following code to the end of the activity's onCreate method:

```
GuestbookListScreenlet screenlet =
    (GuestbookListScreenlet) findViewById(R.id.guestbooklist_screenlet);
screenlet.setListener(this);
```

   This requires you to import com.liferay.docs.guestbooklistscreenlet.GuestbookListScreenlet.

   This code first uses the ID guestbooklist_screenlet to get a reference to GuestbookListScreenlet. It then sets this GuestbooksActivity instance as the Screenlet's listener.

Great! That's it! Your app's GuestbooksActivity now contains Guestbook List Screenlet. You're almost ready to use Entry List Screenlet. Before you do so, however, you must create a fragment to put it in. You'll do this next.

45.4 Creating a Fragment for Entry List Screenlet

Using a fragment for Entry List Screenlet lets you swap out part of GuestbookActivity's contents without recreating the entire activity from scratch each time a guestbook is selected. Your app doesn't currently have any fragments, though. In this step, you'll create a fragment and then add it to GuestbooksActivity. When you finish, you'll be ready to use Entry List Screenlet in this fragment.

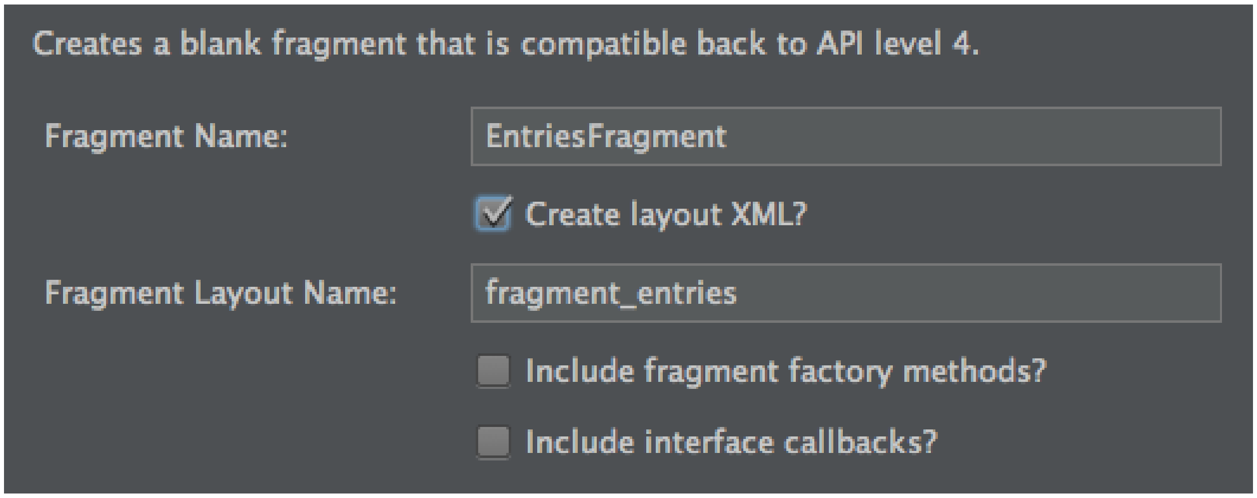
Creating the Fragment

Follow these steps to create the fragment:

1. To create the fragment, right click the com.liferay.docs.liferayguestbook package and select *New → Fragment → Fragment (Blank)*. In the wizard, check only the box to create the layout XML, name the fragment EntriesFragment, and then click *Finish*. The following screenshot illustrates this:

   This creates the EntriesFragment class and its layout file fragment_entries.xml.

2. Replace the class's contents with this code:


Creates a blank fragment that is compatible back to API level 4.

Fragment Name: EntriesFragment

☑ Create layout XML?

Fragment Layout Name: fragment_entries

☐ Include fragment factory methods?

☐ Include interface callbacks?

Figure 45.3: Create a new blank fragment for the entries.

```
package com.liferay.docs.liferayguestbook;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class EntriesFragment extends Fragment {

    public static EntriesFragment newInstance(long guestbookId) {
        EntriesFragment entriesFragment = new EntriesFragment();
        Bundle args = new Bundle();
        args.putLong("guestbookId", guestbookId);
        entriesFragment.setArguments(args);

        return entriesFragment;
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                     Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        View view = inflater.inflate(R.layout.fragment_entries, container, false);
        long guestbookId = getArguments().getLong("guestbookId");

        return view;
    }

}
```

If you have experience with Android fragments, then you're likely familiar with the static `newInstance` method. In short, using such a method instead of an empty constructor lets you create the fragment and initialize its data in one step. This makes it easier to create and restore the fragment. Click here for more information.

Since this fragment will contain Entry List Screenlet, its data must include the guestbook ID the Screenlet retrieves entries from (`guestbookId`). Also, the `onCreateView` method uses the bundle arguments set in newInstance to retrieve the guestbookId. For now, you don't have to do anything with the guestbookId in onCreateView. You'll use this variable when you add the Screenlet to the fragment.

Next, you'll add this fragment to GuestbooksActivity.

Adding the Fragment to GuestbooksActivity

Now that EntriesFragment exists, you can add it to GuestbooksActivity. To do this, you must put an Android fragment container in the layout where you want the fragment. For more information, see Android's documentation on adding fragments at runtime. Since you want Entry List Screenlet to appear in GuestbooksActivity, your first thought might be to put the fragment container directly in activity_guestbooks.xml. Don't do this. Recall that Android Studio's Navigation Drawer Activity template created the layout content_guestbooks.xml to hold the activity's main body content. You'll add the fragment container to this layout, then write the code in GuestbooksActivity that displays the fragment.

Follow these steps to add the fragment to GuestbooksActivity:

1. Open content_guestbooks.xml and place the following fragment container inside the ConstraintLayout. This fragment container should be the only other element inside the ConstraintLayout:

```
<FrameLayout
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

2. Next, you must write the GuestbooksActivity code that displays the fragment when a guestbook is selected in Guestbook List Screenlet. You'll do this with an Android fragment transaction. Recall that you created GuestbooksActivity's showEntries method to process a list item selection in Guestbook List Screenlet. All showEntries does right now is set the action bar's title to the selected guestbook's name. You'll add the fragment transaction to showEntries, so a guestbook selection also shows that guestbook's entries. Replace the showEntries method in GuestbooksActivity with this code:

```
public void showEntries(GuestbookModel guestbook) {
    actionBar.setTitle(guestbook.getName());

    EntriesFragment entriesFragment = EntriesFragment.newInstance(guestbook.getGuestbookId());
    FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
    transaction.replace(R.id.fragment_container, entriesFragment);
    transaction.commit();
}
```

This requires that you import android.support.v4.app.FragmentTransaction.

This method's actionBar.setTitle call is the same as before. Only the fragment code is new. In it, you first use newInstance to create a new EntriesFragment instance with the selected guestbook's ID. A fragment transaction then adds this fragment to the fragment container.

Fantastic! Now you have a fragment to put Entry List Screenlet in, and the code that displays this fragment in GuestbooksActivity. Next, you'll put Entry List Screenlet in the fragment.

45.5 Using Entry List Screenlet

You'll use Entry List Screenlet by following the same steps to use any Screenlet: insert the Screenlet's XML in an activity or fragment layout, and then implement the Screenlet's listener in the activity or fragment class. You'll follow these steps here to insert Entry List Screenlet in `EntriesFragment`.

First, you'll insert Entry List Screenlet's XML in `EntriesFragment`'s layout, `fragment_entries.xml`.

Inserting the Screenlet in the Layout

Inserting Entry List Screenlet's XML is very simple. Since all you want `fragment_entries.xml` to do is display the Screenlet, it must only contain the Screenlet's XML. Replace the contents of `fragment_entries.xml` with the following markup:

```
<com.liferay.docs.entrylistscreenlet.EntryListScreenlet
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/entrylist_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layoutId="@layout/list_entries"/>
```

As with Guestbook List Screenlet, you'll use the `android:id` value to get a Screenlet reference. Next, you must implement the Screenlet's listener.

Implementing the Screenlet's Listener

Recall that to use a Screenlet, you must implement its listener. The listener methods let the app developer respond to the Screenlet's behavior in the activity or fragment class that contains the Screenlet. Also recall that because Guestbook List Screenlet didn't need extra listener methods, you used it in `GuestbooksActivity` by implementing the `BaseListListener` interface with `GuestbookModel` as a type argument. Entry List Screenlet also doesn't need extra listener methods. Like Guestbook List Screenlet, you can use it by implementing `BaseListListener` with its model class as a type argument.

Follow these steps to implement Entry List Screenlet's listener in `EntriesFragment`:

1. Change `EntriesFragment`'s class declaration to implement `BaseListListener<EntryModel>`. The class declaration should now look like this:

   ```
   public class EntriesFragment extends Fragment implements BaseListListener<EntryModel> {...
   ```

   This requires that you add the following imports:

   ```
   import com.liferay.docs.model.EntryModel;
   import com.liferay.mobile.screens.base.list.BaseListListener;
   ```

2. Now you must implement the listener's methods. Recall that this includes the `BaseCacheListener` interface's, `error` method, since `BaseListListener` extends `BaseCacheListener`. For a full explanation of the methods in both listeners, see using Guestbook List Screenlet. Note that in `EntriesFragment`, you don't need to take any action in these methods. There are no UI elements or other parts of the fragment that must be updated or processed in response to the Screenlet's behavior. All this Screenlet must do is display its content, which it does regardless of anything you do in the listener methods. The only thing you may want to add is a toast message in `onListPageFailed` to notify the user if the server call fails, but this isn't required. Implement these methods now:

```
@Override
public void onListPageFailed(int startRow, Exception e) {
    Toast.makeText(getActivity(), "Page request failed", Toast.LENGTH_LONG).show();
}

@Override
public void onListPageReceived(int startRow, int endRow, List<EntryModel> entries, int rowCount) {

}

@Override
public void onListItemSelected(EntryModel entry, View view) {

}

@Override
public void error(Exception e, String userAction) {

}
```

This requires you to add the following imports:

```
import android.widget.Toast;
import java.util.List;
```

3. Now you're ready to register `EntriesFragment` as the Screenlet's listener. You'll do this the same way you registered `GuestbooksActivity` as Guestbook List Screenlet's listener: get a reference to the Screenlet and call its `setListener` method. After doing this, you'll use the Entry List Screenlet reference's `setGuestbookId` method to set its guestbook ID. This sets the guestbook where the Screenlet gets its entries. You'll do these things in the `onCreateView` method. Replace the `onCreateView` method with this updated version:

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                     Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    View view = inflater.inflate(R.layout.fragment_entries, container, false);
    long guestbookId = getArguments().getLong("guestbookId");

    EntryListScreenlet screenlet = (EntryListScreenlet) view.findViewById(R.id.entrylist_screenlet);
    screenlet.setListener(this);
    screenlet.setGuestbookId(guestbookId);

    return view;
}
```

This requires you to import `com.liferay.docs.entrylistscreenlet.EntryListScreenlet`.

As you can see, `onCreateView` now registers `EntriesFragment` as the Screenlet's listener and sets the Screenlet's guestbook ID. The rest of `onCreateView` is unchanged.

Now run the app in the emulator and log in with your credentials when prompted. The app then presents you with the first guestbook's entries. Open the navigation drawer by pressing the hamburger button, then select a different guestbook. The drawer then closes to show the selected guestbook's entries. Nice work! Your app now uses Guestbook List Screenlet and Entry List Screenlet to show the same guestbooks and entries as the Guestbook portlet. The following screenshots show these Screenlets in action.

Although your Screenlets work, you may have noticed something odd about the navigation drawer's header–it's hideous. The action bar is somewhere on the purple-blue spectrum, while the drawer header is

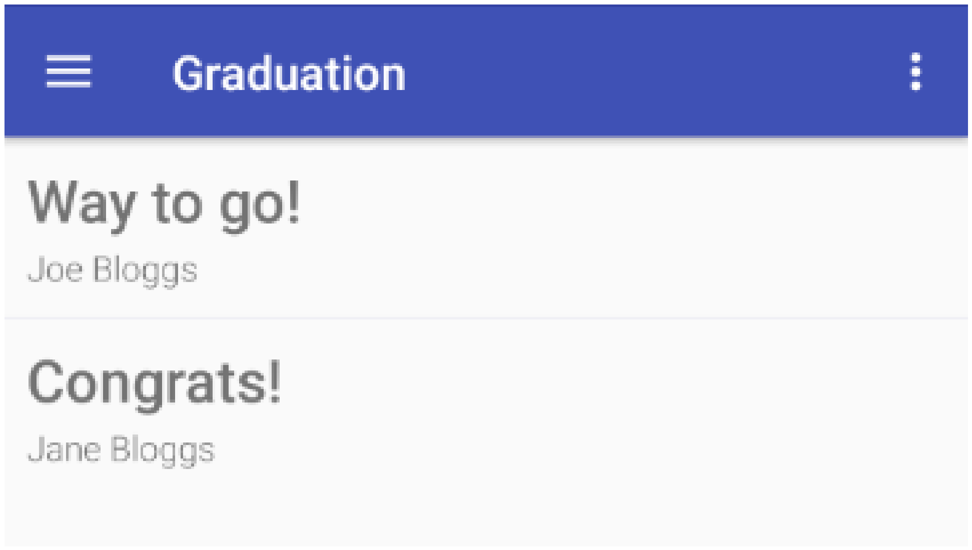


Figure 45.4: Entry List Screenlet displays guestbook entries in your app.

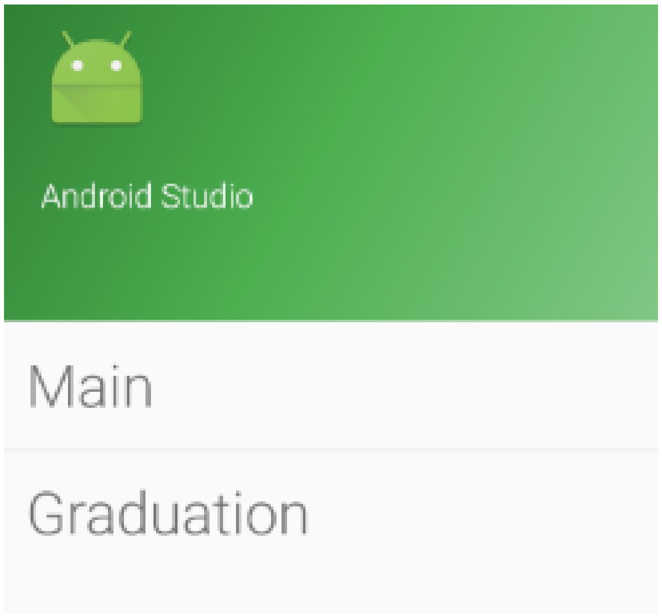


Figure 45.5: Guestbook List Screenlet displays guestbooks in the navigation drawer.

green. You've probably seen more attractive finger paintings. Fortunately, it's simple to change the drawer header's color. Also, the drawer header contains the generic text *Android Studio*. You should change this to something more suitable for your app, like *Liferay Guestbook*.

Follow these steps to apply these changes to the drawer header:

1. In `res/drawable/side_nav_bar.xml`, replace `android:centerColor`, `android:endColor`, and `android:startColor` with the following settings:

   ```
   android:centerColor="@color/colorPrimary"
   android:endColor="@color/colorPrimaryDark"
   android:startColor="@color/colorPrimary"
   ```

   This sets the drawer header's colors to match the colors used in the rest of the app.

2. Define the following string resource in `res/values/strings.xml`:

   ```
   <string name="liferay_guestbook">Liferay Guestbook</string>
   ```

3. In `nav_header_guestbooks.xml`, find the `TextView` element that contains `android:text="Android Studio"`, and replace `Android Studio` with `@string/liferay_guestbook`. You can delete any other `TextView` elements in this file. Run the app again, and open the drawer after signing in. The drawer header now shows your greeting. It's a lot prettier too.

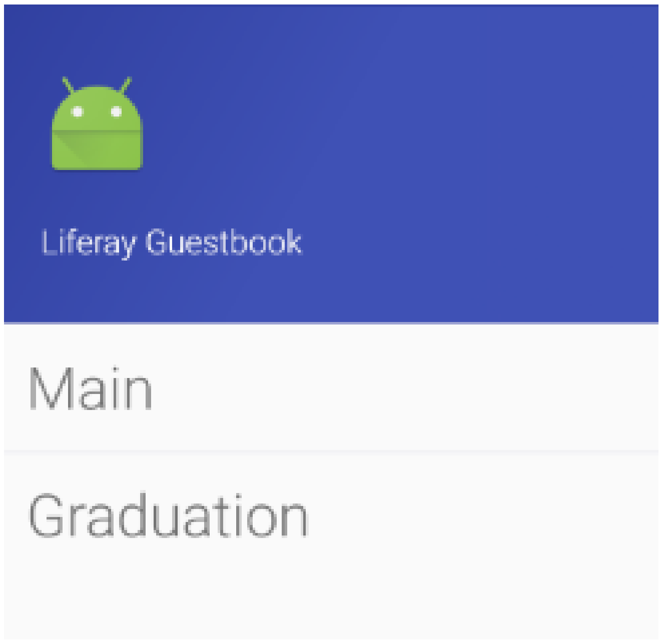


Figure 45.6: The drawer header looks a lot better after some light customization.

Congratulations! Now you know how to use Liferay Screens and create your own Screenlets. This opens up a world of possibilities for developing apps that leverage Liferay DXP. Although you learned a great deal in this Learning Path, there's still more. You can customize your Screenlet's appearance, package it for redistribution, and even configure it to receive push notifications. These topics, and more, are covered in the tutorials on Android apps with Liferay Screens.

Chapter 46

Writing an iOS App with Liferay Screens

Users expect to access portal content from their mobile devices. As an intrepid developer, you naturally want to turn these expectations into reality. Thankfully, Liferay provides a way for your mobile apps to access portal content and applications with Liferay Screens! Screens contains native components called *Screenlets* that can call Liferay DXP's remote services and display the results in your app. Each Screenlet comes complete with its own fully pluggable UI that you can customize to your liking. Although the Screenlets included with Screens only work with Liferay DXP's built-in remote services, you can write your own Screenlets that work with your custom portlets' remote services.

If you're an experienced iOS developer but need a start-to-finish guide on how to integrate iOS apps with Liferay DXP, you're in the right place. This Learning Path walks you through the creation of an iOS app that interacts with the Guestbook portlet developed in the Developing a Web Application Learning Path. Since this is a custom portlet, you'll write your own Screenlets that let your app retrieve and display guestbooks and their entries.

You should note that although this Learning Path provides complete code snippets of the app, not every aspect of iOS development is explained in detail. Focus is instead placed on the code that leverages Liferay Screens. Therefore, you **must** have significant iOS development experience before attempting this Learning Path. Otherwise, you'll likely be confused. Apple provides extensive documentation of the iOS APIs as well as some basic tutorials on developer.apple.com.

Experience in iOS development is all you need to start working. You needn't have completed the Developing a Web Application Learning Path to obtain a working Guestbook portlet. The complete Guestbook portlet's modules are provided for installation into your local Liferay DXP instance.

Now that you know what you'll be doing here, it's time to move on to the first series of articles: Beginning iOS Development for Your Portal. These articles walk you through the steps required to get started developing an iOS app that interacts with Liferay DXP.

Chapter 47

Beginning iOS Development for Your Portal

Getting started with Liferay Screens for iOS is straightforward. This series of Learning Path articles walks you through creating an iOS app and preparing it to work with the Guestbook portlet developed in the Developing a Web Application Learning Path. Since Liferay Screens uses the Liferay Mobile SDK to make remote service calls, you'll build a Mobile SDK capable of calling the Guestbook portlet's remote services (the Guestbook Mobile SDK). You'll then install this Mobile SDK and Screens into your iOS project. You'll also learn about the iOS app's design and implement authentication with Login Screenlet.

This section of the Learning Path covers these topics:

1. Setting up the Guestbook portlet
2. Building the Guestbook Mobile SDK
3. Creating the iOS project
4. Installing the Guestbook Mobile SDK and Liferay Screens in the iOS project
5. Designing Your App
6. Using Login Screenlet for Authentication
7. Creating the Guestbooks Scene

When you finish, you'll be ready to start developing your first Screenlet.

47.1 Setting up the Guestbook Portlet

Before you begin developing the Guestbook app for iOS, you must set up the Guestbook portlet in a Liferay DXP instance. To do this, follow these steps:

1. Install JDK 8
2. Install and Configure a Local Liferay DXP Bundle
3. Deploy the Guestbook Portlet to the Local Liferay DXP Instance

Installing the JDK

To get started, you must have JDK 8 installed. You can download and install the Java SE JDK from the Java downloads page. This page also has links to the JDK installation instructions.

Installing and Configuring a Local Liferay DXP Bundle

Follow these instructions to install and configure a local Liferay DXP instance:

1. Download a Liferay DXP Tomcat bundle from liferay.com.

2. Unzip the bundle to a new `bundles` folder. Note that the bundle's root folder is referred to as *Liferay Home* and is named according to the version, edition, and specific Liferay DXP release. For example, if you downloaded Liferay CE Portal 7.0 GA5 and unzipped it to a `bundles` folder on your machine, that bundle's Liferay Home folder is:

   ```
   bundles/liferay-ce-portal-7.0-ga5
   ```

3. Now you're ready to start Liferay DXP! Open a terminal and navigate to `[Liferay Home]/tomcat-[version]/bin`. Then run these commands:

   ```
   ./startup.sh
   tail -f ../logs/catalina.out
   ```

   The `tail` command ensures that the server log prints to the terminal.

4. After a minute or two, Liferay DXP starts up and automatically takes you to its initial setup page at http://localhost:8080. On this page, provide the following information to set up your Liferay DXP instance:

   - Enter a name for your instance.
   - Select the default language.
   - Uncheck the *Add Sample Data* box.
   - Enter the first name, last name, and email address of the default administrative user. For the purposes of this Learning Path, these don't have to be real.
   - If you want to connect Liferay DXP to a separate database such as MySQL or PostgreSQL, you can configure that connection here. Note that although the default embedded database is fine for development on your local machine, it isn't optimized for production.

   Click *Finish Configuration* when you're done.

5. Accept the terms of use.

6. Set a password and a password reminder query for your administrative user. Your Liferay DXP instance then takes you to its default site.

Great! Next, you'll deploy the Guestbook portlet to your Liferay DXP instance.

Deploying the Guestbook Portlet

Now that your portal is set up, you can deploy the Guestbook portlet to it. Follow these instructions to do so:

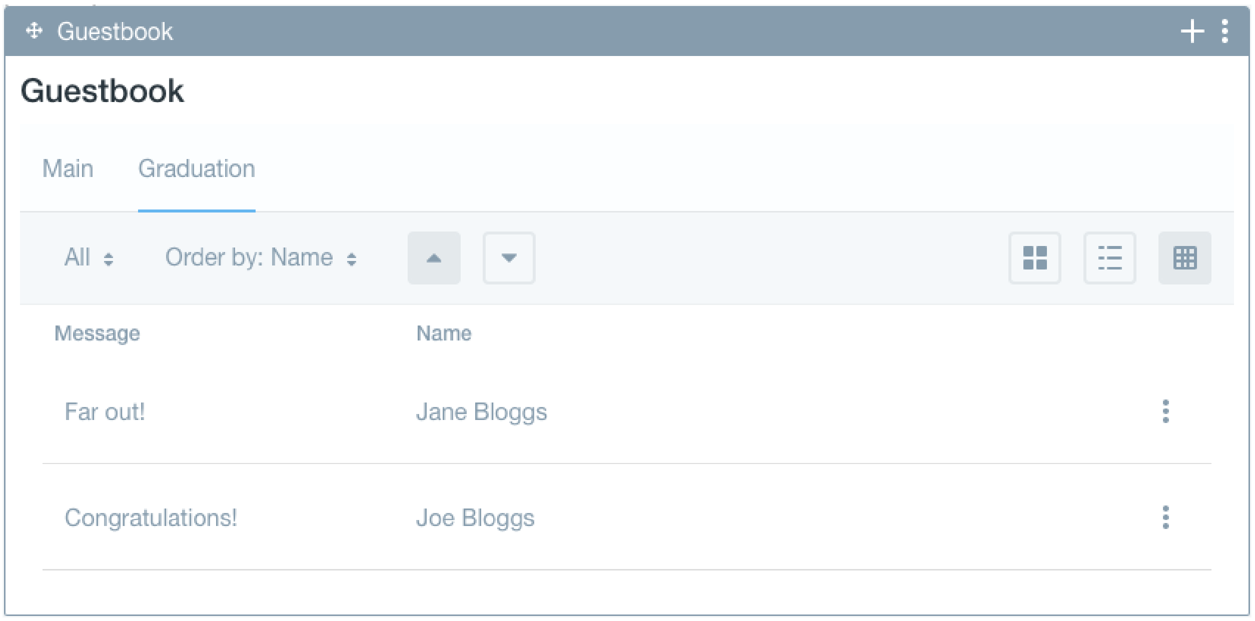
1. Download the Guestbook portlet's modules:

   - `com.liferay.docs.guestbook.api-1.0.0.jar`
   - `com.liferay.docs.guestbook.portlet-1.0.0.jar`
   - `com.liferay.docs.guestbook.service-1.0.0.jar`
   - `com.liferay.docs.guestbook.service-wsdd-1.0.0.jar`

2. Place these modules in your Liferay DXP instance's `[Liferay Home]/deploy` folder. You should then see console messages indicating that the modules have successfully deployed and started.

3. On your Liferay DXP instance's default site, click the *Add* button (+) on the upper-right corner of the screen. Then click the *Applications* → *Sample* category and drag *Guestbook* onto the page. The Guestbook portlet should now appear with the default guestbook (Main).

4. In the portlet, add a new guestbook and an entry or two each from the *Add* menu (+) that appears in the top right of the portlet's border when you mouse over the portlet. When you create the Guestbook iOS app, this ensures there's some content to display in it. The Guestbook portlet on your site should now look like this:



Message	Name
Far out!	Jane Bloggs
Congratulations!	Joe Bloggs

Figure 47.1: The Guestbook portlet, with a new guestbook and some entries.

Stupendous! You've successfully set up a Liferay DXP instance and added the Guestbook portlet to it. Now you're ready to get started with the Liferay Mobile SDK.

47.2 Building the Guestbook Mobile SDK

Once you've deployed the Guestbook portlet, you're ready to build the Guestbook Mobile SDK. You might be asking yourself, "Why do I have to build a separate Mobile SDK? Can't I just use the pre-built Mobile SDK that Liferay already provides?" Fantastic question! The reason is that Liferay's pre-built Mobile SDK doesn't have the classes it needs to call the Guestbook portlet's remote services. The pre-built Mobile SDK includes only the framework necessary to make server calls to the remote services of Liferay DXP's *core* apps. Core apps (also referred to as *out-of-the-box* apps) are those included with every Liferay DXP instance. Since you're calling services of an app the default Mobile SDK doesn't know about (the Guestbook portlet), you must build a Mobile SDK that can call its services. Now put on your hard hat, because it's time to get building!

Building the Mobile SDK

In the Mobile SDK source code, Liferay provides a Mobile SDK Builder that you can use to build your own Mobile SDKs. For the builder to generate the classes that can call a non-core app's remote services, those services must be available and accompanied by a Web Service Deployment Descriptor (WSDD). To learn how the Guestbook portlet's remote services and WSDD were generated, see the section Generating Web Services in the web application Learning Path. Since the Guestbook portlet's web services already exist, you don't need to generate them. Just remember that you must generate web services when developing your own portlets.

1. Download the Mobile SDK's source code and unzip the file to a location on your machine where you want the Mobile SDK to reside. This location is purely personal preference; the builder works the same no matter where you put the Mobile SDK's source code. Once unzipped, the Mobile SDK's source code is in the folder `liferay-mobile-sdk-ios-7.0.13`.

2. Now you're ready to build the Guestbook Mobile SDK. The builder contains a convenient command line wizard to assist you in building Mobile SDKs. To start it, navigate to the `liferay-mobile-sdk-ios-7.0.13` folder and run the following command:

   ```
   ./gradlew createModule
   ```

   The wizard launches and asks you to enter your project's properties. You'll do this next.

3. You must first provide the `Context` property. This is the context path of the remote services the builder will generate classes and methods for. To view your Liferay DXP instance's remote service context paths, go to http://localhost:8080/api/jsonws. On the page's upper left, there's a menu for selecting the context path. Select *gb*, which is the Guestbook portlet's context path. The UI updates to show only the remote services available in the selected context path. Return to the terminal and enter `gb` for the `Context` property.

4. Next, the wizard needs the `Package Name` property. Because the builder also builds an Android version of the Mobile SDK, this property defines the Java package path for the classes the builder generates. Accept the default value of `com.liferay.mobile.android`.

5. The wizard then asks for the `POM Description` property. This property also applies to the Android version of the Mobile SDK. Since the builder requires it, however, enter `Guestbook SDK`. The following screenshot shows these properties entered in the wizard:

   Once you enter the final property, the builder runs and generates a `BUILD SUCCESSFUL` message.

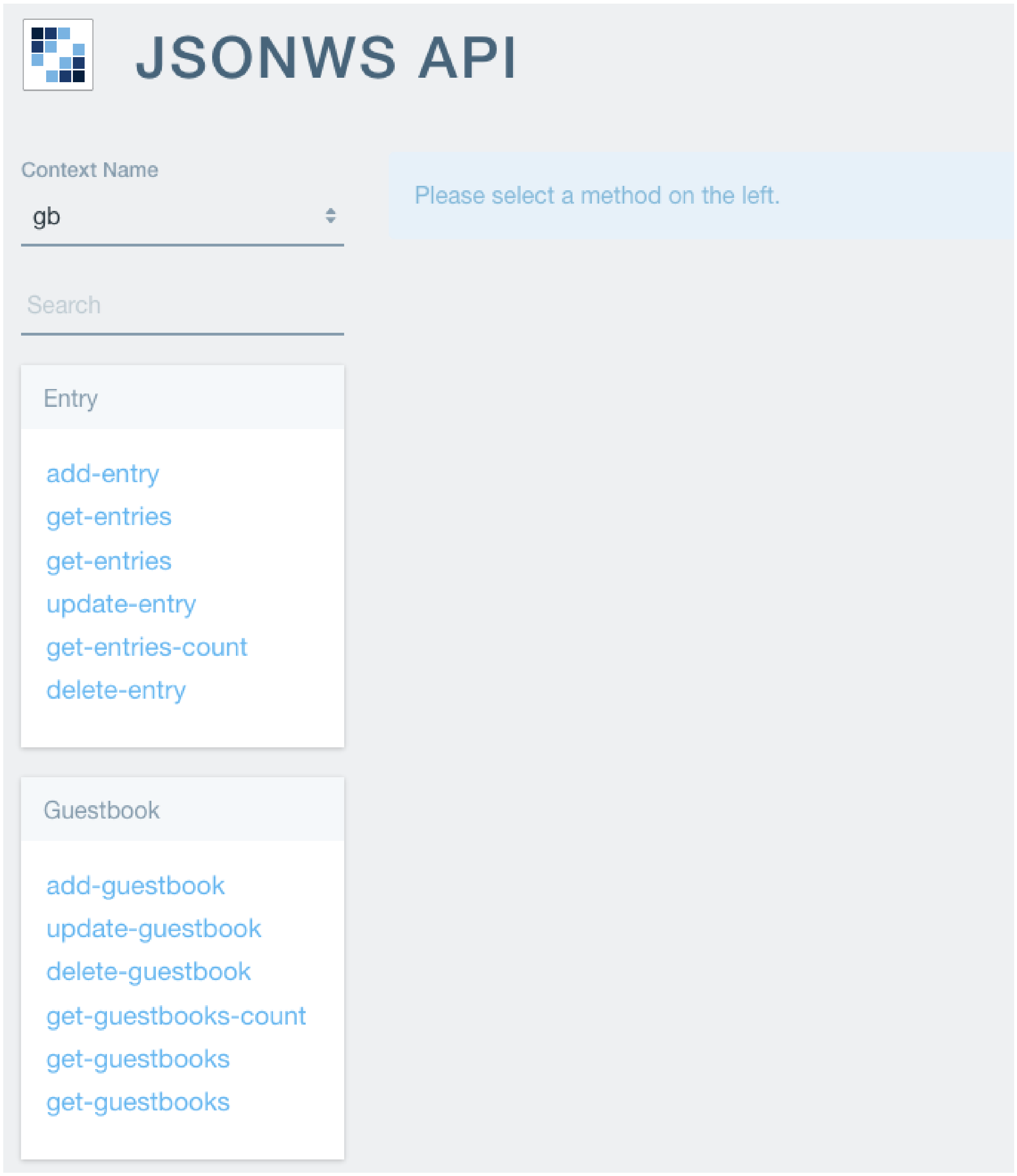
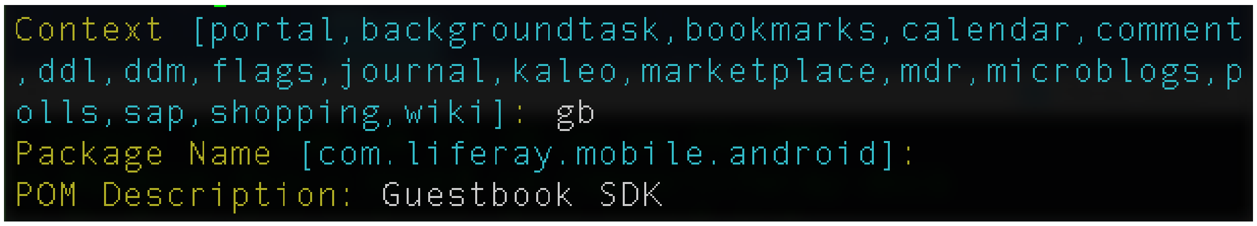


Figure 47.2: The Guestbook Portlet's context path (gb) on the server.



```
Context [portal,backgroundtask,bookmarks,calendar,comment
,ddl,ddm,flags,journal,kaleo,marketplace,mdr,microblogs,p
olls,sap,shopping,wiki]: gb
Package Name [com.liferay.mobile.android]:
POM Description: Guestbook SDK
```

Figure 47.3: To build your Mobile SDK, you must enter values for the Context, Package Name, and POM Description properties. The blue values in square brackets are defaults.

6. Now that the builder contains a gb module, you must generate that module's remote services. To do this, first navigate to this folder:

   ```
   liferay-mobile-sdk-ios-7.0.13/modules/gb
   ```

   Then run this command:

   ```
   ../../gradlew generate
   ```

   As before, the builder runs and generates a BUILD SUCCESSFUL message. Great! You're probably wondering what just happened, though. The builder generated the source classes you'll use in your iOS app to interact with the Guestbook portlet. You can find these source classes in the following folder of the Mobile SDK's source code:

   ```
   modules/gb/ios/Source/Service/v7
   ```

   The last folder in this path, v7, denotes the Liferay DXP version the classes work with. This folder has two subfolders that correspond to each entity in the Guestbook portlet: guestbook and entry. Each subfolder contains an Objective-C class header and implementation file for that entity's class (LRGuestbookService_v7 and LREntryService_v7, respectively).

7. There's one last thing to do before you can use these classes in your iOS app: put them in a JAR file. To do this, make sure you're still in the modules/gb folder on the command line and run this command:

   ```
   ../../gradlew zip
   ```

   This command generates this ZIP file:

   ```
   modules/gb/build/liferay-gb-ios-sdk-1.0.zip
   ```

   This ZIP file is the Guestbook Mobile SDK. It contains the service classes required to call the Guestbook portlet's remote services.

Congratulations! You just built the Guestbook Mobile SDK. Now that's an accomplishment worth writing in a guestbook. All you need now is an iOS app to install this Mobile SDK in. The next article shows you how to create it.

47.3 Creating the iOS Project

Now that you've built the Guestbook Mobile SDK, you're ready to create the Guestbook iOS app. This article walks you through the steps required to create the app's project in Xcode. After this, you'll be ready to install the Guestbook Mobile SDK and Liferay Screens.

Note: This learning path presumes that you've installed and know how to use Xcode. If you need assistance with Xcode, see Apple's developer site.

Follow these steps to create the Liferay Guestbook iOS project:

1. Open Xcode and click *Create a new Xcode project* on the welcome screen. Alternatively, you can select *File* → *New* → *Project*.
2. In the *Application* section of the *iOS* tab, select *Single View Application* and click *Next*.

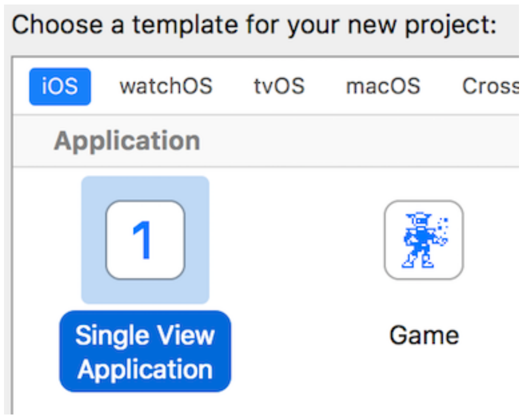


Figure 47.4: Use the Single View Application project template.

3. Enter the following fields as listed here:

    - **Product Name:** Liferay Guestbook
    - **Organization Name:** Your name
    - **Organization Identifier:** com.liferay.docs
    - **Language:** Swift

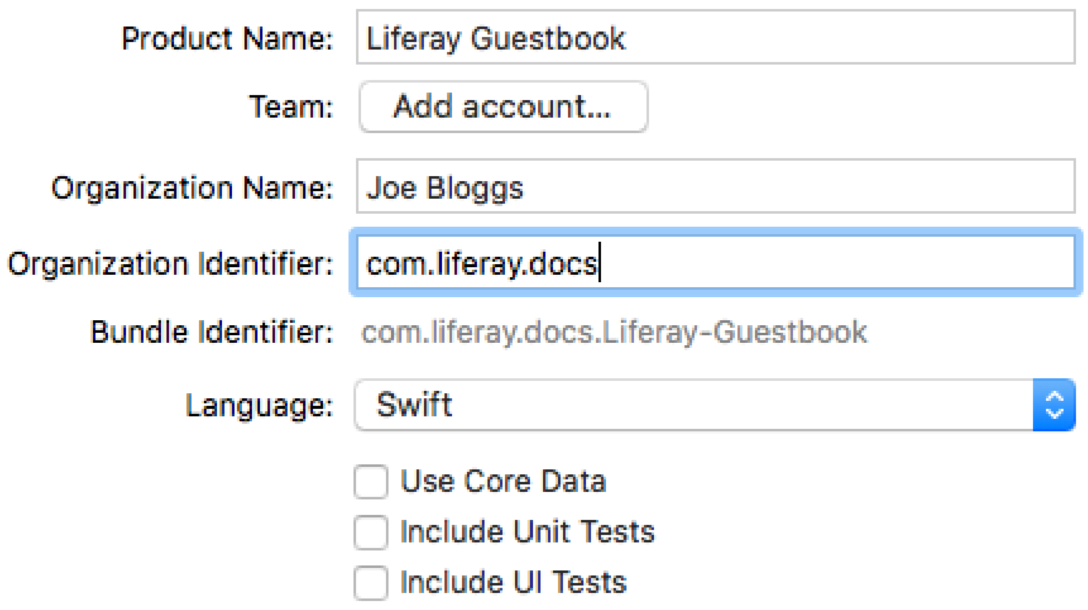
    Make sure the checkboxes below these fields are unchecked, and click *Next*.

4. Select a location for your project and click *Create*. Xcode creates and opens your new project.

Nice! You now have the iOS project in which you'll develop the Liferay Guestbook app. Before doing any development, however, you must install Liferay Screens and the Guestbook Mobile SDK. You'll do this next.

47.4 Installing Liferay Screens and the Guestbook Mobile SDK

For your iOS app to work with the Guestbook portlet, you must install the following in your iOS project:



Product Name:	Liferay Guestbook
Team:	Add account...
Organization Name:	Joe Bloggs
Organization Identifier:	com.liferay.docs
Bundle Identifier:	com.liferay.docs.Liferay-Guestbook
Language:	Swift
☐ Use Core Data	
☐ Include Unit Tests	
☐ Include UI Tests	

Figure 47.5: Fill out this form as shown.

- **Liferay's pre-built Mobile SDK:** This Mobile SDK contains the classes that call Liferay DXP's core remote services. It also contains the framework necessary for any Mobile SDK to make server calls.
- **The Guestbook Mobile SDK:** This Mobile SDK contains only the classes that call the Guestbook portlet's remote services.
- **Liferay Screens:** Screens contains the Screenlet framework and several built-in Screenlets like Login Screenlet. Because these built-in Screenlets work with Liferay DXP's core apps, they make their server calls with Liferay's pre-built Mobile SDK. Note that all Screenlets, including those that make server calls with a custom-built Mobile SDK, must use the framework in Liferay's pre-built Mobile SDK to issue their calls.

Since Liferay's pre-built Mobile SDK is a dependency of Liferay Screens, installing Screens automatically installs this Mobile SDK. You must, however, install the Guestbook Mobile SDK manually.

This article walks you through the installation of the Guestbook Mobile SDK and Liferay Screens. When you finish, you'll be ready to start developing the app.

Anatomy of the Liferay Guestbook iOS Project

Before getting started, you should learn a couple of terms this Learning Path uses when referring to the project's structure. Knowing these terms ensures that you know where to add folders and files in the project. In Xcode's Project navigator, there are two *Liferay Guestbook* items:

1. **The root project:** This is the first item in the Project navigator. It contains all other items in the project, and is labeled with a blue application document icon. The root project corresponds with a folder in

your file system that this Learning Path refers to as the *root project folder*. For example, the root project folder for the Liferay Guestbook project is `Liferay Guestbook`.

2. **The Liferay Guestbook folder:** This is immediately under the root project. It contains the app's files, and is labeled with a manila folder icon. Even though this folder shares a name with the root project folder on your file system, it **is not** the same thing. The root project folder contains this `Liferay Guestbook` folder.

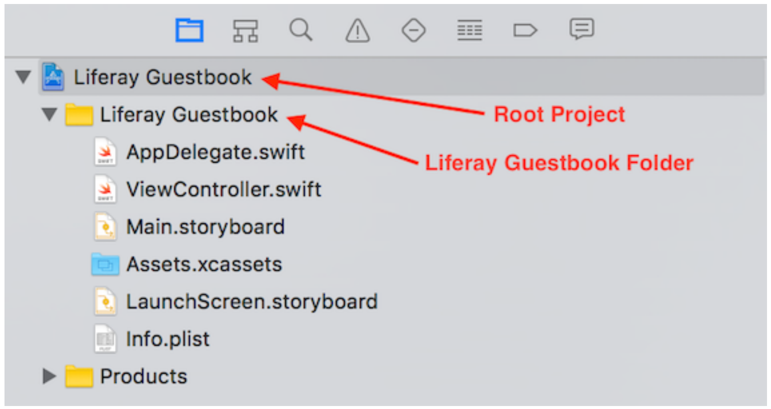


Figure 47.6: The root project and Liferay Guestbook folder are labeled in this screenshot.

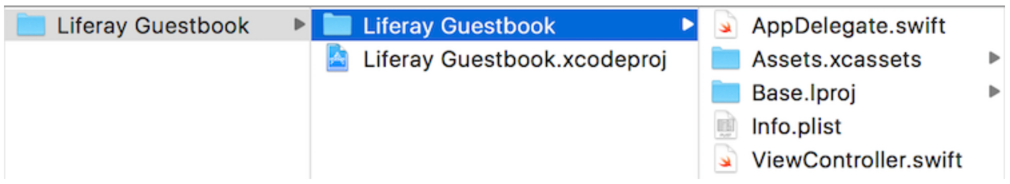


Figure 47.7: On your file system, the `Liferay Guestbook` root project folder contains the app's `Liferay Guestbook` folder. The latter is selected in this screenshot.

It's important not to confuse these two items. If you're ever confused about where things should go, click here to see the finished app in GitHub.

Now you're ready to install Liferay Screens!

Installing Liferay Screens

You'll use CocoaPods to install Liferay Screens. Click here for instructions on installing CocoaPods. After installing it, use these steps to install Screens:

1. In your root project's folder, create a file named `Podfile` that contains the following:

```
source 'https://github.com/CocoaPods/Specs.git'

platform :ios, '9.0'
use_frameworks!

target "Liferay Guestbook" do
    pod 'LiferayScreens', '3.0.2'
end

post_install do |installer|
  incompatiblePods = [
    'Cosmos',
    'CryptoSwift',
    'KeychainAccess',
    'Liferay-iOS-SDK',
    'Liferay-OAuth',
    'LiferayScreens',
    'Kingfisher'
  ]

  installer.pods_project.targets.each do |target|
    if incompatiblePods.include? target.name
      target.build_configurations.each do |config|
        config.build_settings['SWIFT_VERSION'] = '3.2'
      end
    end
    target.build_configurations.each do |config|
        config.build_settings['CONFIGURATION_BUILD_DIR'] = '$PODS_CONFIGURATION_BUILD_DIR'
    end
  end
end
```

This adds Liferay Screens 3.0.2 (the most recent version at the time this Learning Path was published) as a dependency. Since Screens 3.0.2 is incompatible with Swift 4, this `Podfile` also specifies that Screens and several of its dependencies (`incompatiblePods`) should be compiled by Swift 3.2. This lets you develop the app in Swift 4, while Screens itself is compiled in Swift 3.2.

Also note the setting for `CONFIGURATION_BUILD_DIR`. This is a workaround for a benign bug that causes Screenlet previews to fail in Interface Builder.

2. On the terminal, navigate to your root project's folder and run this command:

```
pod repo update
```

This ensures you have the latest version of the CocoaPods repository on your machine. Note that this command can take a while to run.

3. Still in your root project's folder in the terminal, run this command:

```
pod install
```

This installs the Liferay Screens as specified in your `Podfile`. Once this completes, quit Xcode and reopen your project by using the `LiferayGuestbook.xcworkspace` file in your root project's folder. From now on, you must use this file to open your project.

Great! You just installed Liferay Screens and the Liferay Mobile SDK! Next, you'll install the Guestbook Mobile SDK.

Installing the Guestbook Mobile SDK

To install the Guestbook Mobile SDK, you must add its service classes to your project. Recall that these service classes are Objective-C. To use them from your project's Swift code, you must also add and configure an Objective-C bridging header. You'll do these things now:

1. Recall that you created the following ZIP file containing the Guestbook Mobile SDK:

```
modules/gb/build/liferay-gb-ios-sdk-1.0.zip
```

Unzip this file to a location of your choosing on your machine. This creates the following directory hierarchy:

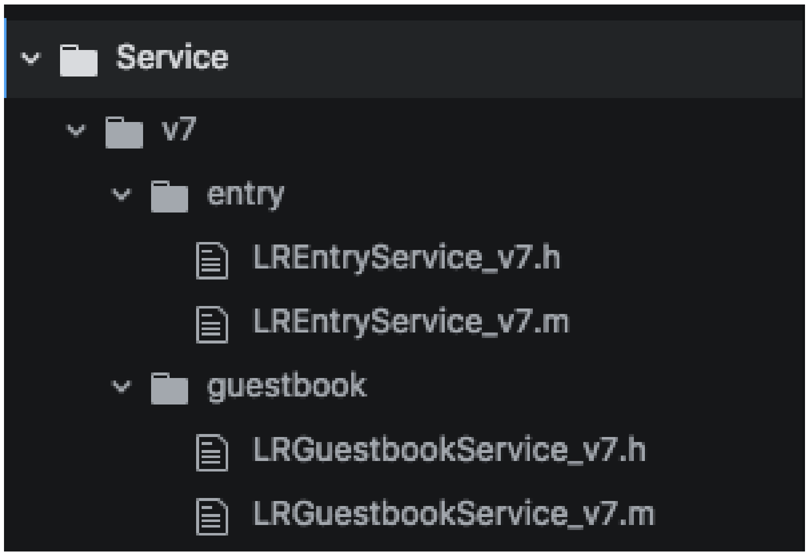

Service
v7
entry
LREntryService_v7.h
LREntryService_v7.m
guestbook
LRGuestbookService_v7.h
LRGuestbookService_v7.m

Figure 47.8: The Guestbook Mobile SDK's Objective-C classes unzip to this folder structure.

This should look familiar. It's the same `Service` folder, contents and all, from the Guestbook Mobile SDK you built earlier.

2. To install the service classes in your project, drag the v7 folder from your Finder into your Xcode project, directly under the root project. In the dialog that appears, make sure you select the following items, and then click *Finish*:

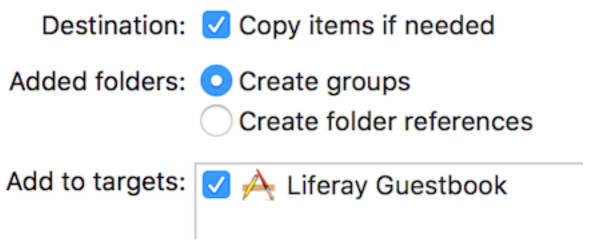

Destination: [✓ Copy items if needed]
Added folders: [✓ Create groups] Create folder references
Add to targets: [✓ Liferay Guestbook]

Figure 47.9: When adding the Guestbook Mobile SDK to your project, select these options and then click *Finish*.

3. The v7 folder and its contents are now inside your Xcode project. Now you must change each Objective-C class header file in the Guestbook Mobile SDK to always import the Liferay Mobile SDK framework. This is necessary because you used `use_frameworks!` in your `Podfile`.

   In `LREntryService_v7.h` and `LRGuestbookService_v7.h`, replace the `#if ... #endif` statement with `@import LRMobileSDK;`. Don't worry if Xcode doesn't recognize this import–you'll fix this shortly by adding and configuring an Objective-C bridging header in your project.

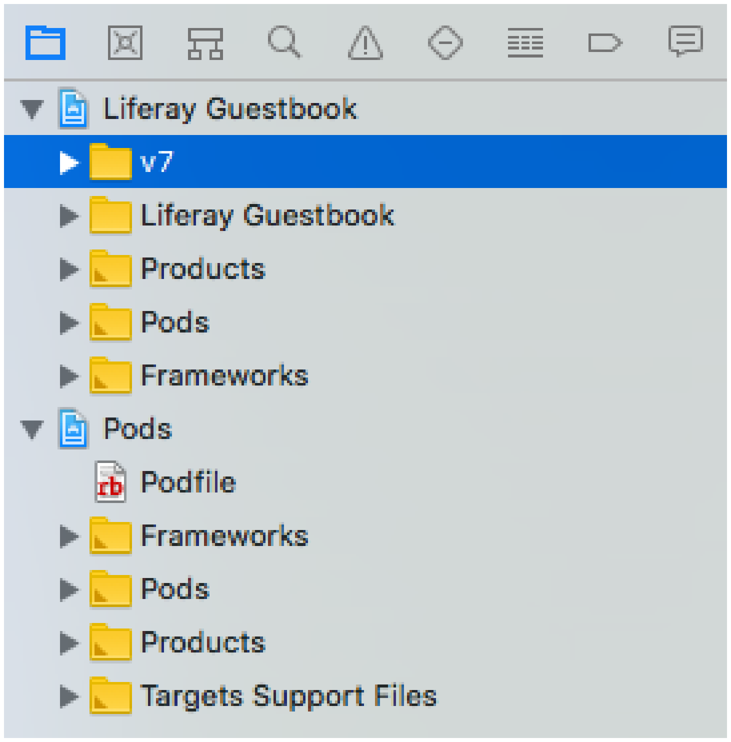


Figure 47.10: Your project structure should look like this after adding the Guestbook SDK.

4. In Xcode, for each `*.m` file in the Guestbook Mobile SDK (`LREntryService_v7.m` and `LRGuestbookService_v7.m`), make sure the checkbox for the *Liferay Guestbook* target is selected in the File inspector's *Target Membership* section.

To use the Guestbook Mobile SDK's Objective-C classes from Swift, you must add and configure an Objective-C bridging header in your project. Follow these instructions to do so:

1. In Xcode's project navigator, right-click the root project and select *New File*. In the window that appears, select *Header File* from the *Source* section of the *iOS* tab, and click *Next*.

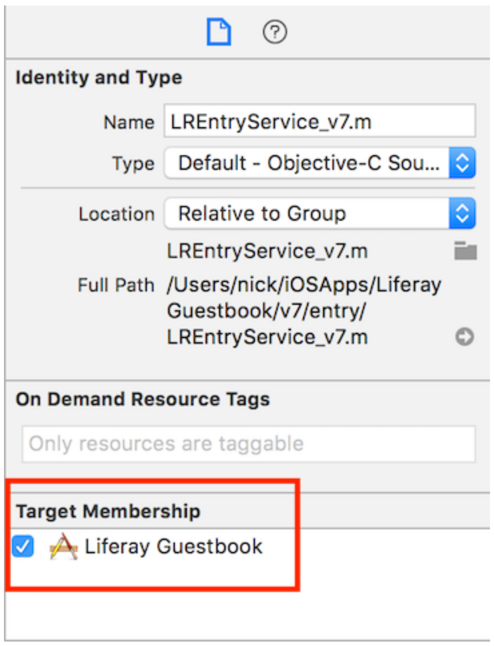
2. Name the file `Liferay Guestbook-Bridging-Header.h` and make sure that *Liferay Guestbook* with the blue icon is selected in the *Group* menu. To finish creating the file, uncheck any items in *Targets* and click *Create*.

3. Upon creating the header file, Xcode opens it in the editor. In this file, you must import the Guestbook Mobile SDK's header files. Add these imports immediately below the comments at the top of the file:

```
#import "LRGuestbookService_v7.h"
#import "LREntryService_v7.h"
```

Your bridging header file should now look like this:

```
#import "LRGuestbookService_v7.h"
#import "LREntryService_v7.h"

#ifndef Liferay_Guestbook_Bridging_Header_h
#define Liferay_Guestbook_Bridging_Header_h
```


Identity and Type

Name LREntryService_v7.m

Type Default - Objective-C Sou...

Location Relative to Group

LREntryService_v7.m

Full Path /Users/nick/iOSApps/Liferay Guestbook/v7/entry/LREntryService_v7.m

On Demand Resource Tags

Only resources are taggable

Target Membership

☑ Liferay Guestbook

Figure 47.11: Each `*.m` file in the Guestbook Mobile SDK must be part of the *Liferay Guestbook* target.

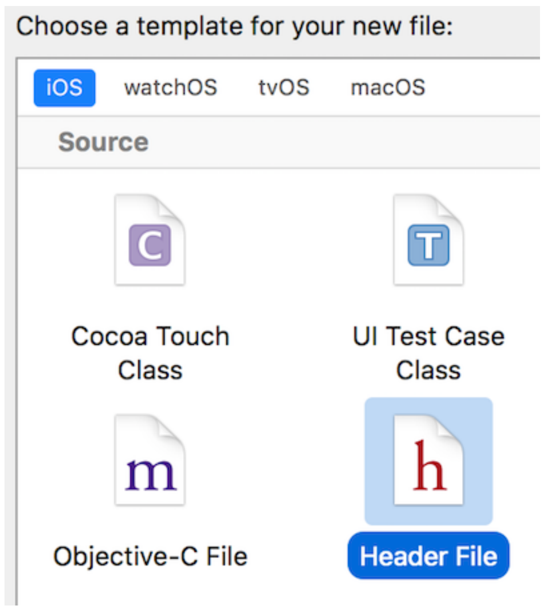

Choose a template for your new file:

Figure 47.12: Create a new iOS header file.

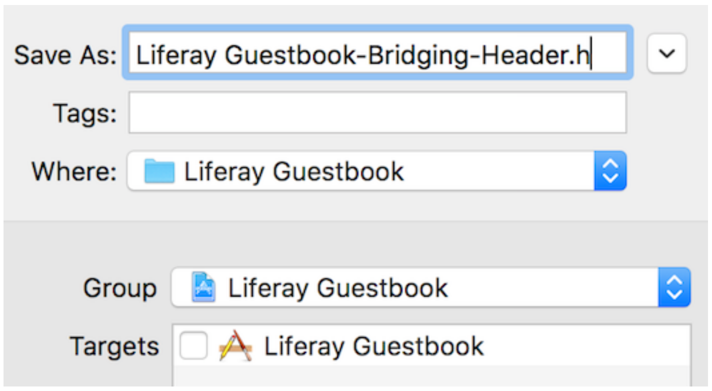
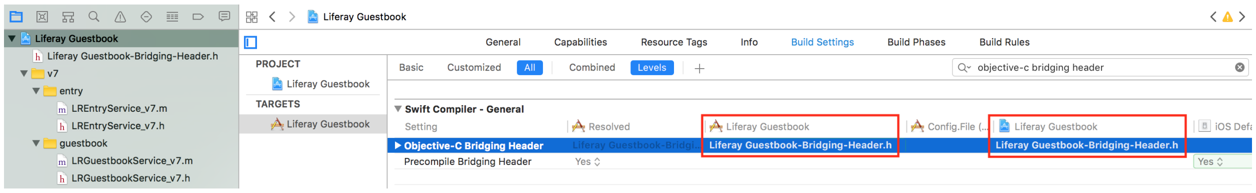

Save As: Liferay Guestbook-Bridging-Header.h
Tags:
Where: Liferay Guestbook
Group: Liferay Guestbook
Targets: Liferay Guestbook

Figure 47.13: Use these options to create the header file.

```
#endif
```

4. Now you must configure your project to use this file. Select the root project on the left and then click *Build Settings*. Enter for *Objective-C Bridging Header* in the search box. The matching build setting appears under the section *Swift Compiler - General*. In the two *Liferay Guestbook* fields for this build setting, enter the bridging header's file name.



Swift Compiler - General

Setting	Resolved	Liferay Guestbook	Config.File (...	Liferay Guestbook	iOS Defa
Objective-C Bridging Header	Liferay Guestbook-Bridgi...	Liferay Guestbook-Bridging-Header.h		Liferay Guestbook-Bridging-Header.h	
Precompile Bridging Header	Yes				Yes

Figure 47.14: The red boxes highlight the two Liferay Guestbook fields configured to use the bridging header file.

5. Build the project.

Awesome! You've successfully installed the Guestbook Mobile SDK. Next, you'll configure your app to communicate with your Liferay DXP installation.

Configuring Communication with Liferay Portal

For Liferay Screens to work with your app, you must configure it to communicate with your Liferay DXP installation. You'll do this by setting attributes in a `plist` file:

1. In Xcode's project navigator, right-click the *Liferay Guestbook* folder (**not** the root project) and select *New File*. In the dialog that appears, select the *iOS* tab then scroll down to the *Resource* section and select *Property List*. Click *Next*.

2. Name the file `liferay-server-context.plist` and make sure you're creating it in the *Liferay Guestbook* folder, which should be selected in the *Group* menu. Also make sure that *Liferay Guestbook* is selected in the *Targets* menu. Then click *Create*.

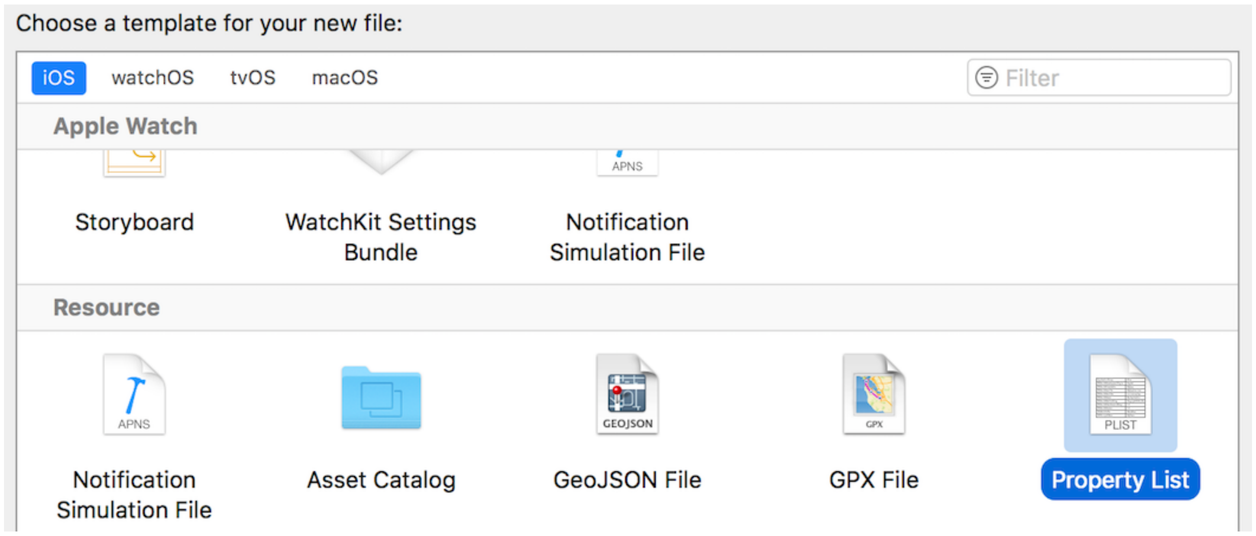


Figure 47.15: : Use the Property List template to create a new plist file.

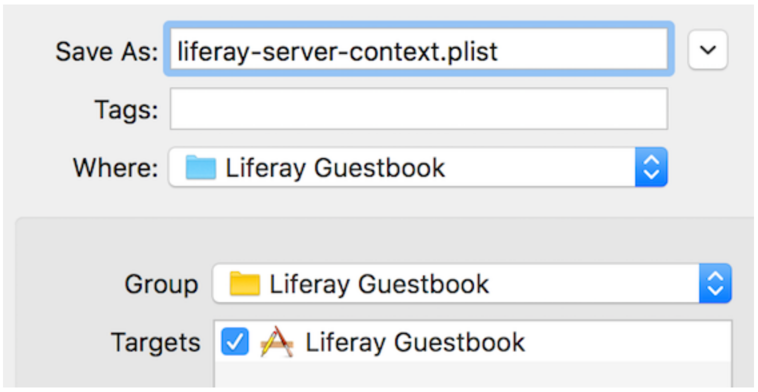

Save As: liferay-server-context.plist
Tags:
Where: Liferay Guestbook
Group Liferay Guestbook
Targets ✓ Liferay Guestbook

Figure 47.16: : Create the plist file as shown here.

3. The plist file now opens in the editor. Right-click the file in the Project navigator and select *Open As → Source Code*. Replace the file's contents with this code:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>server</key>
    <string>http://localhost:8080</string>
    <key>version</key>
    <integer>70</integer>
    <key>companyId</key>
    <integer>20116</integer>
    <key>groupId</key>
    <integer>20143</integer>
</dict>
</plist>
```

This plist file sets the server address (http://localhost:8080), Liferay DXP version (70 specifies Liferay CE Portal 7.0 and Liferay DXP 7.0), companyId (Liferay DXP instance ID), and groupId (site ID) the app retrieves data from.

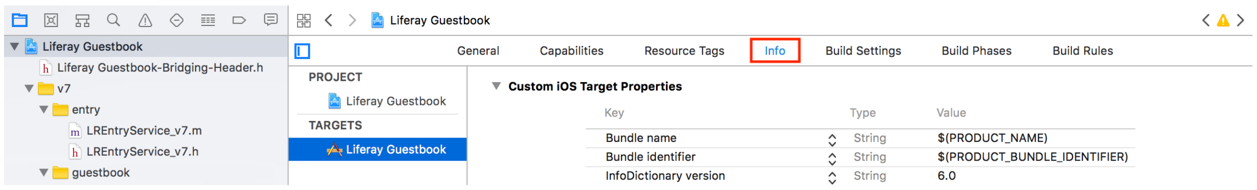
4. Change the `companyId` and `groupId` in the `plist` file to match those of your Liferay DXP instance. You can find your company ID in your portal at *Control Panel* → *Configuration* → *Virtual Instances*. The instance's ID is in the *Instance ID* column. You can find your site ID from the site you put the Guestbook portlet on. Navigate to this site, and in the *Site Administration* menu select *Configuration* → *Site Settings*. The site ID is listed at the top of the *General* tab.

Next, you'll configure iOS App Transport Security.

Disabling App Transport Security

App Transport Security is an iOS security feature that restricts all network activity to HTTPS. It isn't necessary for use in development and testing. Since your local Liferay DXP instance uses HTTP by default, App Transport Security prevents your app from communicating with the portal. You must therefore disable it:

1. Select your project in Xcode's Project navigator. With the *Liferay Guestbook* target selected in the outline, click the *Info* tab.



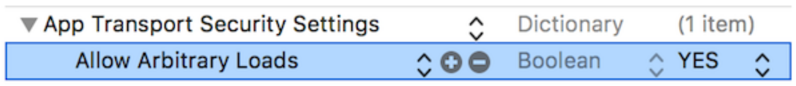
Custom iOS Target Properties

Key	Type	Value
Bundle name	String	$(PRODUCT_NAME)
Bundle identifier	String	$(PRODUCT_BUNDLE_IDENTIFIER)
InfoDictionary version	String	6.0

Figure 47.17: : You'll disable App Transport Security in the Info tab.

2. In *Custom iOS Target Properties*, right-click *Bundle OS Type Code* and select *Add Row*. In the new row's text field, enter *App Transport Security Settings*.

3. Even though it doesn't yet contain any items, ensure that the App Transport Security Settings category is open (click the triangle icon to its left to open/close it). Now click the + icon next to App Transport Security Settings and select *Allow Arbitrary Loads*. Then select *YES* for this field's value.



App Transport Security Settings	Dictionary	(1 item)
Allow Arbitrary Loads	Boolean	YES

Figure 47.18: : Your App Transport Security settings should look like this.

The Allow Arbitrary Loads setting should look like this when viewing your app's `Info.plist` file as code:

```
...
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
...
```

Stupendous! You've successfully installed Liferay Screens and the Guestbook Mobile SDK, and configured your app to communicate with your Liferay DXP instance. Before starting work on the app, however, you should learn the app's design. The next article walks you through this.

47.5 Designing Your App

As a developer, you know that developing any kind of app without an overall design goal and plan to implement it is a recipe for disaster. To avoid this, you need to decide some things upfront. The Liferay Guestbook app needs a straightforward way to do three things:

1. Authenticate users
2. Display guestbooks
3. Display entries

To authenticate users, all you need to do is insert and configure Login Screenlet in your app. Login Screenlet comes complete with its own UI. The design for authentication, therefore, like with Liferay DXP itself, is done for you.

You must, however, create the UI for displaying guestbooks and entries. What sort of UI would be best for this? Although the *best* UI for any purpose is a matter of opinion, displaying guestbooks and entries in lists is a good choice. Lists are common, compact design elements familiar to mobile users. Since most mobile devices don't have room to display a list of guestbooks and a list of entries at the same time, you also need a user-friendly way to display and manage these lists. It makes sense to show a list of guestbooks automatically after the user authenticates. The user can then select a guestbook to view a list of its entries. Users are familiar with this drill-down navigation style, as it's common throughout iOS.

To display these lists, you'll create your own Screenlets: Guestbook List Screenlet and Entry List Screenlet. Guestbook List Screenlet needs to retrieve guestbooks from the portlet and display them in a simple list. Once written, using this Screenlet is a simple matter of inserting it in a scene. Entry List Screenlet needs to retrieve and display a guestbook's entries in a similar list. You'll display the entries by inserting Entry List Screenlet in another scene.

Also note that these Screenlets are *list Screenlets*. You develop list Screenlets by using the list Screenlet framework, which sits on top of the core Screenlet framework. The list Screenlet framework makes it easy for developers to write Screenlets that display lists of entities from a Liferay DXP instance.

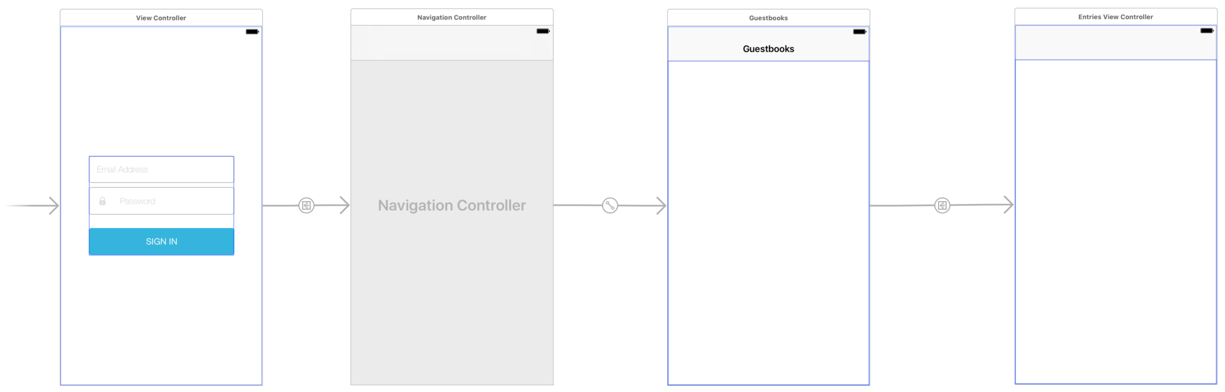


Figure 47.19: After login, the user transitions to the guestbooks scene where Guestbook List Screenlet displays a list of guestbooks. Upon selecting a guestbook, the entries scene displays a list of that guestbook's entries with Entry List Screenlet. Because the guestbooks and entries scenes are embedded in a navigation controller, the user can navigate back to the guestbooks scene via a back button in the navigation bar.

Awesome! Now you have a basic UI design and know the Screenlets you'll create to implement it. But the app currently contains only one empty scene, which you'll use for authentication with Login Screenlet. To use your custom list Screenlets, you'll need to create two additional scenes. You'll do this as you develop the app.

Great! Now you understand the Liferay Guestbook app's design. Next, you'll use Login Screenlet to implement authentication.

47.6 Using Login Screenlet for Authentication

For the app to retrieve data from the Guestbook portlet, the user must first authenticate to the Liferay DXP instance. You can implement authentication using the Liferay Mobile SDK, but it takes time to write. Using Liferay Screens to authenticate takes about ten minutes. In this article, you'll use Login Screenlet to implement authentication in your app.

Adding Login Screenlet to the App

To use any Screenlet, you must follow two steps:

1. Insert the Screenlet in the storyboard scene where you want it to appear. You do this by adding an empty view to the scene, and then setting the Screenlet class as the view's custom class.
2. Conform the scene's view controller's class to the Screenlet's delegate protocol. This lets the view controller respond to the Screenlet's events.

In this app, you'll use Login Screenlet in the app's first (and at this time, only) scene. After adding the Screenlet to this scene, you'll conform `ViewController` (the scene's view controller class) to the `LoginScreenletDelegate` protocol.

Adding Login Screenlet to the Scene

Follow these steps to add Login Screenlet to the scene:

1. In `Main.storyboard`, first select the scene's view controller. Then drag and drop a plain view (`UIView`) from the Object Library onto the view controller. In the outline view, this new view should be nested under the view controller's existing view.

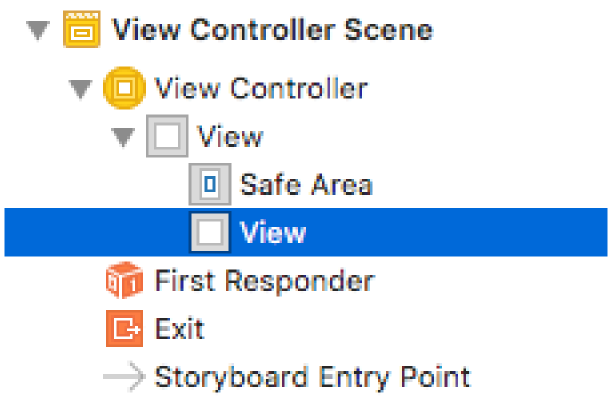
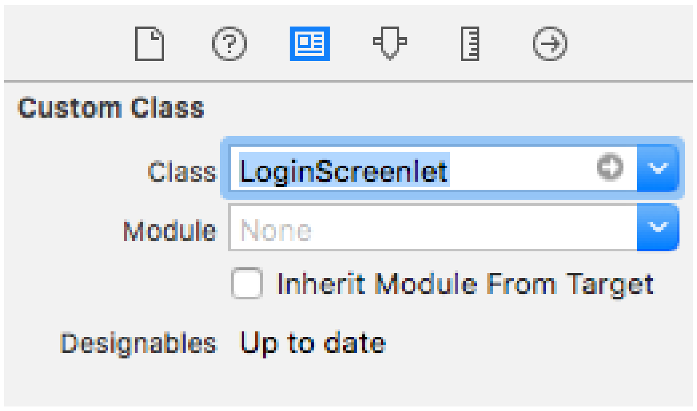


Figure 47.20: The new view is nested under the view controller's existing view.

2. With the new view selected, open the Identity inspector and set the view's Custom Class as `LoginScreenlet`. Xcode now builds the project and renders Login Screenlet's preview in the view. Also note that the view now appears as Login Screenlet in the outline view.


Custom Class

Class LoginScreenlet

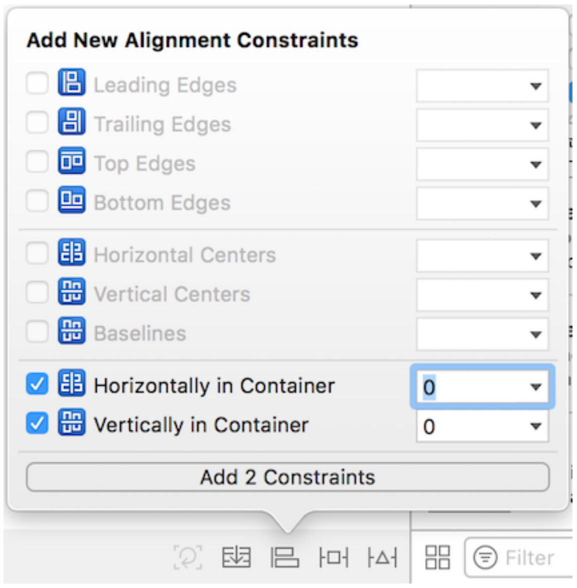
Module None

☐ Inherit Module From Target

Designables Up to date

Figure 47.21: You must set the view's Custom Class to `LoginScreenlet`.

3. Now you'll set the constraints to center Login Screenlet in the scene. Although this isn't required (you can technically position Login Screenlet anywhere you want), centering an authentication UI is common in mobile apps. Center Login Screenlet in the scene, and click the *Align* menu at the bottom-right of the canvas. In this menu, check the checkboxes for *Horizontally in Container* and *Vertically in Container*, and click the *Add 2 Constraints* button (don't worry about the Auto Layout errors that appear–you'll resolve these in the next step).

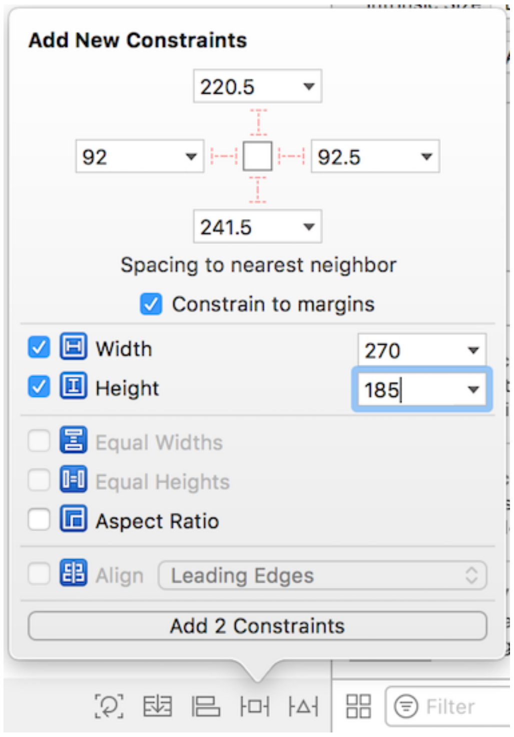

Add New Alignment Constraints

☐ Leading Edges	
☐ Trailing Edges	
☐ Top Edges	
☐ Bottom Edges	
☐ Horizontal Centers	
☐ Vertical Centers	
☐ Baselines	
✓ Horizontally in Container	0
✓ Vertically in Container	0

Add 2 Constraints

Figure 47.22: These alignment constraints center Login Screenlet in the scene.

4. By default, Login Screenlet stretches or compresses to fill the view. It's compressed at the moment because of the alignment constraints. To avoid any ill-effects caused by automatic resizing, you'll set the Screenlet to a fixed size. With the view selected, open the *Add New Constraints* menu at the bottom-right of the canvas. In this menu, set the *Width* to 270 and the *Height* to 185, and click the *Add 2 Constraints* button. The Screenlet looks better now and the Auto Layout errors are gone. Note that you don't have to use these exact width and height values when using Login Screenlet. You can size the Screenlet however you wish.



Add New Constraints

220.5

92

92.5

241.5

Spacing to nearest neighbor

✓ Constrain to margins

✓ Width 270

✓ Height 185

Equal Widths

Equal Heights

Aspect Ratio

Align Leading Edges

Add 2 Constraints

Figure 47.23: Setting these size constraints ensures that Login Screenlet isn't too stretched out or compressed.

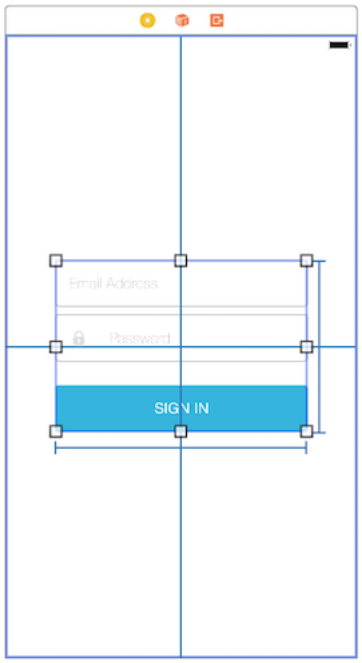


Figure 47.24: With the alignment and size constraints set, Login Screenlet appears in the center of the scene and its UI components aren't too compressed or stretched out.

Nicely done! The scene now contains Login Screenlet. Next, you'll conform `ViewController` (the scene's view controller class) to the `LoginScreenletDelegate` protocol.

Conforming to the Screenlet's Delegate Protocol

A view controller can respond to a Screenlet's events by conforming to the Screenlet's delegate protocol. This lets the app developer choose how their app behaves with the Screenlet. To respond to Login Screenlet's events, `ViewController` must conform to the `LoginScreenletDelegate` protocol. The Login Screenlet events that need a response are login success and failure. The delegate defines methods for both.

Follow these steps to conform `ViewController` to the `LoginScreenletDelegate` protocol:

1. Import `LiferayScreens` and set `ViewController` to adopt the `LoginScreenletDelegate` protocol. The first few lines of the class should look like this:

```
import UIKit
import LiferayScreens

class ViewController: UIViewController, LoginScreenletDelegate {...
```

2. Implement the `LoginScreenletDelegate` method `screenlet(_:onLoginResponseUserAttributes:)`. This method is called when authentication with Login Screenlet succeeds. Right now, you don't need to do anything in this method besides indicate that login succeeded:

```
func screenlet(_ screenlet: BaseScreenlet,
    onLoginResponseUserAttributes attributes: [String:AnyObject]) {
        print("Login Successful!")
}
```

3. Implement the `LoginScreenletDelegate` method `screenlet(_:onLoginError:)`. This method is called when authentication with Login Screenlet fails. All you need to do in this method is print a message indicating that login failed:

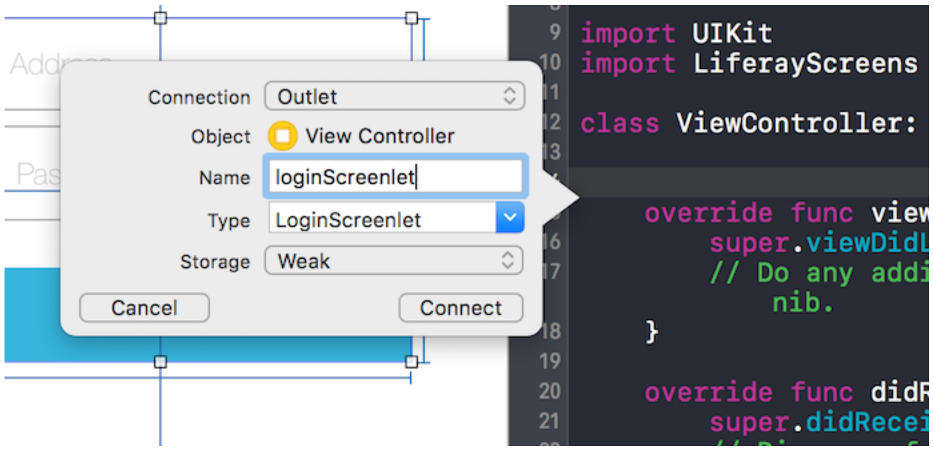
```
func screenlet(_ screenlet: BaseScreenlet, onLoginError error: NSError) {
    print("Login Failed!")
}
```

4. Now you must get a Login Screenlet reference in `ViewController`. You'll do this by creating an outlet to the Screenlet. Return to your storyboard and enter the Assistant editor to display `ViewController`'s code and the storyboard side by side. With Login Screenlet selected in the storyboard, Control-drag from the Screenlet to the `ViewController` class to create the outlet. In the dialog that appears upon releasing your mouse button, enter the following information and click *Connect*:

    - **Connection:** Outlet
    - **Name:** loginScreenlet
    - **Type:** LoginScreenlet
    - **Storage:** Weak

   Xcode then adds the following code inside the `ViewController` class:

```
@IBOutlet weak var loginScreenlet: LoginScreenlet!
```


Connection: Outlet
Object: View Controller
Name: loginScreenlet
Type: LoginScreenlet
Storage: Weak

```
import UIKit
import LiferayScreens

class ViewController:

    override func view
        super.viewDidL
        // Do any addi
            nib.
    }

    override func didR
        super.didRecei
```

Figure 47.25: Create an outlet from Login Screenlet to the `ViewController` class.

5. In the `ViewController` class, use the new `loginScreenlet` variable to set the view controller as the Screenlet's delegate. Do this in the `viewDidLoad()` method by deleting the placeholder comment and inserting this code below the call to `super.viewDidLoad()`:

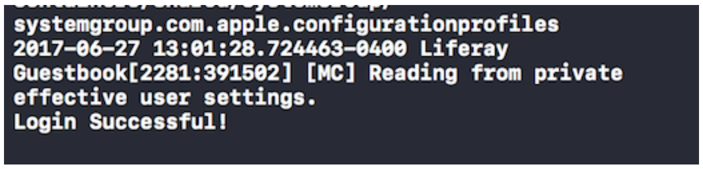
   ```
   self.loginScreenlet?.delegate = self
   ```

Great, you're finished! Before running the app, make sure that your `ViewController` class looks like this:

```
import UIKit
import LiferayScreens

class ViewController: UIViewController, LoginScreenletDelegate {

    @IBOutlet weak var loginScreenlet: LoginScreenlet!

    override func viewDidLoad() {
        super.viewDidLoad()
        self.loginScreenlet?.delegate = self
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    func screenlet(_ screenlet: BaseScreenlet, onLoginResponseUserAttributes attributes: [String:AnyObject]) {
        print("Login Successful!")
    }

    func screenlet(_ screenlet: BaseScreenlet, onLoginError error: NSError) {
        print("Login Failed!")
    }

}
```

Now you're ready to test your work. With your Liferay DXP instance running, launch the app using the iPhone simulator (any iPhone version supported by Xcode will work). Enter your credentials when Login Screenlet appears and click *SIGN IN*. In Xcode, the `Login Successful!` message appears in the console.

Nice job! Nothing else happens, though. Your app still displays Login Screenlet. This is expected. There aren't any other scenes for your app to navigate to. In the next article, you'll create the scene for displaying guestbooks: the guestbooks scene.



```
systemgroup.com.apple.configurationprofiles
2017-06-27 13:01:28.724463-0400 Liferay
Guestbook[2281:391502] [MC] Reading from private
effective user settings.
Login Successful!
```

Figure 47.26: It worked!

47.7 Creating the Guestbooks Scene

In the previous article, you used Login Screenlet to implement authentication in the login scene. Now you must create the scene you want the user to see following login: the guestbooks scene. Later in this Learning Path, you'll display guestbooks in this scene via Guestbook List Screenlet. For now, though, all you need to do is create the scene and take the user to it after login. This article shows you how to do this with the following steps:

1. Add a view controller to your storyboard. You'll then embed the view controller in a navigation controller. The navigation controller gives the scene a navigation bar and automatically implements back-navigation in the entries scene you'll create later.
2. Create the guestbook scene's view controller class. This class controls the view controller's behavior.
3. Create a segue from the login scene to the guestbooks scene. Upon login, this segue takes the user to the guestbooks scene.

First, you'll add a view controller to the storyboard.

Adding a View Controller to the Storyboard

Follow these steps to add a view controller to the storyboard:

1. Open your storyboard and drag and drop a *View Controller* from the Object Library to the right of the login scene.
2. Now you must embed this view controller in a navigation controller. Navigation controllers in iOS implement a *navigation stack*. You can loosely think of a navigation stack as a deck of cards where each card is a view controller with a navigation bar. The navigation bar contains a back button that lets you navigate to the previous view controller in the stack. You can also change the navigation bar's title to reflect the scene's content.

   With the new view controller selected in the storyboard, select *Editor → Embed In → Navigation Controller*. Your storyboard now shows the navigation controller with a segue to the guestbooks scene. Also, the guestbooks scene now contains an empty navigation bar.
3. Select the Navigation Item in the guestbooks scene's navigation bar. In the Attributes inspector, enter *Guestbooks* in the *Title* field, then enter a single space in the *Back Button* field and press return. This labels the scene via the navigation bar, and ensures that the back button in the Navigation controller has no label. The back button's default left chevron indicates the button's purpose without the need for additional text.

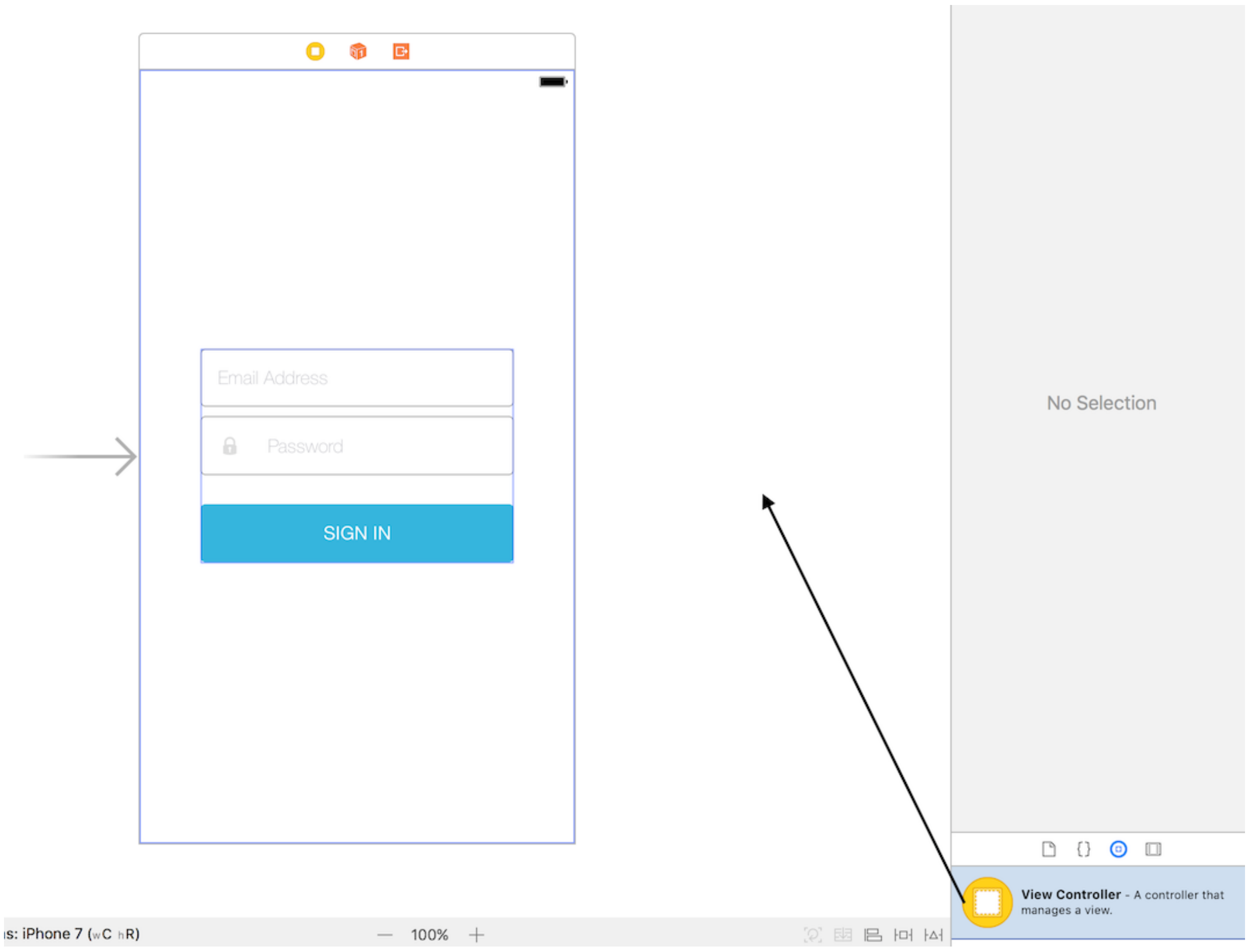
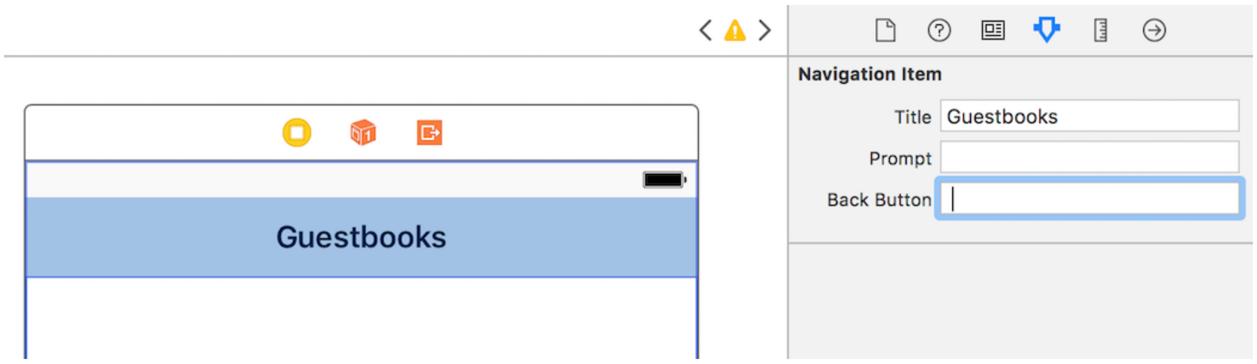
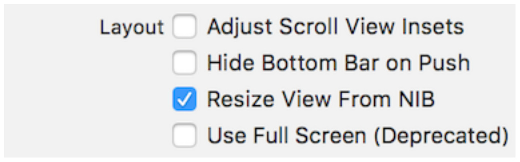


Figure 47.27: The arrow shows where to drag and drop the View Controller to create the new guestbooks scene.



Title: Guestbooks
Prompt:
Back Button:

Figure 47.28: Label the scene in the navigation bar, and set the back button's label to an empty space.


Layout

☐ Adjust Scroll View Insets

☐ Hide Bottom Bar on Push

☑ Resize View From NIB

☐ Use Full Screen (Deprecated)

Figure 47.29: Make sure the view controller's *Adjust Scroll View Insets* option is unchecked in the Attributes inspector.

4. With the guestbooks scene's view controller selected in the storyboard, open the Attributes inspector and uncheck *Adjust Scroll View Insets*. This ensures that the scene's contents are flush with the navigation bar.

Great! You now have the guestbooks scene, embedded in a navigation controller. For this scene's view controller to work, it must have a class that controls its behavior. You'll create this class next.

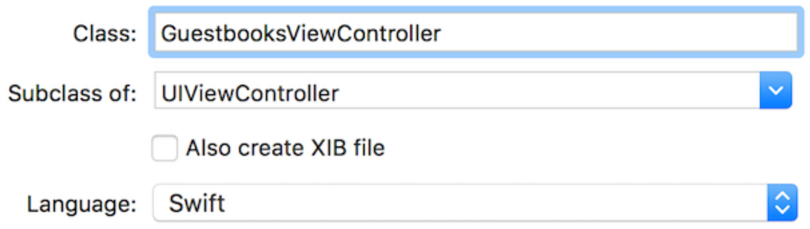
Creating the Guestbooks Scene's View Controller Class

Each view controller must have a class that controls its behavior. In this section, you'll create this class for the guestbooks scene's view controller. In the storyboard, you'll then set this class as the view controller's custom class.

1. Right-click the *Liferay Guestbook* folder in Xcode's project navigator and select *New File*. In the *iOS → Source* section of the dialog that appears, select *Cocoa Touch Class* and click *Next*.

2. The next screen in the dialog lets you set the class's name, subclass, and language. You can also choose whether to create an XIB file for the class. Enter the following information and click *Next*:

   - **Class:** `GuestbooksViewController`
   - **Subclass of:** `UIViewController`
   - **Also create XIB file:** Unchecked
   - **Language:** Swift


Class: GuestbooksViewController

Subclass of: UIViewController

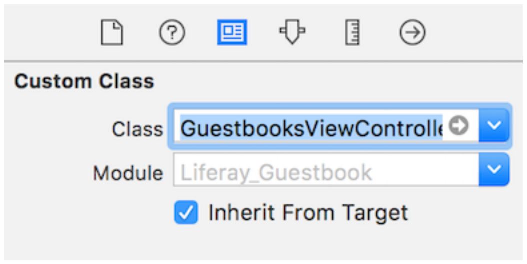
☐ Also create XIB file

Language: Swift

Figure 47.30: Set these options when creating the view controller's class.

3. The final screen in the dialog lets you set the class's location, group, and targets. Make sure *Liferay Guestbook* is selected for both the *Group* and *Targets* menus, and click *Create*.

4. In the storyboard, select the guestbooks scene's view controller. In the Identity inspector, set `GuestbooksViewController` as the Custom Class.


Custom Class

Class: GuestbooksViewControlle

Module: Liferay_Guestbook

☑ Inherit From Target

Figure 47.31: Set `GuestbooksViewController` as the custom class of the guestbooks scene's view controller.

Nice! The guestbooks scene's view controller now has a class that governs its behavior. You may have noticed a big problem, though. There's no way for the user to get from the login scene to the guestbooks scene. This is because there's no segue from the login scene to the navigation controller the guestbooks scene is embedded in. You'll fix this next.

Creating the Segue

Follow these steps to create and trigger the segue:

1. Control-drag from the login scene's view controller to the navigation controller. In the dialog that appears when you release your mouse button, select *show* for the segue type. The segue now connects the login scene's view controller and the navigation controller.

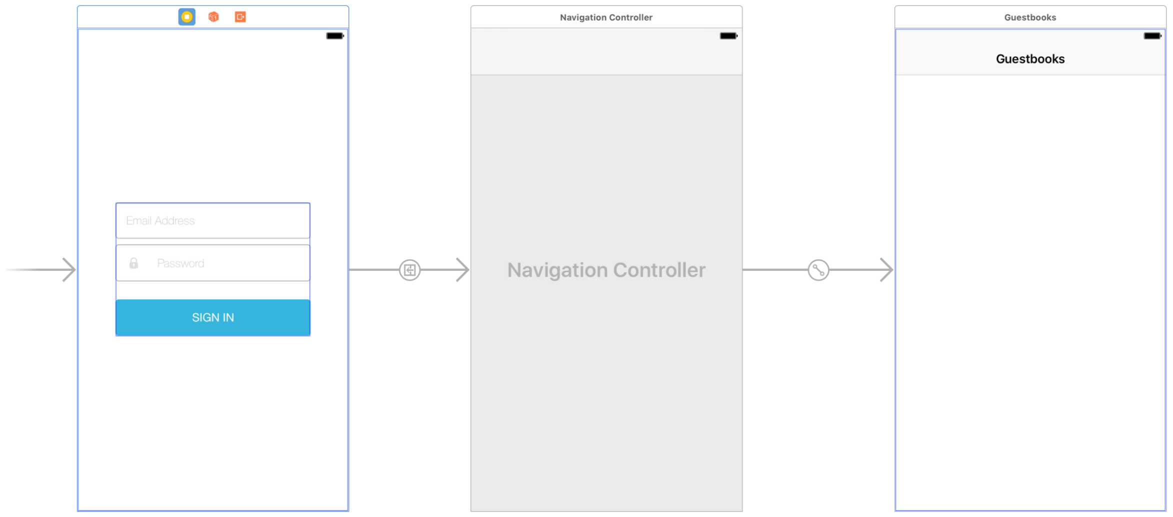
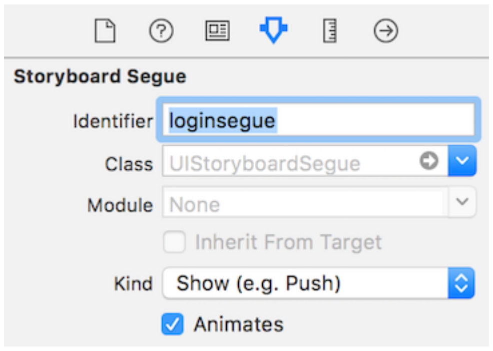


Figure 47.32: A segue now exists from the login scene to the navigation controller.

2. Now you must tell the login scene's view controller when to perform this segue. You'll do this programmatically in the `ViewController` class. To perform a segue programmatically, you must first give it an identifier in your storyboard. You'll then use this identifier in `ViewController` to perform the segue when a user logs in.

   In your storyboard, select the segue and then enter the Attributes inspector. Enter *loginsegue* in the *Identifier* field, and press *return*.


Storyboard Segue

Identifier loginsegue

Class UIStoryboardSegue

Module None

Inherit From Target

Kind Show (e.g. Push)

Animates

Figure 47.33: Set the segue's ID in the Attributes inspector.

3. Recall that the ViewController class's screenlet(_:onLoginResponseUserAttributes:) method is called upon successful login. You'll therefore trigger the segue in this method. Currently, this method only prints a success message. Below the line that prints this message, add the following code:

```
performSegue(withIdentifier: "loginsegue", sender: nil)
```

The performSegue(withIdentifier:sender:) method performs the segue with the specified identifier and includes any additional sender code. You send nil here since you don't need to send any information with the segue. Your screenlet(_:onLoginResponseUserAttributes:) method should now look like this:

```
func screenlet(_ screenlet: BaseScreenlet,
    onLoginResponseUserAttributes attributes: [String:AnyObject]) {
        print("Login Successful!")
        performSegue(withIdentifier: "loginsegue", sender: nil)
}
```

Great! Your app can now navigate to the guestbooks scene after login. To verify this, run the app and log in.

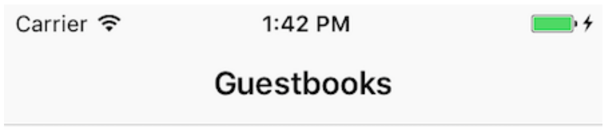


Figure 47.34: Following successful login, the app now navigates to the empty guestbooks scene.

Awesome! You've successfully added a scene for displaying guestbooks, and set the app to take the user there after login. Now you're ready to develop Guestbook List Screenlet. The next section in this Learning Path walks you through this.

Chapter 48

Creating Guestbook List Screenlet

In the previous section, you created an iOS app that contains the Guestbook Mobile SDK and Liferay Screens. You also used Login Screenlet to implement authentication to Liferay DXP. That's all your app does though. It doesn't display any Guestbook portlet content. In this section of the Learning Path, you'll create Guestbook List Screenlet to retrieve and display the portlet's guestbooks in your app's guestbooks scene.

Creating your own Screenlets brings additional benefits. Since you use a consistent, repeatable development model to create them, you can often reuse code when creating other Screenlets. You can also package and reuse Screenlets in other apps. What's more, Screenlet UIs are fully pluggable. This lets you change a Screenlet's appearance quickly without affecting its functionality. In summary, Screenlets are pretty much the greatest thing since sliced bread. Now it's time to make a sandwich.

You'll use these steps to create Guestbook List Screenlet:

1. Getting started: Create the Screenlet's folder, and the model class. The model class creates objects that represent guestbooks retrieved from the portlet, making it easier to work with guestbooks in your app.
2. Create the Screenlet's UI (its Theme).
3. Create the Connector. Connectors are Screenlet components that make server calls.
4. Create the Interactor. In list Screenlets, Interactors are Screenlet components that instantiate Connectors and receive their results.
5. Create the delegate. Delegates let other classes respond to the Screenlet's events.
6. Create the Screenlet class. The Screenlet class governs the Screenlet's behavior.

As background material, the following materials are helpful:

- Architecture of Liferay Screens for iOS: Explains the components that constitute a Screenlet, and how they relate to one another.
- Creating iOS Screenlets: Explains the general steps for creating a Screenlet.
- Creating iOS List Screenlets: Explains the general steps for creating a list Screenlet. This section of the Learning Path follows this tutorial.

Note that these tutorials explain Screenlet and list Screenlet concepts that this Learning Path doesn't cover in depth. Although it's possible to complete this Learning Path without reading these tutorials, they explain how Screenlets work in more detail. By reading them you'll be better able to apply the Learning Path material to your own Screenlets.

If you get confused or stuck while creating Guestbook List Screenlet, refer to the finished app that contains the Screenlet code here in GitHub.

48.1 Getting Started with Guestbook List Screenlet

Before creating a Screenlet, you should know how you'll use it. If you plan to use it in only one app, then you can create it in that app's project. If you need to use it in several apps, however, then it's best to create it in a separate project for redistribution. For information on creating Screenlets for redistribution, see the tutorial Packaging iOS Themes. Even though that tutorial is for packaging Themes, you can use the same steps to package Screenlets.

Since you'll use Guestbook List Screenlet in only this app, you can create it in a new folder inside the app's project. Create this folder now:

1. In the Finder, create the `GuestbookListScreenlet` folder inside the root project folder.

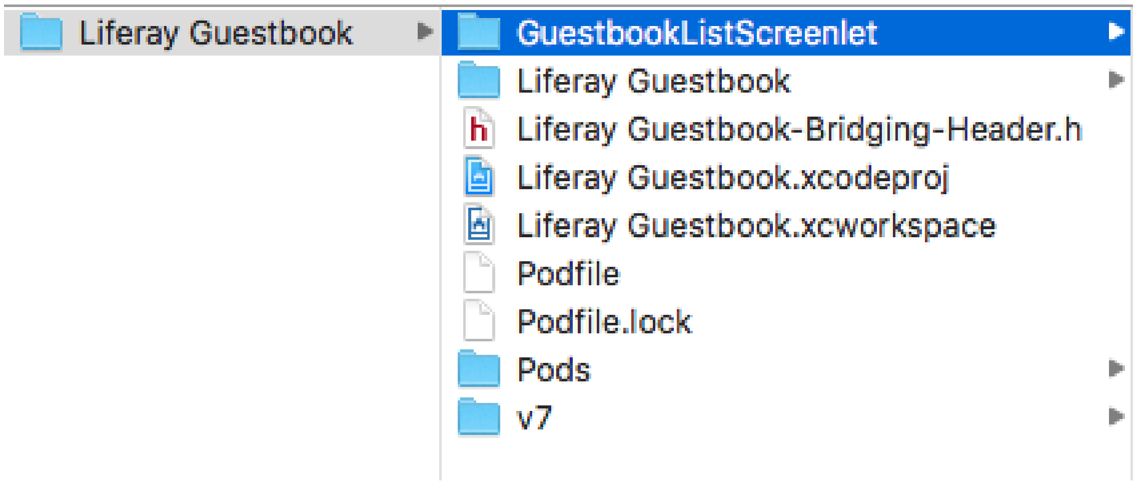

Liferay Guestbook
GuestbookListScreenlet
Liferay Guestbook
Liferay Guestbook-Bridging-Header.h
Liferay Guestbook.xcodeproj
Liferay Guestbook.xcworkspace
Podfile
Podfile.lock
Pods
v7

Figure 48.1: The new `GuestbookListScreenlet` folder should be inside your root project folder.

2. Drag and drop the `GuestbookListScreenlet` folder from the Finder into your Xcode project, under the root project. In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target. Then click *Finish*. The `GuestbookListScreenlet` folder now appears in your project.

Great! Now you have a folder to create Guestbook List Screenlet in. Before getting started, you should understand how pagination works in in list Screenlets.

Pagination

To ensure that users can scroll smoothly through large lists of items, list Screenlets support fluent pagination. Support for this is built into the list Screenlet framework. You'll see this as you construct your list Screenlet. For example, several methods have parameters for the start row and end row of a page in the list.

Now you're ready to begin creating the Screenlet!

Creating the Model Class

Liferay Screens typically receives entities from a Liferay DXP instance as a `[String:AnyObject]` dictionary, where `String` is the entity's attribute and `AnyObject` is the attribute's value. Although your Screenlet can

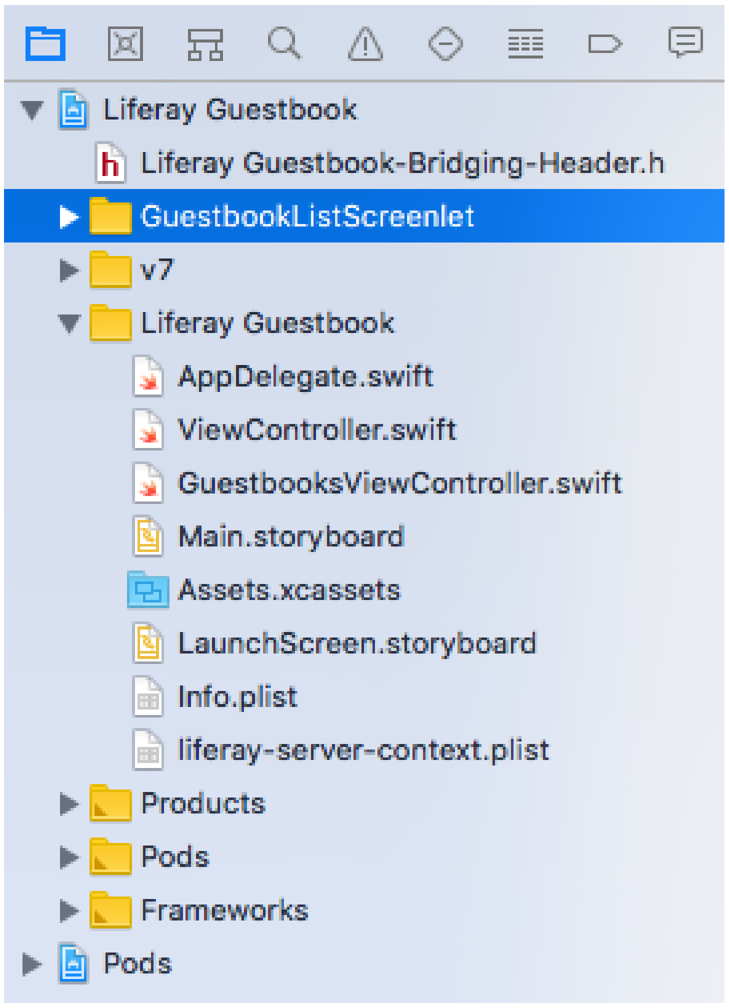


Figure 48.2: After adding the `GuestbookListScreenlet` folder, your project should look something like this.

use these dictionary objects, it's often easier to create a *model class* that converts each into an object that represents the corresponding entity in the portal. Model classes are especially convenient for complex entities composed of many attribute-value pairs, like guestbooks in the Guestbook portlet.

Your model class must contain all the code necessary to transform each `[String:AnyObject]` dictionary that comes back from the server into a model object that represents a guestbook. This includes a public constant for holding each `[String:AnyObject]` dictionary, a public initializer that sets this constant, and a public property for each attribute value.

The model class you'll create for Guestbook List Screenlet, `GuestbookModel`, creates `GuestbookModel` objects that represent guestbooks retrieved from the Guestbook portlet. You'll create this model class in a separate folder outside of the `GuestbookListScreenlet` folder. In this case, it makes sense to organize your code this way because other Screenlets may also use the model class. For example, if a Screenlet that edits guestbooks existed, it would also need `GuestbookModel` objects. Putting the model class in a separate folder makes it clear that this class doesn't belong exclusively to a single Screenlet.

Follow these steps to create Guestbook List Screenlet's model class:

1. In the Finder, create the `model` folder inside your root project folder.

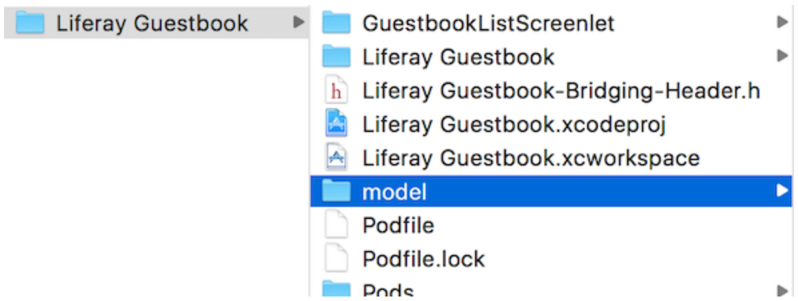

Liferay Guestbook
GuestbookListScreenlet
Liferay Guestbook
Liferay Guestbook-Bridging-Header.h
Liferay Guestbook.xcodeproj
Liferay Guestbook.xcworkspace
model
Podfile
Podfile.lock
Pods

Figure 48.3: The new `model` folder should be inside your root project folder.

2. Drag and drop the `model` folder from the Finder into your Xcode project, under the root project. In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target. Then click *Finish*. The `model` folder now appears in your project.

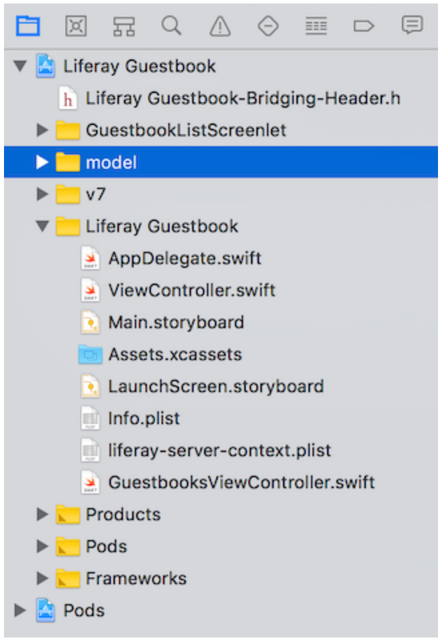


Figure 48.4: After adding the `model` folder, your project should look something like this.

3. In the Project navigator, right-click the `model` folder and select *New File*. In the dialog that appears, fill out each screen as follows:

   - Select *iOS* → *Source* → *Cocoa Touch Class*, and click *Next*.
   - Name the class `GuestbookModel`, set it to extend `NSObject`, select *Swift* for the language, and click *Next*.
   - Make sure the `model` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

   The new class now opens in the editor.

4. Replace the class file's contents with this code:

```
import UIKit

@objc public class GuestbookModel: NSObject {

    public let attributes: [String:AnyObject]

    public var guestbookId: Int64 {
        return attributes["guestbookId"]?.int64Value ?? 0
    }

    public var groupId: Int64 {
        return attributes["groupId"]?.int64Value ?? 0
    }

    public var companyId: Int64 {
        return attributes["companyId"]?.int64Value ?? 0
    }

    public var userId: Int64 {
        return attributes["userId"]?.int64Value ?? 0
    }

    public var userName: String {
        return attributes["userName"] as? String ?? ""
    }

    public var createDate: Int64 {
        return attributes["createDate"]?.int64Value ?? 0
    }

    public var modifiedDate: Int64 {
        return attributes["modifiedDate"]?.int64Value ?? 0
    }

    public var name: String {
        return attributes["name"] as? String ?? ""
    }

    //MARK: Initializer

    public init(attributes: [String:AnyObject]) {
        self.attributes = attributes
    }

}
```

This class creates `GuestbookModel` objects that represent guestbooks from the Guestbook portlet. The `[String:AnyObject]` dictionary contains the data of a guestbook retrieved from the portlet. The initializer sets this dictionary to the `attributes` property. Each computed property returns the value of a guestbook parameter in `attributes`. For example, the `guestbookId` property returns the value of the `guestbookId` parameter, the `groupId` property returns the value of the `groupId` parameter, and so on. To see how the Guestbook portlet defines these parameters, see the section on generating the portlet's back end in the Liferay Web Application Learning Path.

Also note that each computed property defaults to an empty string or `0`, depending on the property's type, if the parameter contains a value that can't be represented as that type. This prevents the app from crashing if the parameter doesn't have an appropriate value. For example, if the `guestbookId` parameter contains `nil`, then the `guestbookId` property returns `0`.

Great! Now you have a model class for guestbooks. Next, you'll create the Screenlet's UI.

48.2 Creating Guestbook List Screenlet's UI

In Liferay Screens for iOS, Screenlet UIs are called *Themes*. Every Screenlet must have at least one Theme. You'll use the following steps to create a Theme for Guestbook List Screenlet:

1. Create your Theme's folder and add it to your Xcode project.
2. Create an XIB file and use it to construct the UI.
3. Create your Theme's View class and set it as the XIB file's custom class.

Creating Your Theme's Folder

Even if you only plan on creating one Theme, it's best practice to create it in its own folder inside a parent `Themes` folder. The parent `Themes` folder gives you a place to put any additional Themes you create. You'll create a single Theme for Guestbook List Screenlet: the Default Theme. You'll therefore create the `Themes/Default` folder path inside the `GuestbookListScreenlet` folder.

Follow these steps to create your Theme's folder:

1. In the Finder, create the `Themes` folder inside your project's `GuestbookListScreenlet` folder. Then create the `Default` folder inside the new `Themes` folder.

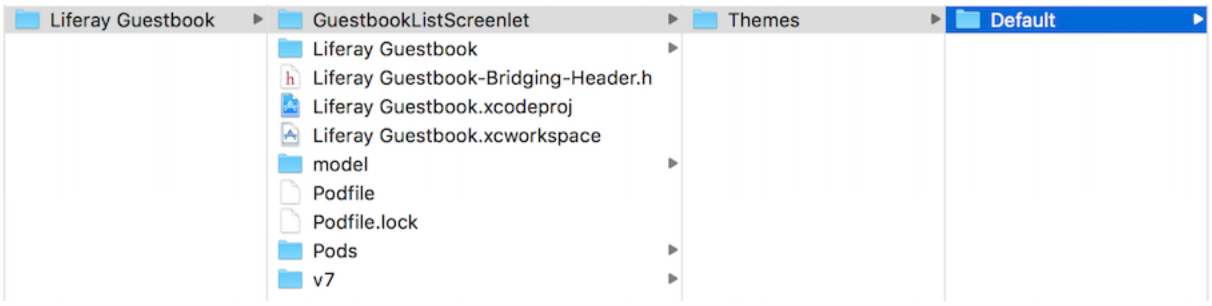


Figure 48.5: The new `Themes/Default` folder structure should be inside the Screenlet's folder.

2. Drag and drop the `Themes` folder from the Finder into your Xcode project, under the `GuestbookListScreenlet` folder. In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target. Then click *Finish*. The `Themes/Default` folder structure now appears in your project.

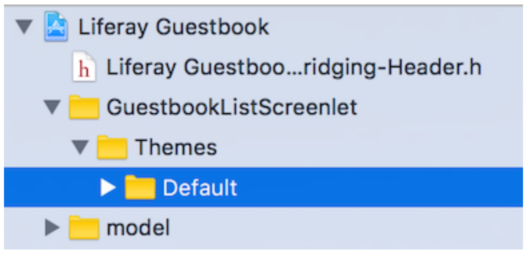


Figure 48.6: After adding the `Themes` folder to your project, the `Themes/Default` folder structure should appear in the Project navigator.

Now you're ready to start creating your Theme. First, you'll create its XIB file.

Creating the XIB File

A Theme requires an XIB file to define the UI's components and layout. Use these steps to create your Theme's XIB file:

1. In the Project navigator, right-click the `Default` folder and select *New File*. In the dialog that appears, select *iOS* → *User Interface* → *Empty*, and click *Next*. Name the file `GuestbookListView_default.xib`, and ensure that *Default* is selected for the save location and group. The *Liferay Guestbook* target should also be selected. Click *Create*. The file should then open in Interface Builder.

2. In Interface Builder, drag and drop a View from the Object Library to the canvas. Then add a Table View to the View. Set the Table View to take up the entire View.

3. With the Table View selected, open the *Add New Constraints* menu at the bottom-right of the canvas. In this menu, set *Spacing to nearest neighbor* to 0 on all sides, select *Constrain to margins*, and then click the *Add 4 Constraints* button.

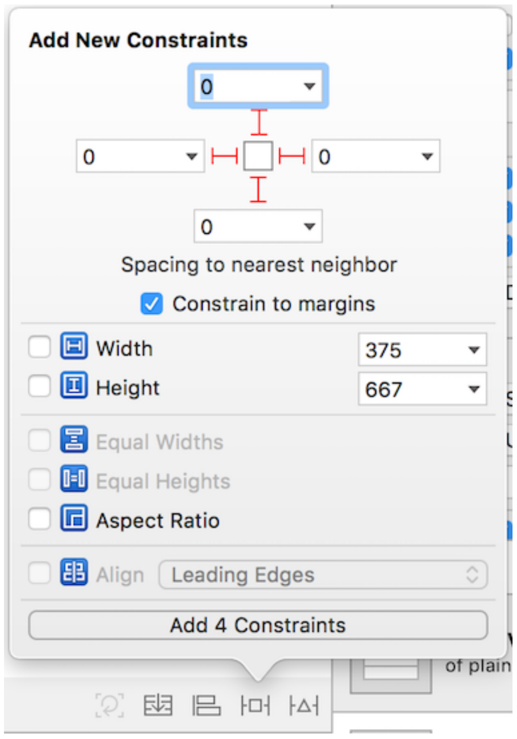


Figure 48.7: Add these constraints to the Table View in the XIB.

Your Theme's XIB is now finished. Next, you'll create your View class.

Creating the Theme's View Class

Every Theme needs a View class that controls its behavior. Since the XIB file uses a `UITableView` to show a list of guestbooks, your View class must extend the `BaseListTableView` class. Liferay Screens provides this class to serve as the base class for list Screenlets' View classes. Since `BaseListTableView` provides most of the required functionality, extending it lets you focus on the parts of your View class that are unique to your Screenlet.

Follow these steps to create your Screenlet's View class:

1. In the Project navigator, right-click the `Default` folder and select *New File*. In the dialog that appears, fill out each screen as follows:

    - Select *iOS → Source → Cocoa Touch Class*, and click *Next*.
    - Name the class `GuestbookListView_default`, set it to extend `BaseListTableView`, select *Swift* for the language, and click *Next*.
    - Make sure the `Default` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

2. In `GuestbookListView_default`, add an import for `LiferayScreens` and delete any placeholder comments in the class body.

3. Now you must override the View class methods that fill the table cells' contents. There are two methods for this, depending on the cell type:

    - **Normal cells:** the cells that show the entities. These cells typically use `UILabel`, `UIImage`, or another UI component to show the entity. Override the `doFillLoadedCell` method to fill these cells. Guestbook List Screenlet's View class must override `doFillLoadedCell` to set each cell's `textLabel` to a guestbook's name:

      ```
      override public func doFillLoadedCell(row: Int, cell: UITableViewCell,
          object: AnyObject) {

              let guestbook = object as! GuestbookModel
              cell.textLabel?.text = guestbook.name
      }
      ```

    - **Progress cell:** the cell at the bottom of the list that indicates the list is loading the next page of items. Override the `doFillInProgressCell` method to fill this cell. Guestbook List Screenlet's View class must override this method to set the cell's `textLabel` to the string `"Loading..."`:

      ```
      override public func doFillInProgressCell(row: Int, cell: UITableViewCell) {
          cell.textLabel?.text = "Loading..."
      }
      ```

    Your complete View class should look like this:

    ```
    import UIKit
    import LiferayScreens

    class GuestbookListView_default: BaseListTableView {

        override public func doFillLoadedCell(row: Int, cell: UITableViewCell,
            object: AnyObject) {

                let guestbook = object as! GuestbookModel
                cell.textLabel?.text = guestbook.name
        }

        override public func doFillInProgressCell(row: Int, cell: UITableViewCell) {
            cell.textLabel?.text = "Loading..."
        }

    }
    ```

4. Return to the Theme's XIB in Interface Builder, and set `GuestbookListView_default` as the the parent View's custom class. To do this, select the Table View's parent View, click the Identity inspector, and enter `GuestbookListView_default` as the Custom Class.

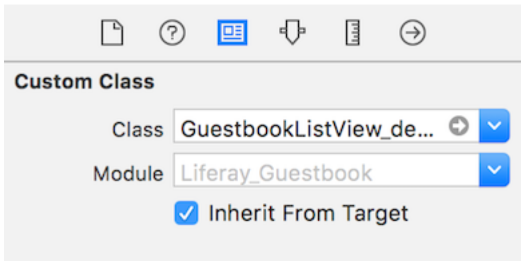

Custom Class
Class: GuestbookListView_de...
Module: Liferay_Guestbook
[✓] Inherit From Target

Figure 48.8: In the XIB file, set the Custom Class of the Table View's parent View to `GuestbookListView_default`.

5. With the Theme's XIB still open in Interface Builder, set the parent View's `tableView` outlet to the Table View. To do this, select the parent View and click the Connections inspector. In the Outlets section, drag and drop from the `tableView`'s circle icon (it turns into a plus icon on mouseover) to the Table View in the XIB. The new outlet then appears in the Connections inspector.

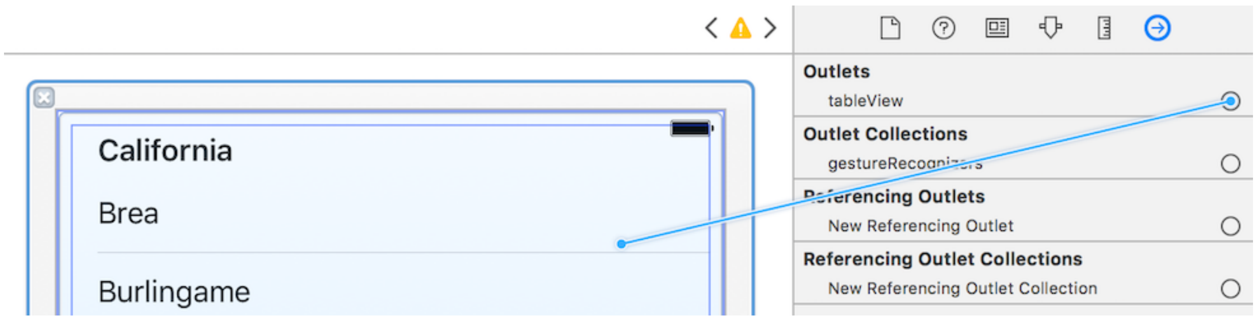


Figure 48.9: In the XIB, drag and drop from the `tableView` outlet to the Table View.

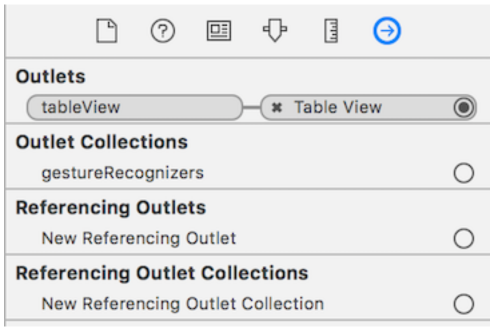

Outlets
tableView — Table View
Outlet Collections
gestureRecognizers
Referencing Outlets
New Referencing Outlet
Referencing Outlet Collections
New Referencing Outlet Collection

Figure 48.10: After creating the connection, it appears in the Connections inspector.

Great! Your Theme is finished. Next, you'll create Guestbook List Screenlet's Connector.

48.3 Creating Guestbook List Screenlet's Connector

Connectors are Screenlet components that make server calls. Non-list Screenlets don't require Connectors–they can make server calls in Interactors instead. Connectors, however, provide a layer of abstraction by making the server call outside the Interactor. This leaves the Interactor to instantiate the Connector and receive the server call's results. List Screenlets exploit this architectural advantage by requiring Connectors.

First, you'll create a folder for the Connector.

Creating Your Connector's Folder

Follow these steps to create your Connector's folder:

1. In the Finder, create the `Connector` folder inside your project's `GuestbookListScreenlet` folder.

2. Drag and drop the `Connector` folder from the Finder into your Xcode project, under the `GuestbookListScreenlet` folder. In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target. Then click *Finish*. The `Connector` folder now appears in your project.

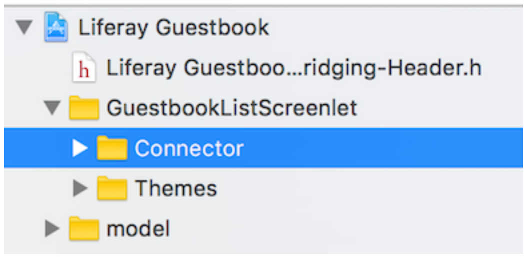


Figure 48.11: The new `Connector` folder should be inside the Screenlet's folder.

Now you're ready to create the Connector.

Creating the Connector

List Screenlet Connectors must extend the `PaginationLiferayConnector` class, which Liferay Screens provides to enable most of the functionality required by list Screenlet Connectors. Extending this class lets you focus on the functionality unique to your Connector. Your list Screenlet's Connector class must contain any properties it needs to make the server call, and an initializer that sets them. To support pagination, the initializer must also contain the following arguments, which you'll pass to the superclass initializer:

- `startRow`: The number representing the page's first row.
- `endRow`: The number representing the page's last row.
- `computeRowCount`: Whether to call the Connector's `doAddRowCountServiceCall` method (you'll learn about this method shortly).

Follow these steps to create Guestbook List Screenlet's Connector:

1. In the Project navigator, right-click the `Connector` folder and select *New File*. In the dialog that appears, fill out each screen as follows:

    - Select *iOS* → *Source* → *Cocoa Touch Class*, and click *Next*.

- Name the class `GuestbookListPageLiferayConnector`, set it to extend `PaginationLiferayConnector`, select *Swift* for the language, and click *Next*.
- Make sure the `Connector` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

2. In the new class, import `LiferayScreens` and add a public `Int64` constant called `groupId`. This constant holds the ID of the site the Connector retrieves guestbooks from. Your Connector should now look like this:

```
import UIKit
import LiferayScreens

class GuestbookListPageLiferayConnector: PaginationLiferayConnector {

    public let groupId: Int64

}
```

3. Create an initializer that takes the arguments `startRow`, `endRow`, `computeRowCount`, and `groupId`. In this initializer, set the `groupId` constant to the corresponding argument, then call the superclass initializer with the remaining arguments. Add the initializer as follows:

```
public init(startRow: Int, endRow: Int, computeRowCount: Bool, groupId: Int64) {
    self.groupId = groupId

    super.init(startRow: startRow, endRow: endRow, computeRowCount: computeRowCount)
}
```

4. Override the `doAddPageRowsServiceCall` method to make the server call that retrieves guestbooks from the portlet. This method must call the Guestbook SDK service method `getGuestbooksWithGroupId`, which retrieves guestbooks. To do this, you must first create a `LRGuestbookService_v7` instance from the session. Then call the service's `getGuestbooksWithGroupId` method with `groupId`, `startRow`, and `endRow`:

```
public override func doAddPageRowsServiceCall(session: LRBatchSession, startRow: Int, endRow: Int,
    obc: LRJSONObjectWrapper?) {
        let service = LRGuestbookService_v7(session: session)

        do {
            try service!.getGuestbooksWithGroupId(groupId, start: Int32(startRow), end: Int32(endRow))
        }
        catch {
            // ignore error: the service method returns nil because
            // the request is sent later, in batch
        }

}
```

Note that you don't need to do anything in the `catch` statement because the request is sent later, in batch. The session type `LRBatchSession` handles this for you. You'll receive the request's results elsewhere, once the request completes.

5. Override the `doAddRowCountServiceCall` method to make the server call that retrieves the total number of guestbooks from the portlet. This enables pagination. This method must call the Guestbook SDK service method `getGuestbooksCount`, which retrieves the total number of guestbooks. To do this, you must first create a `LRGuestbookService_v7` service instance from the session. Then call the service's `getGuestbooksCount` method with `groupId`:

```
override public func doAddRowCountServiceCall(session: LRBatchSession) {
    let service = LRGuestbookService_v7(session: session)

    do {
        try service!.getGuestbooksCount(withGroupId: groupId)
    }
    catch {
        // ignore error: the service method returns nil because
        // the request is sent later, in batch
    }
}
```

As in the previous step, you don't need to do anything in the `catch` statement.

Awesome! Your Connector is finished. Now you're ready to create the Interactor.

48.4 Creating Guestbook List Screenlet's Interactor

Interactors implement your Screenlet's actions. In non-list Screenlets, this can include making the server call. List Screenlets, however, make server calls via Connectors. Also, loading entities is usually the only action a user can take in a list Screenlet. Therefore, list Screenlet Interactors typically only need to instantiate the Connector and receive the server call's results. This is the case for Guestbook List Screenlet's Interactor. You'll create this Interactor now.

Creating Your Interactor's Folder

Follow these steps to create your Interactor's folder:

1. In the Finder, create the `Interactor` folder inside your project's `GuestbookListScreenlet` folder.

2. Drag and drop the `Interactor` folder from the Finder into your Xcode project, under the `GuestbookListScreenlet` folder. In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target. Then click *Finish*. The `Interactor` folder now appears in your project.

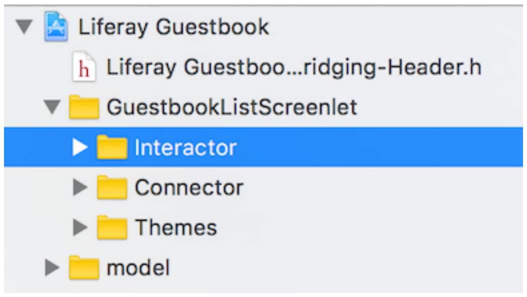


Figure 48.12: The new `Interactor` folder should be inside the Screenlet's folder.

Now you're ready to create the Interactor.

Creating the Interactor

The Interactor class of a list Screenlet that implements fluent pagination must extend Liferay Screens's `BaseListPageLoadInteractor` class. This class provides most of the functionality required by list Screenlet Interactors. Your Interactor class must also contain any properties your Screenlet needs and an initializer that sets them. This initializer needs arguments for the following properties, which it passes to the superclass initializer:

- `screenlet`: A `BaseListScreenlet` reference. This ensures the Interactor always has a Screenlet reference.
- `page`: The page number to retrieve.
- `computeRowCount`: Whether to call the Connector's `doAddRowCountServiceCall` method.

Follow these steps to create Guestbook List Screenlet's Interactor:

1. In the Project navigator, right-click the `Interactor` folder and select *New File*. In the dialog that appears, fill out each screen as follows:

    - Select *iOS* → *Source* → *Cocoa Touch Class*, and click *Next*.
    - Name the class `GuestbookListPageLoadInteractor`, set it to extend `BaseListPageLoadInteractor`, select *Swift* for the language, and click *Next*.
    - Make sure the `Interactor` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

2. In the new class, import `LiferayScreens` and add a private `Int64` constant called `groupId`. This constant holds the ID of the site guestbooks are retrieved from. Your Interactor should now look like this:

```
import UIKit
import LiferayScreens

class GuestbookListPageLoadInteractor: BaseListPageLoadInteractor {

    private let groupId: Int64

}
```

3. Create an initializer that takes the arguments `screenlet`, `page`, `computeRowCount`, and `groupId`. In this initializer, set the `groupId` constant to the corresponding argument, then call the superclass initializer with the remaining arguments. Note that if the the `groupId` is `0`, the `groupId` setting in `liferay-server-context.plist` is used instead. Add the initializer as follows:

```
init(screenlet: BaseListScreenlet,
    page: Int,
    computeRowCount: Bool,
    groupId: Int64) {

    self.groupId = (groupId != 0) ? groupId : LiferayServerContext.groupId

    super.init(screenlet: screenlet, page: page, computeRowCount: computeRowCount)
}
```

4. Override the `createListPageConnector` method to create and return an instance of your Connector, `GuestbookListPageLiferayConnector`. This method must first get a reference to the Screenlet via the `screenlet` property. When calling the Connector's initializer, use `screenlet.firstRowForPage` to convert the page number (`page`) to the page's start and end indices. You must also pass the initializer any other properties it needs, like `groupId`. Add this `createListPageConnector` method to your Interactor class:

```
public override func createListPageConnector() -> PaginationLiferayConnector {
    let screenlet = self.screenlet as! BaseListScreenlet

    return GuestbookListPageLiferayConnector(
        startRow: screenlet.firstRowForPage(self.page),
        endRow: screenlet.firstRowForPage(self.page + 1),
        computeRowCount: self.computeRowCount,
        groupId: groupId)
}
```

5. Override the `convertResult` method to convert each `[String:AnyObject]` result into a `GuestbookModel` object. The Screenlet calls this method once for each guestbook retrieved from the server. Add this method as follows:

```
override public func convertResult(_ serverResult: [String:AnyObject]) -> AnyObject {

    return GuestbookModel(attributes: serverResult)
}
```

6. Override the `cacheKey` method to return a key that can be used with offline mode. Although Guestbook List Screenlet won't initially support offline mode, this method is still required. For this Screenlet, the `groupId` serves as a sufficient key. Add this method as follows:

```
override public func cacheKey(_ op: PaginationLiferayConnector) -> String {
    return "\(groupId)"
}
```

Great! Your Interactor is finished. Next, you'll create the delegate.

48.5 Creating Guestbook List Screenlet's Delegate

A delegate is a Screenlet component that lets other classes to respond to a Screenlet's actions. For example, Login Screenlet's delegate lets the app developer implement methods that respond to login success or failure. Note that the reference documentation for each Screenlet that comes with Liferay Screens lists the Screenlet's delegate methods.

You can also create a delegate for your own Screenlet. Guestbook List Screenlet should have a delegate protocol that defines the following methods:

- `screenlet(_:onGuestbookListResponse:)`: Receives the `GuestbookModel` results when the server call succeeds. This lets app developers respond to a successful server call.
- `screenlet(_:onGuestbookListError:)`: Receives the `NSError` object when the server call fails. This lets app developers respond to a failed server call.
- `screenlet(_:onGuestbookSelected:)`: Receives the `GuestbookModel` when a user selects it in the list. This lets app developers respond to a guestbook selection by the user.

You'll create this delegate in the same file as the Screenlet class. Later, you'll finish creating the Screenlet class itself.

Follow these steps to create Guestbook List Screenlet's delegate:

1. In the Project navigator, right-click the `GuestbookListScreenlet` folder and select *New File*. In the dialog that appears, fill out each screen as follows:

- Select *iOS* → *Source* → *Cocoa Touch Class*, and click *Next*.
- Name the class `GuestbookListScreenlet`, set it to extend `BaseListScreenlet`, select *Swift* for the language, and click *Next*.
- Make sure the `GuestbookListScreenlet` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

2. In the new `GuestbookListScreenlet` class, import `LiferayScreens`, make the class `public`, and delete any placeholder comments in the class body.

3. In between the `import` statements and the class declaration, add the following code:

   ```
   @objc public protocol GuestbookListScreenletDelegate : BaseScreenletDelegate {

       @objc optional func screenlet(screenlet: GuestbookListScreenlet,
                               onGuestbookListResponse guestbooks: [GuestbookModel])

       @objc optional func screenlet(screenlet: GuestbookListScreenlet,
                               onGuestbookListError error: NSError)

       @objc optional func screenlet(screenlet: GuestbookListScreenlet,
                               onGuestbookSelected guestbook: GuestbookModel)

   }
   ```

   This defines the `GuestbookListScreenletDelegate` protocol, which extends the `BaseScreenletDelegate` protocol. Delegates for custom Screenlets, like Guestbook List Screenlet, must extend `BaseScreenletDelegate`. The rest of `GuestbookListScreenletDelegate` defines the delegate methods you'll use later to respond to the Screenlet's events.

The contents of `GuestbookListScreenlet.swift` should now look like this:

```
import UIKit
import LiferayScreens

@objc public protocol GuestbookListScreenletDelegate : BaseScreenletDelegate {

    @objc optional func screenlet(screenlet: GuestbookListScreenlet,
                            onGuestbookListResponse guestbooks: [GuestbookModel])

    @objc optional func screenlet(screenlet: GuestbookListScreenlet,
                            onGuestbookListError error: NSError)

    @objc optional func screenlet(screenlet: GuestbookListScreenlet,
                            onGuestbookSelected guestbook: GuestbookModel)

}

public class GuestbookListScreenlet: BaseListScreenlet {

}
```

Nice work! Now you're ready to complete the Screenlet class.

48.6 Creating the Screenlet Class

The Screenlet class is the main component that governs the Screenlet's behavior. When using a Screenlet, app developers primarily interact with its Screenlet class. A list Screenlet's Screenlet class must extend the

BaseListScreenlet class. Since BaseListScreenlet provides most of the functionality needed by Screenlet classes, extending it lets you focus on the functionality unique to your Screenlet. The Screenlet class must also define the configurable IBInspectable properties the Screenlet needs, create and return an instance of your Interactor, and respond to the Screenlet's events via the delegate.

The Screenlet class you created while creating the delegate is currently empty. You'll complete it now.

Follow these steps to complete the GuestbookListScreenlet class in GuestbookListScreenlet.swift:

1. Define a public, IBInspectable, Int64 property for the groupId. Although the app developer can set this value via liferay-server-context.plist, it's also a good idea to let them set it in Interface Builder when using the Screenlet. Give this property an initial value of 0. Your Screenlet class should now look like this:

   ```
   public class GuestbookListScreenlet: BaseListScreenlet {

       @IBInspectable public var groupId: Int64 = 0

   }
   ```

2. Override the createPageLoadInteractor method to create and return an instance of your Interactor, GuestbookListPageLoadInteractor. This method includes page and computeRowCount arguments, which you can pass to the Interactor's constructor along with groupId:

   ```
   override public func createPageLoadInteractor(
       page: Int,
       computeRowCount: Bool) -> BaseListPageLoadInteractor {

       return GuestbookListPageLoadInteractor(screenlet: self,
                                              page: page,
                                              computeRowCount: computeRowCount,
                                              groupId: self.groupId)
   }
   ```

3. Create a computed property to get a reference to your delegate, GuestbookListScreenletDelegate:

   ```
   public var guestbookListDelegate: GuestbookListScreenletDelegate? {
       return delegate as? GuestbookListScreenletDelegate
   }
   ```

   You'll use this property to handle the Screenlet's events via the delegate's methods.

4. Override the BaseListScreenlet methods that handle the Screenlet's events. Because these methods correspond to your delegate methods, you'll call your delegate methods in them:

   - onLoadPageResult: Called when the Screenlet loads a page successfully. Override this method to call the superclass's onLoadPageResult method, then call your delegate's screenlet(_:onGuestbookListResponse:) method:

     ```
     override public func onLoadPageResult(page: Int, rows: [AnyObject], rowCount: Int) {
         super.onLoadPageResult(page: page, rows: rows, rowCount: rowCount)

         guestbookListDelegate?.screenlet?(screenlet: self,
             onGuestbookListResponse: rows as! [GuestbookModel])
     }
     ```

- `onLoadPageError`: Called when the Screenlet fails to load a page. Override this method to call the superclass's `onLoadPageError` method, then call your delegate's `screenlet(_:onGuestbookListError:)` method:

```
override public func onLoadPageError(page: Int, error: NSError) {
    super.onLoadPageError(page: page, error: error)

    guestbookListDelegate?.screenlet?(screenlet: self,
        onGuestbookListError: error)
}
```

- `onSelectedRow`: Called when an item is selected in the list. Override this method to call your delegate's `screenlet(_:onGuestbookSelected:)` method:

```
override public func onSelectedRow(_ row: AnyObject) {
    guestbookListDelegate?.screenlet?(screenlet: self,
        onGuestbookSelected: row as! GuestbookModel)
}
```

Awesome! Your Screenlet class is finished. You're also done with Guestbook List Screenlet! The next section in this Learning Path shows you how to create Entry List Screenlet to display a guestbook's entries.

Chapter 49

Creating Entry List Screenlet

In the previous section, you created Guestbook List Screenlet to retrieve and display guestbooks from the Guestbook portlet. You still need a way to retrieve and display each guestbook's entries, though. You'll do this by creating another list Screenlet: Entry List Screenlet. This section walks you through the steps required to create it.

Because you use a consistent development model to create Screenlets, similar Screenlets have similar code. As with guestbooks, it makes sense to display entries in a list using a list Screenlet. This means you can reuse most of Guestbook List Screenlet's code in Entry List Screenlet. You'll therefore create Entry List Screenlet using the same sequence of steps you used to create Guestbook List Screenlet:

1. Getting started: create the Screenlet's folder, and the model class. This model class creates objects that represent entries retrieved from the portlet, making it easier to work with entries in the app.
2. Create the Screenlet's UI (its Theme).
3. Create the Connector. Connectors are Screenlet components that make server calls.
4. Create the Interactor. In list Screenlets, Interactors are Screenlet components that instantiate Connectors and receive their results.
5. Create the delegate. Delegates let other classes respond to the Screenlet's events.
6. Create the Screenlet class. The Screenlet class governs the Screenlet's behavior.

Although this Learning Path section presents complete code snippets, it only discusses the code unique to Entry List Screenlet. Refer back to the previous section for detailed explanations of the code shared with Guestbook List Screenlet.

If you get confused or stuck while creating Guestbook List Screenlet, refer to the finished app that contains the Screenlet code here in GitHub.

49.1 Getting Started with Entry List Screenlet

Like Guestbook List Screenlet, you'll create Entry List Screenlet in its own folder inside your app's project. Create this folder now:

1. In the Finder, create the `EntryListScreenlet` folder inside your root project folder (on the same level as the `GuestbookListScreenlet` folder).

2. Drag and drop the `EntryListScreenlet` folder from the Finder into your Xcode project, under the root project (on the same level as the `GuestbookListScreenlet` folder). In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target. Then click *Finish*. The `EntryListScreenlet` folder now appears in your project.

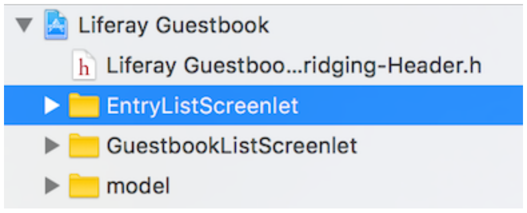


Figure 49.1: After adding the `EntryListScreenlet` folder, your project should look something like this.

Now you're ready to begin!

Creating the Model Class

Recall that you need a model class to represent entities retrieved from Liferay DXP. The model class you'll create for guestbook entries, `EntryModel`, creates `EntryModel` objects that represent guestbook entries retrieved from the Guestbook portlet.

Create the following `EntryModel` class alongside the `GuestbookModel` class in the `model` folder:

1. In the Project navigator, right-click the `model` folder and select *New File*. In the dialog that appears, fill out each screen as follows:

   - Select *iOS* → *Source* → *Cocoa Touch Class*, and click *Next*.
   - Name the class `EntryModel`, set it to extend `NSObject`, select *Swift* for the language, and click *Next*.
   - Make sure the `model` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

   The new class now opens in the editor.

2. Replace the class file's contents with this code:

```
import UIKit

@objc public class EntryModel: NSObject {

    public let attributes: [String:AnyObject]

    public var entryId: Int64 {
        return attributes["entryId"]?.int64Value ?? 0
    }

    public var groupId: Int64 {
        return attributes["groupId"]?.int64Value ?? 0
    }

    public var companyId: Int64 {
        return attributes["companyId"]?.int64Value ?? 0
    }

    public var userId: Int64 {
        return attributes["userId"]?.int64Value ?? 0
    }

    public var userName: String {
        return attributes["userName"] as? String ?? ""
    }

    public var createDate: Int64 {
        return attributes["createDate"]?.int64Value ?? 0
    }

    public var modifiedDate: Int64 {
        return attributes["modifiedDate"]?.int64Value ?? 0
    }

    public var name: String {
        return attributes["name"] as? String ?? ""
    }

    public var email: String {
        return attributes["email"] as? String ?? ""
    }

    public var message: String {
        return attributes["message"] as? String ?? ""
    }

    public var guestbookId: Int64 {
        return attributes["guestbookId"]?.int64Value ?? 0
    }

    //MARK: Initializer

    public init(attributes: [String:AnyObject]) {
        self.attributes = attributes
    }

}
```

Besides working with entries instead of guestbooks, this class is almost identical to `GuestbookModel`. For an explanation of the code, see the article on getting started with Guestbook List Screenlet.

Next, you'll create the Screenlet's UI.

49.2 Creating Entry List Screenlet's UI

Recall that in Liferay Screens for iOS, Screenlet UIs are called *Themes*, and every Screenlet must have at least one Theme. You'll create Entry List Screenlet's Theme with the same steps you used to create Guestbook List Screenlet's Theme:

1. Create your Theme's folder and add it to your Xcode project.
2. Create an XIB file and use it to construct the UI.
3. Create your Theme's View class and set it as the XIB file's custom class.

Creating Your Theme's Folder

Like Guestbook List Screenlet's Theme, you'll create Entry List Screenlet's Theme in a `Themes/Default` folder. Follow these steps to create this folder structure in Entry List Screenlet's folder:

1. In the Finder, create the `Themes` folder inside your project's `EntryListScreenlet` folder. Then create the `Default` folder inside the new `Themes` folder.

2. Drag and drop the `Themes` folder from the Finder into your Xcode project, under the `EntryListScreenlet` folder. In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target (these should be selected by default). Then click *Finish*. The `Themes/Default` folder structure now appears in your project.

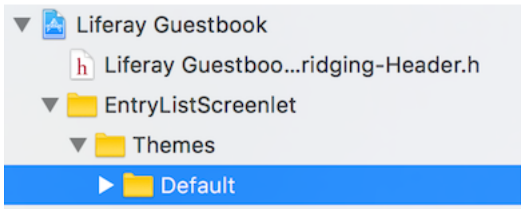


Figure 49.2: After adding the `Themes` folder to Entry List Screenlet, the `Themes/Default` folder structure should look like this in the Project navigator.

Now you're ready to start creating your Theme. First, you'll create its XIB file.

Creating the XIB File

A Theme requires an XIB file to define the UI's components and layout. Use these steps to create your Theme's XIB file:

1. In the Project navigator, right-click the `Default` folder you added above and select *New File*. In the dialog that appears, select *iOS* → *User Interface* → *Empty*, and click *Next*. Name the file `EntryListView_default.xib`, and ensure that *Default* is selected for the save location and group. The *Liferay Guestbook* target should also be selected. Click *Create*. The file then opens in Interface Builder.

2. In Interface Builder, drag and drop a View from the Object Library onto the canvas. Then add a Table View to the View. Set the Table View to take up the entire View.

3. With the Table View selected, open the *Add New Constraints* menu at the bottom-right of the canvas. In this menu, set *Spacing to nearest neighbor* to 0 on all sides, select *Constrain to margins*, and then click the *Add 4 Constraints* button.

Your Theme's XIB is now finished. Next, you'll create your View class.

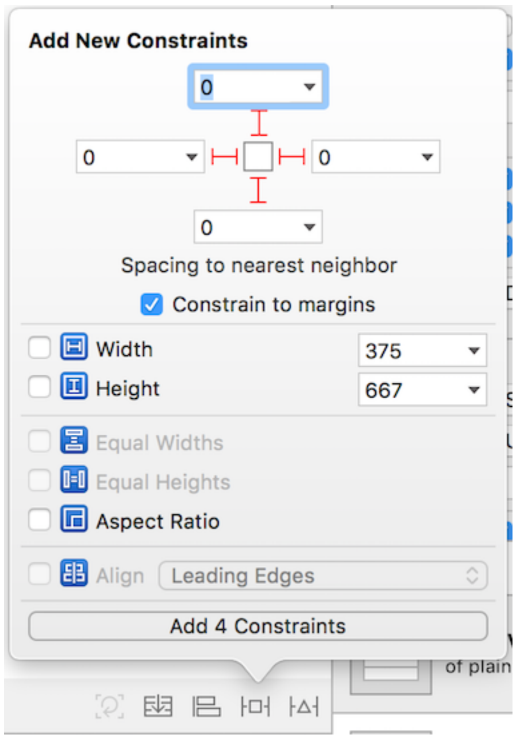
Creating the Theme's View Class

Every Theme needs a View class that controls its behavior. Recall that a list Screenlet's View class gets most of its functionality by extending the `BaseListTableView` class. This lets you focus on the parts of your View class that are unique to your Screenlet.

Follow these steps to create your Screenlet's View class:

1. In the Project navigator, right-click Entry List Screenlet's `Default` folder and select *New File*. In the dialog that appears, fill out each screen as follows:

   - Select *iOS* → *Source* → *Cocoa Touch Class*, and click *Next*.
   - Name the class `EntryListView_default`, set it to extend `BaseListTableView`, select *Swift* for the language, and click *Next*.
   - Make sure the `Default` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.



Add New Constraints

0

0 0

0

Spacing to nearest neighbor

[✓] Constrain to margins

[ ] Width 375

[ ] Height 667

[ ] Equal Widths

[ ] Equal Heights

[ ] Aspect Ratio

[ ] Align Leading Edges

Figure 49.3: Add these constraints to the Table View in the XIB.

2. Replace the class file's contents with this code:

```
import UIKit
import LiferayScreens

class EntryListView_default: BaseListTableView {

    override public func doFillLoadedCell(row: Int, cell: UITableViewCell, object: AnyObject) {
        let entry = object as! EntryModel

        cell.textLabel?.text = entry.message
        cell.detailTextLabel?.text = entry.name
    }

    override open func doCreateCell(_ cellId: String) -> UITableViewCell {
        return UITableViewCell(style: .subtitle, reuseIdentifier: cellId)
    }

    override public func doFillInProgressCell(row: Int, cell: UITableViewCell) {
        cell.textLabel?.text = "Loading..."
    }

}
```

Note that this class is almost identical to Guestbook List Screenlet's View class, `GuestbookListView_default`. The only difference is that `EntryListView_default` handles entries (`EntryModel`) instead of guestbooks (`GuestbookModel`). The `doFillLoadedCell` method sets the cell's main text label to the entry's message, and sets the cell's secondary text label to the name of the person who left the message. This way, a single cell displays both pieces of information. For a description of the code shared with `GuestbookListView_default`, see the article on creating Guestbook List Screenlet's Theme.

3. Return to the Theme's XIB in Interface Builder and set `EntryListView_default` as the the parent View's custom class. To do this, select the Table View's parent View, click the Identity inspector, and enter `EntryListView_default` as the custom class.

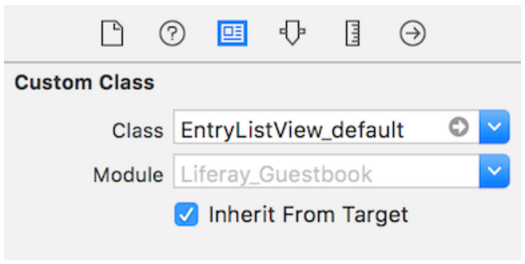

Custom Class
Class EntryListView_default
Module Liferay_Guestbook
☑ Inherit From Target

Figure 49.4: In the XIB file, set the custom class of the Table View's parent View to `EntryListView_default`.

4. With the Theme's XIB still open in Interface Builder, set the parent View's `tableView` outlet to the Table View. To do this, select the parent View and click the Connections inspector. In the *Outlets* section, drag and drop from the `tableView`'s circle icon (on mouseover, it turns into a plus icon) to the Table View in the XIB. The new outlet then appears in the Connections inspector.

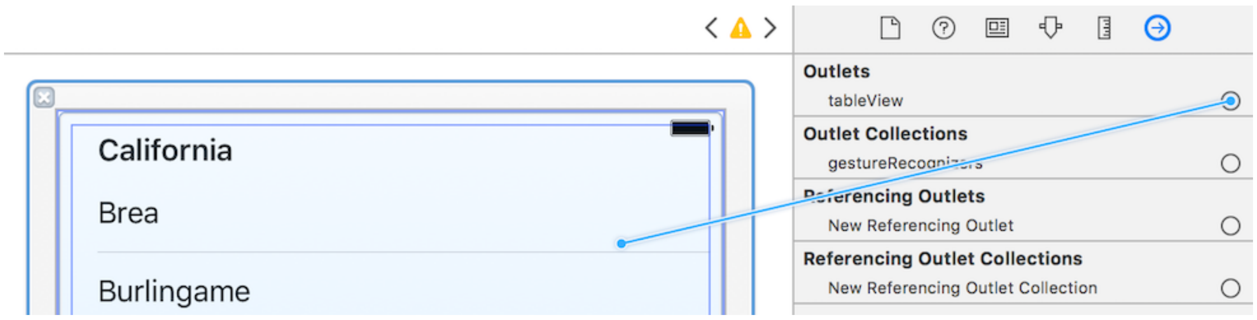


Figure 49.5: Drag and drop from the `tableView` outlet to the Table View in the XIB.

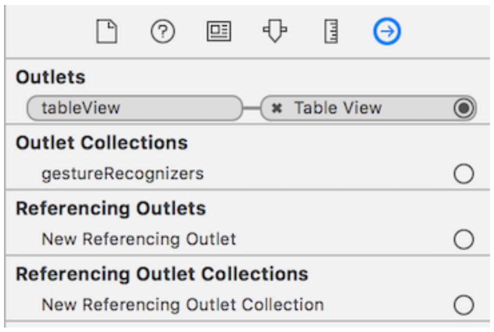

Outlets
tableView — ✖ Table View
Outlet Collections
gestureRecognizers
Referencing Outlets
New Referencing Outlet
Referencing Outlet Collections
New Referencing Outlet Collection

Figure 49.6: After creating the connection, the outlet looks like this in the Connections inspector.

Great! Your Theme is finished. Next, you'll create Entry List Screenlet's Connector.

49.3 Creating Entry List Screenlet's Connector

Recall that Connectors are Screenlet components that make server calls. Also recall that by making your server calls in Connectors instead of Interactors, you gain an additional layer of abstraction.

In this article, you'll create Entry List Screenlet's Connector. Because this Connector is so similar to that of Guestbook List Screenlet, the steps to create it aren't explained in detail. Focus is instead placed on the few places in the code where the Connectors diverge. For a full explanation of the code, see the article on creating Guestbook List Screenlet's Connector.

Creating Your Connector's Folder

Follow these steps to create your Connector's folder:

1. In the Finder, create the `Connector` folder inside your project's `EntryListScreenlet` folder.

2. Drag and drop the `Connector` folder from the Finder into your Xcode project, under the `EntryListScreenlet` folder. In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target. Then click *Finish*. The `Connector` folder now appears in your project.

Now you're ready to create the Connector.

Creating the Connector

Recall that list Screenlet Connectors must extend the `PaginationLiferayConnector` class. Your list Screenlet's Connector class must also contain any properties it needs to make the server call, and an initializer that sets them. To support pagination, the initializer must also contain the following arguments, which you'll pass to the superclass initializer:

- `startRow`: The number representing the page's first row.
- `endRow`: The number representing the page's last row.
- `computeRowCount`: Whether to call the Connector's `doAddRowCountServiceCall` method.

Follow these steps to create Guestbook List Screenlet's Connector:

1. In the Project navigator, right-click the `Connector` folder you added above and select *New File*. In the dialog that appears, fill out each screen as follows:

    - Select *iOS → Source → Cocoa Touch Class*, and click *Next*.
    - Name the class `EntryListPageLiferayConnector`, set it to extend `PaginationLiferayConnector`, select *Swift* for the language, and click *Next*.
    - Make sure the `Connector` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

2. Replace the class file's contents with this code:

```
import UIKit
import LiferayScreens

class EntryListPageLiferayConnector: PaginationLiferayConnector {

    public let groupId: Int64
    public let guestbookId: Int64

    //MARK: Initializer

    public init(startRow: Int, endRow: Int, computeRowCount: Bool, groupId: Int64,
        guestbookId: Int64) {

            self.groupId = groupId
            self.guestbookId = guestbookId

            super.init(startRow: startRow, endRow: endRow, computeRowCount: computeRowCount)
    }

    //MARK: PaginationLiferayConnector

    public override func doAddPageRowsServiceCall(session: LRBatchSession, startRow: Int, endRow: Int,
        obc: LRJSONObjectWrapper?) {

        let service = LREntryService_v7(session: session)

        do {
            try service!.getEntriesWithGroupId(groupId, guestbookId: guestbookId,
                    start: Int32(startRow), end: Int32(endRow))
        }
        catch {
            // the service method returns nil because the request is sent later, in batch
        }

    }

    override public func doAddRowCountServiceCall(session: LRBatchSession) {
        let service = LREntryService_v7(session: session)

        do {
            try service!.getEntriesCount(withGroupId: groupId, guestbookId: guestbookId)
        }
        catch {
            // the service method returns nil because the request is sent later, in batch
        }
    }

}
```

This class is almost identical to Guestbook List Screenlet's Connector class, `GuestbookListPageLiferayConnector`. The only differences are due to the service calls. To define the guestbook to retrieve entries from, `EntryListPageLiferayConnector` needs a `guestbookId` property. It then uses this property with the service methods `getEntriesWithGroupId` and `getEntriesCount` to retrieve the entries and number of entries, respectively. Also note that the service is an `LREntryService_v7` instance.

Nicely done! Now that Entry List Screenlet has a Connector, you must create its Interactor. The next article shows you how to do this.

49.4 Creating Entry List Screenlet's Interactor

Recall that list Screenlets require an Interactor to instantiate the Connector and receive the server call's results. In this article, you'll create Entry List Screenlet's Interactor.

Because this Interactor is so similar to that of Guestbook List Screenlet, the steps to create it aren't explained in detail. Focus is instead placed on the few places in the code where the Interactors diverge. For a full explanation of the code, see the article on creating Guestbook List Screenlet's Interactor.

Creating Your Interactor's Folder

Follow these steps to create your Interactor's folder:

1. In the Finder, create the `Interactor` folder inside your project's `EntryListScreenlet` folder.

2. Drag and drop the `Interactor` folder from the Finder into your Xcode project, under the `EntryListScreenlet` folder. In the dialog that appears, select *Copy items if needed*, *Create groups*, and the *Liferay Guestbook* target. Then click *Finish*. The `Interactor` folder now appears in your project.

Now you're ready to create the Interactor.

Creating the Interactor

Recall that the Interactor class of a list Screenlet that implements fluent pagination must extend the `BaseListPageLoadInteractor` class. Your Interactor class must also contain any properties the Screenlet needs, and an initializer that sets them. This initializer also needs arguments for the following properties, which it passes to the superclass initializer:

- `screenlet`: A `BaseListScreenlet` reference. This ensures the Interactor always has a Screenlet reference.
- `page`: The page number to retrieve.
- `computeRowCount`: Whether to call the Connector's `doAddRowCountServiceCall` method.

Follow these steps to create Entry List Screenlet's Interactor:

1. In the Project navigator, right-click the `Interactor` folder you added above and select *New File*. In the dialog that appears, fill out each screen as follows:

    - Select *iOS* → *Source* → *Cocoa Touch Class*, and click *Next*.
    - Name the class `EntryListPageLoadInteractor`, set it to extend `BaseListPageLoadInteractor`, select *Swift* for the language, and click *Next*.
    - Make sure the `Interactor` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

2. Replace the class file's contents with this code:

```
import UIKit
import LiferayScreens

class EntryListPageLoadInteractor: BaseListPageLoadInteractor {

    private let groupId: Int64
    private let guestbookId: Int64

    init(screenlet: BaseListScreenlet,
         page: Int,
         computeRowCount: Bool,
         groupId: Int64,
         guestbookId: Int64) {

        self.groupId = (groupId ≠ 0) ? groupId : LiferayServerContext.groupId
        self.guestbookId = guestbookId

        super.init(screenlet: screenlet, page: page, computeRowCount: computeRowCount)
    }
    public override func createListPageConnector() -> PaginationLiferayConnector {
        let screenlet = self.screenlet as! BaseListScreenlet

        return EntryListPageLiferayConnector(
            startRow: screenlet.firstRowForPage(self.page),
            endRow: screenlet.firstRowForPage(self.page + 1),
            computeRowCount: self.computeRowCount,
            groupId: groupId,
            guestbookId: guestbookId)
    }

    override public func convertResult(_ serverResult: [String:AnyObject]) -> AnyObject {

        return EntryModel(attributes: serverResult)
    }

    override public func cacheKey(_ op: PaginationLiferayConnector) -> String {
        return "\(groupId)-\(guestbookId)"
    }

}
```

This class is almost identical to Guestbook List Screenlet's Interactor, `GuestbookListPageLoadInteractor`. The only real difference is that `EntryListPageLoadInteractor` handles entries. It therefore needs a `guestbookId` variable to define the guestbook to retrieve entries from. This variable is set in the initializer and then used in the `createListPageConnector` method to create a `EntryListPageLiferayConnector` instance. The `convertResult` method receives each `[String:AnyObject]` entry from the server and transforms it into an `EntryModel` object. Also recall that the `cacheKey` method must return a key that can be used with online mode. For entries, a combination of the `groupId` and `guestbookId` is a sufficient key.

Great! Your Interactor is finished. Next, you'll create the delegate.

49.5 Creating Entry List Screenlet's Delegate

Recall that a delegate lets other classes respond to your Screenlet's actions. Like you did for Guestbook List Screenlet, you'll create a delegate for Entry List Screenlet that can respond to a successful server call, a failed server call, and an item selection in the list. This delegate must therefore define these methods:

- `screenlet(_:onEntryListResponse:)`: Receives the `EntryModel` results when the server call succeeds. This lets app developers respond to a successful server call.
- `screenlet(_:onEntryListError:)`: Receives the `NSError` object when the server call fails. This lets app developers respond to a failed server call.
- `screenlet(_:onEntrySelected:)`: Receives the `EntryModel` when a user selects it in the list. This lets app developers respond to an entry selection by the user.

You'll create this delegate in the same file as the Screenlet class. Later, you'll finish creating the Screenlet class itself.

Follow these steps to create Entry List Screenlet's delegate:

1. In the Project navigator, right-click the `EntryListScreenlet` folder and select *New File*. In the dialog that appears, fill out each screen as follows:

    - Select *iOS* → *Source* → *Cocoa Touch Class*, and click *Next*.

    - Name the class `EntryListScreenlet`, set it to extend `BaseListScreenlet`, select *Swift* for the language, and click *Next*.
    - Make sure the `EntryListScreenlet` folder and group is selected, as well as the *Liferay Guestbook* target (these should be selected by default). Click *Create*.

2. In the new `EntryListScreenlet` class, import `LiferayScreens`, make the class `public`, and delete any placeholder comments in the class body.

3. In between the `import` statements and the class declaration, add the following code:

    ```
    @objc public protocol EntryListScreenletDelegate : BaseScreenletDelegate {

        @objc optional func screenlet(screenlet: EntryListScreenlet,
                              onEntryListResponse entries: [EntryModel])

        @objc optional func screenlet(screenlet: EntryListScreenlet,
                              onEntryListError error: NSError)

        @objc optional func screenlet(screenlet: EntryListScreenlet,
                              onEntrySelected entry: EntryModel)

    }
    ```

    This delegate is almost identical to that of Guestbook List Screenlet, `GuestbookListScreenletDelegate`. The only difference is that `EntryListScreenletDelegate` works with entries instead of guestbooks.

The contents of `EntryListScreenlet.swift` should now look like this:

```
import UIKit
import LiferayScreens

@objc public protocol EntryListScreenletDelegate : BaseScreenletDelegate {

    @objc optional func screenlet(screenlet: EntryListScreenlet,
                          onEntryListResponse entries: [EntryModel])

    @objc optional func screenlet(screenlet: EntryListScreenlet,
                          onEntryListError error: NSError)

    @objc optional func screenlet(screenlet: EntryListScreenlet,
                          onEntrySelected entry: EntryModel)

}

public class EntryListScreenlet: BaseListScreenlet {

}
```

Nice work! Now you're ready to complete the Screenlet class.

49.6 Creating the Screenlet Class

Recall that the Screenlet class is the main component that governs the Screenlet's behavior. Also recall that a list Screenlet's class must do the following:

- Extend `BaseListScreenlet`. Since `BaseListScreenlet` provides most of the functionality needed by Screenlet classes, extending it lets you focus on the functionality unique to your Screenlet.

- Define the configurable IBInspectable properties the Screenlet needs.
- Create and return an instance of your Interactor.
- Respond to the Screenlet's events via the delegate.

The Screenlet class you created when you created the delegate is currently empty. You'll complete it now. Follow these steps to complete the EntryListScreenlet class in EntryListScreenlet.swift:

1. Define public, IBInspectable, Int64 properties for the groupId and guestbookId. Although the app developer can set groupId via liferay-server-context.plist, and the guestbookId is set dynamically, it's also a good idea to let the developer set their values in Interface Builder. Give both properties an initial value of 0. Your Screenlet class should now look like this:

```
public class EntryListScreenlet: BaseListScreenlet {

    @IBInspectable public var groupId: Int64 = 0
    @IBInspectable public var guestbookId: Int64 = 0
}
```

2. Override the createPageLoadInteractor method to create and return an instance of EntryListPageLoadInteractor. This method includes page and computeRowCount arguments, which you pass to the Interactor's constructor along with groupId and guestbookId:

```
override public func createPageLoadInteractor(
    page: Int,
    computeRowCount: Bool) -> BaseListPageLoadInteractor {

    return EntryListPageLoadInteractor(screenlet: self,
                                        page: page,
                                        computeRowCount: computeRowCount,
                                        groupId: self.groupId,
                                        guestbookId: self.guestbookId)
}
```

3. Create a computed property to get a reference to EntryListScreenletDelegate:

```
public var entryListDelegate: EntryListScreenletDelegate? {
    return delegate as? EntryListScreenletDelegate
}
```

   You'll use this property to handle the Screenlet's events via the delegate's methods.

4. Override the BaseListScreenlet methods that handle the Screenlet's events. Because these events correspond to those handled by your delegate, you'll call the corresponding delegate methods in these BaseListScreenlet methods:

   - onLoadPageResult: Called when the Screenlet loads a page successfully. Override this method to call the superclass's onLoadPageResult method, then call your delegate's screenlet(_:onEntryListResponse:) method:

```
override public func onLoadPageResult(page: Int, rows: [AnyObject], rowCount: Int) {
    super.onLoadPageResult(page: page, rows: rows, rowCount: rowCount)

    entryListDelegate?.screenlet?(screenlet: self, onEntryListResponse: rows as! [EntryModel])
}
```

- `onLoadPageError`: Called when the Screenlet fails to load a page. Override this method to call the superclass's `onLoadPageError` method, then call your delegate's `screenlet(_:onEntryListError:)` method:

```
override public func onLoadPageError(page: Int, error: NSError) {
    super.onLoadPageError(page: page, error: error)

    entryListDelegate?.screenlet?(screenlet: self, onEntryListError: error)
}
```

- `onSelectedRow`: Called when an item is selected in the list. Override this method to call your delegate's `screenlet(_:onEntrySelected:)` method:

```
override public func onSelectedRow(_ row: AnyObject) {
    entryListDelegate?.screenlet?(screenlet: self, onEntrySelected: row as! EntryModel)
}
```

Awesome! Your Screenlet class is finished. You're also done with Entry List Screenlet! Now you're ready to use Entry List Screenlet alongside Guestbook List Screenlet. The following section of this Learning Path concludes with both Screenlets working together in harmony.

Chapter 50

Using Your Screenlets

Now that you have the Guestbook List and Entry List Screenlets, you're ready to put them to work. As you'll see, using these Screenlets isn't much more difficult than using Login Screenlet. This is an advantage of Screenlets; it typically takes only a few minutes to get them up and running.

To add your Screenlets to the app, you'll follow these steps:

1. Create the entries scene.
2. Add Guestbook List Screenlet to the guestbooks scene.
3. Add Entry List Screenlet to the entries scene.

If you get confused or stuck at any point in this section of the Learning Path, refer to the finished app that contains the Screenlet code here in GitHub.

First, you'll create the entries scene.

50.1 Creating the Entries Scene

Currently, the login and guestbooks scenes are the only two scenes in your app. The login scene contains Login Screenlet, and you'll put Guestbook List Screenlet in the guestbooks scene. Before you can use Entry List Screenlet, you must create a scene to put it in: the entries scene.

In this article, you'll use these steps to create the entries scene:

1. Add a new view controller to your storyboard, and create a segue to it from the guestbooks scene.
2. Create the entries scene's view controller class.

Adding a View Controller to the Storyboard

Follow these steps to create a view controller for the entries scene:

1. Open your storyboard and drag and drop a *View Controller* from the Object Library to the right of the guestbooks scene.

2. With the new view controller selected in the storyboard, open the Attributes inspector and uncheck *Adjust Scroll View Insets*. This ensures that the scene's contents are flush with the navigation bar.

3. Create a segue from the guestbooks scene's view controller to the new view controller. To do this, control-drag from the guestbooks scene's view controller to the new view controller. In the dialog that appears upon releasing your mouse button, select *show* for the segue type. The segue now connects the two view controllers.

4. Click the new segue, and then enter the Attributes inspector. Enter *entriessegue* in the *Identifier* field, and press *return*. Later, you'll use this identifier to perform the segue programmatically when a user selects a guestbook in Guestbook List Screenlet.

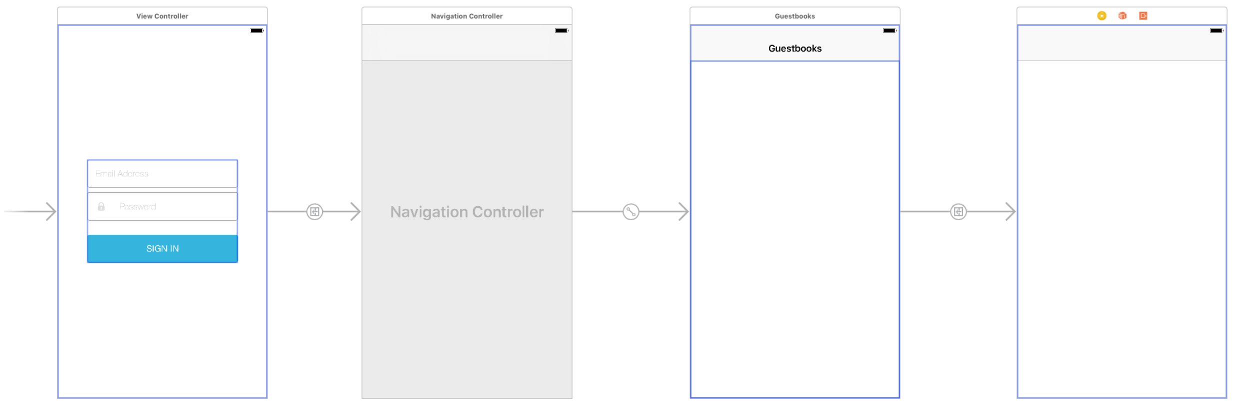


Figure 50.1: The entries scene now exists to the right of the guestbooks scene, with a segue connecting the two scenes.

Great! The entries scene now exists, and there's a segue going to it from the guestbooks scene. Next, you'll create the entries scene's view controller class.

Creating the Entries Scene's View Controller Class

Each view controller must have a class that controls its behavior. In this section, you'll create this class for the entries scene's view controller. In the storyboard, you'll then set this class as the view controller's custom class.

Follow these steps to create the entries scene's view controller class:

1. Right-click the `Liferay Guestbook` folder in Xcode's project navigator and select *New File*. In the *iOS → Source* section of the dialog that appears, select *Cocoa Touch Class* and click *Next*.

2. The next screen in the dialog lets you set the class's name, subclass, and language. You can also choose whether to create an XIB file for the class. Enter the following information and click *Next*:

   - **Class:** `EntriesViewController`
   - **Subclass of:** `UIViewController`
   - **Also create XIB file:** Unchecked
   - **Language:** Swift

3. The final screen in the dialog lets you set the class's location, group, and targets. Make sure *Liferay Guestbook* is selected for both the *Group* and *Targets* menus (it should be by default), and click *Create*.

4. `EntriesViewController` needs a `GuestbookModel` variable to hold the guestbook it shows entries from. Add this variable to the top of the class:

   ```
   var selectedGuestbook: GuestbookModel?
   ```

   As its name implies, this variable holds the guestbook the user selects in Guestbook List Screenlet.

5. In the storyboard, select the entries scene's view controller. In the Identity inspector, set `EntriesViewController` as the custom class.

Nice work! The entries scene's view controller now has a class that governs its behavior. Now you're ready to put your Screenlets to use.

50.2 Using Guestbook List Screenlet

The steps for using Guestbook List Screenlet are the same as those for using any Screenlet:

1. Insert the Screenlet in the storyboard scene where you want it to appear. You do this by adding an empty view to the scene, and then setting the Screenlet class as the view's custom class.
2. Conform the scene's view controller class to the Screenlet's delegate protocol. This lets the view controller respond to the Screenlet's events.

You'll follow these steps to use Guestbook List Screenlet in the guestbooks scene. You'll also take an extra step to trigger the segue to the entries scene when a user selects a guestbook.

Adding Guestbook List Screenlet to the Guestbooks Scene

Follow these steps to add Guestbook List Screenlet to the guestbooks scene:

1. In your storyboard, first select the guestbooks scene's view controller. Then drag and drop a plain view (`UIView`) from the Object Library to the view controller. In the outline view, this new view should be nested under the view controller's existing view.

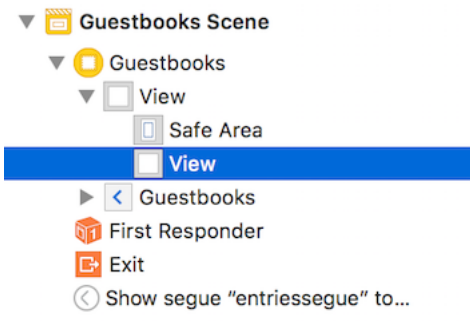


Figure 50.2: The new view is nested under the view controller's existing view.

2. Resize the new view to take up all the space below the navigation bar. With the new view selected, open the *Add New Constraints* menu at the bottom-right of the canvas. In this menu, set *Spacing to nearest neighbor* to 0 on all sides, and click the *Add 4 Constraints* button.

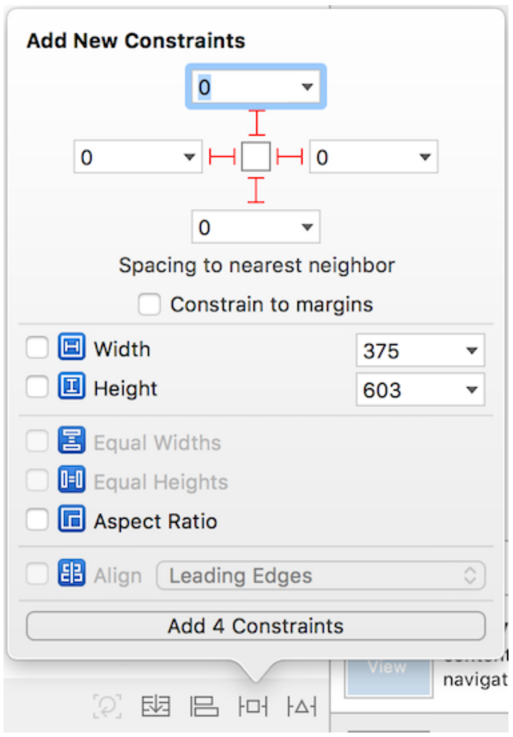


Figure 50.3: Set the new view's *Spacing to nearest neighbor* constraints to 0 on all sides.

3. With the new view still selected, open the Identity inspector and set the view's custom class to `GuestbookListScreenlet`. The view now appears as Guestbook List Screenlet in the outline view.

Fantastic! The guestbooks scene now contains Guestbook List Screenlet. Next, you'll conform the scene's view controller class to the Screenlet's delegate.

Conforming to the Screenlet's Delegate Protocol

Recall that a view controller can respond to a Screenlet's events by conforming to the Screenlet's delegate protocol. To respond to Guestbook List Screenlet's events, `GuestbooksViewController` (the guestbooks scene's view controller class) must conform to the `GuestbookListScreenletDelegate` protocol. You created this delegate when creating the Screenlet. This delegate defines methods for responding to the success or failure to retrieve guestbooks, and the selection of a guestbook in the list.

Follow these steps to conform `GuestbooksViewController` to the `GuestbookListScreenletDelegate` protocol:

1. Import `LiferayScreens`, and in the class declaration set `GuestbooksViewController` to adopt the `GuestbookListScreenletDelegate` protocol. The first few lines of the class should look like this:

```
import UIKit
import LiferayScreens

class GuestbooksViewController: UIViewController, GuestbookListScreenletDelegate {...
```

2. Implement the `GuestbookListScreenletDelegate` method `screenlet(_:onGuestbookListResponse:)`. Recall that this method lets you respond to a successful server call. Its arguments include the `GuestbookModel` objects that result from such a call. Since the Screenlet already displays these objects, you don't need to do anything in this method:

```
func screenlet(screenlet: GuestbookListScreenlet,
    onGuestbookListResponse guestbooks: [GuestbookModel]) {

}
```

3. Implement the `GuestbookListScreenletDelegate` method `screenlet(_:onGuestbookListError:)`. Recall that this method lets you respond to a failed server call. Its arguments include the `NSError` object that results from such a call. You don't have to do anything in this method, but it's a good idea to print the error:

```
func screenlet(screenlet: GuestbookListScreenlet, onGuestbookListError error: NSError) {
    print("Failed to retrieve guestbooks: \(error.localizedDescription)")
}
```

4. Implement the `GuestbookListScreenletDelegate` method `screenlet(_:onGuestbookSelected:)`. Recall that this method lets you respond when a user selects a guestbook in the list. It does so by including the selected `GuestbookModel` object in its arguments. When a user selects a guestbook, the app should transition to the entries scene and display that guestbook's entries with Entry List Screenlet. To do this, you must trigger the segue to the entries scene by using the method `performSegue(withIdentifier:sender:)` with the segue's ID and the selected `GuestbookModel`. Recall that you assigned the segue's ID, `entriessegue`, when you created the segue. Including the selected `GuestbookModel` lets Entry List Screenlet know which guestbook to display entries from:

```
func screenlet(screenlet: GuestbookListScreenlet,
    onGuestbookSelected guestbook: GuestbookModel) {

    performSegue(withIdentifier: "entriessegue", sender: guestbook)
}
```

5. Next, you must set the segue's destination view controller to an `EntriesViewController` instance, and set that instance's `selectedGuestbook` variable to the selected guestbook. This ensures that Entry List Screenlet receives the guestbook you sent in `performSegue(withIdentifier:sender:)`.

   You do this by overriding the `prepare(for:sender:)` method. This method is called by `performSegue(withIdentifier:send`
   before the segue occurs. The `prepare(for:sender:)` method's sender parameter receives the guestbook sent by `performSegue(withIdentifier:sender:)`. Currently, `prepare(for:sender:)` is commented out at the bottom of `GuestbooksViewController`. Xcode created this method for you when you used the Cocoa Touch Class template to create a view controller class. Uncomment the method, and replace it with this:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {

    guard segue.identifier ==  "entriessegue",
        let entriesViewController = segue.destination as? EntriesViewController,
        let selectedGuestbook = sender as? GuestbookModel
        else {return}

    entriesViewController.selectedGuestbook = selectedGuestbook
}
```

   The guard statement ensures that the code only runs when the segue ID is `entriessegue`, the destination view controller is `EntriesViewController`, and the sender is a `GuestbookModel`. In other words, the code only runs in preparation for the segue to the entries scene. The code that runs when that condition is met is only one line long:

```
entriesViewController.selectedGuestbook = selectedGuestbook
```

This sets the guestbook received by `prepare(for:sender:)` to the guestbook in `EntriesViewController`. As the variable names indicate, this is the guestbook the user selects in Guestbook List Screenlet.

6. Now you must get a Guestbook List Screenlet reference. You'll do this by creating an outlet to the Screenlet. Return to your storyboard and enter the Assistant editor to display `GuestbooksViewController`'s code and the storyboard side by side. With Guestbook List Screenlet selected in the storyboard, Control-drag from the Screenlet to the `GuestbooksViewController` class, just below the class declaration. Release your mouse button, enter the following information in the dialog that appears, and click *Connect*:

    - **Connection:** Outlet
    - **Name:** screenlet
    - **Type:** GuestbookListScreenlet
    - **Storage:** Weak

    Xcode then adds the following code inside the `GuestbooksViewController` class:

    ```
    @IBOutlet weak var screenlet: GuestbookListScreenlet!
    ```

7. Use this new `screenlet` variable to set the view controller as the Screenlet's delegate. Do this in the `viewDidLoad()` method by deleting the placeholder comment and inserting this code below the call to `super.viewDidLoad()`:

    ```
    self.screenlet.delegate = self
    ```

Great! The guestbooks scene now contains Guestbook List Screenlet. Next, you'll use Entry List Screenlet in the entries scene.

50.3 Using Entry List Screenlet

You'll use Entry List Screenlet the same way you use any Screenlet: insert it in a storyboard scene, then conform the scene's view controller class to the Screenlet's delegate protocol. You'll follow these steps now to use Entry List Screenlet in the entries scene.

Adding Entry List Screenlet to the Entries Scene

Follow these steps to add Entry List Screenlet to the entries scene:

1. In your storyboard, select the entries scene's view controller. Then drag and drop a plain view (`UIView`) from the Object Library to the view controller. In the outline view, this new view should be nested under the view controller's existing view.

2. Resize the new view to take up all the space below the navigation bar. Then open the *Add New Constraints* menu at the bottom-right of the canvas. In this menu, set *Spacing to nearest neighbor* to 0 on all sides, and click the *Add 4 Constraints* button.

3. With the new view still selected, open the Identity inspector and set the view's custom class to `EntryListScreenlet`. The view now appears as Entry List Screenlet in the outline view.

Great! The entries scene now contains Entry List Screenlet. Next, you'll conform the scene's view controller class to the Screenlet's delegate.

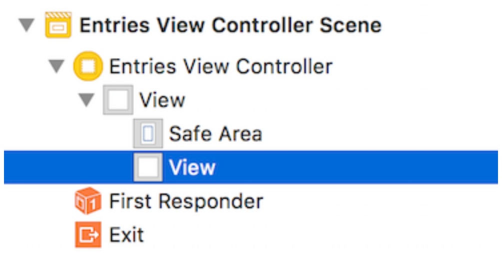


Figure 50.4: The new view is nested under the view controller's existing view.

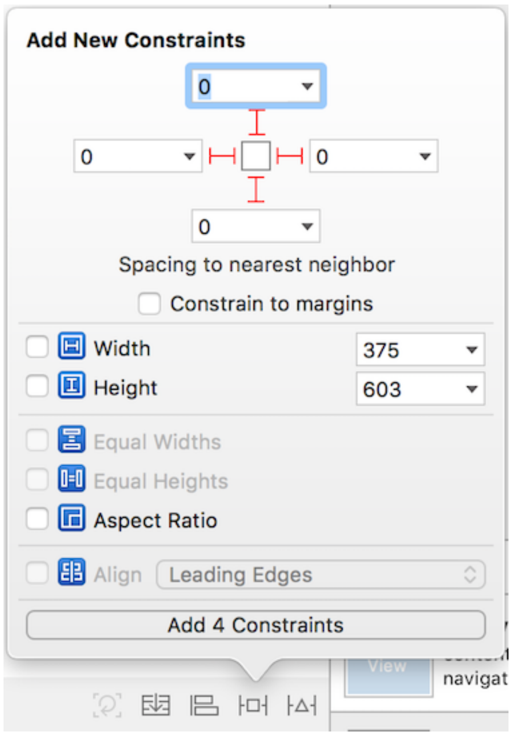


Figure 50.5: Set the new view's *Spacing to nearest neighbor* constraints to 0 on all sides.

Conforming to the Screenlet's Delegate Protocol

To respond to Entry List Screenlet's events, `EntriesViewController` must conform to the `EntryListScreenletDelegate` protocol. You created this delegate when creating the Screenlet. This delegate defines methods for responding to the success or failure to retrieve entries, and the selection of an entry in the list.

Follow these steps to conform `EntriesViewController` to the `EntryListScreenletDelegate` protocol:

1. Import `LiferayScreens`, and in the class declaration set `EntriesViewController` to adopt the `EntryListScreenletDelegate` protocol. The first few lines of the class should look like this:

```
import UIKit
import LiferayScreens

class EntriesViewController: UIViewController, EntryListScreenletDelegate {...
```

2. Implement the `EntryListScreenletDelegate` method `screenlet(_:onEntryListResponse:)`. Recall that this method lets you respond to a successful server call. Its arguments include the `EntryModel` objects

that result from such a call. Since the Screenlet already displays these objects, you don't need to do anything in this method:

```
func screenlet(screenlet: EntryListScreenlet, onEntryListResponse entries: [EntryModel]) {

}
```

3. Implement the `EntryListScreenletDelegate` method `screenlet(_:onEntryListError:)`. Recall that this method lets you respond to a failed server call. Its arguments include the resulting `NSError` object. You don't have to do anything in this method, but it's a good idea to print the error:

```
func screenlet(screenlet: EntryListScreenlet, onEntryListError error: NSError) {
    print("Failed to retrieve guestbook entries: \(error.localizedDescription)")
}
```

4. Implement the `EntryListScreenletDelegate` method `screenlet(_:onEntrySelected:)`. Recall that this method lets you respond when a user selects an entry in the list. It does so by including the selected `EntryModel` object in its arguments. Since there's currently not a scene or other action that handles detailed information about an entry, you don't need to do anything in this method:

```
func screenlet(screenlet: EntryListScreenlet, onEntrySelected entry: EntryModel) {

}
```

5. Now you must get an Entry List Screenlet reference. You'll do this by creating an outlet to the Screenlet. Return to your storyboard and enter the Assistant editor to display `EntriesViewController`'s code and the storyboard side by side. With Entry List Screenlet selected in the storyboard, Control-drag from the Screenlet to the `EntriesViewController` class, just below the class declaration. Release your mouse button, enter the following information in the dialog that appears, and click *Connect*:

    - **Connection:** Outlet
    - **Name:** screenlet
    - **Type:** EntryListScreenlet
    - **Storage:** Weak

   Xcode then adds the following code inside the `EntriesViewController` class:

```
@IBOutlet weak var screenlet: EntryListScreenlet!
```

6. Use this new `screenlet` variable to set the view controller as the Screenlet's delegate. Do this in the `viewDidLoad()` method by deleting the placeholder comment and inserting this code below the call to `super.viewDidLoad()`:

```
self.screenlet.delegate = self
```

7. Next, you must set the guestbook the Screenlet retrieves entries from. To do this, set the Screenlet's `guestbookId` property to the selected guestbook's ID, immediately below the Screenlet's delegate assignment in the `viewDidLoad()` method:

```
self.screenlet.guestbookId = selectedGuestbook!.guestbookId
```

8. Lastly, you must set the selected guestbook's name to the navigation bar's title. This lets the scene reflect the guestbook selection in the UI. To do this, add the following line of code at the end of the `viewDidLoad()` method:

   ```
   self.navigationItem.title = selectedGuestbook!.name
   ```

   Your `viewDidLoad()` method should now look like this:

   ```
   override func viewDidLoad() {
       super.viewDidLoad()

       self.screenlet.delegate = self
       self.screenlet.guestbookId = selectedGuestbook!.guestbookId
       self.navigationItem.title = selectedGuestbook!.name
   }
   ```

Now you're ready to test your handiwork. Make sure your portal containing the Guestbook portlet is running, and that the portlet contains a couple guestbooks that have entries. Then run the app and log in with your credentials. You should see the list of guestbooks displayed by Guestbook List Screenlet. Selecting a guestbook in the list takes you to the entries scene, which uses Entry List Screenlet to display a list of that guestbook's entries. Pressing the back button returns you to the guestbooks scene, where you can select a different guestbook.

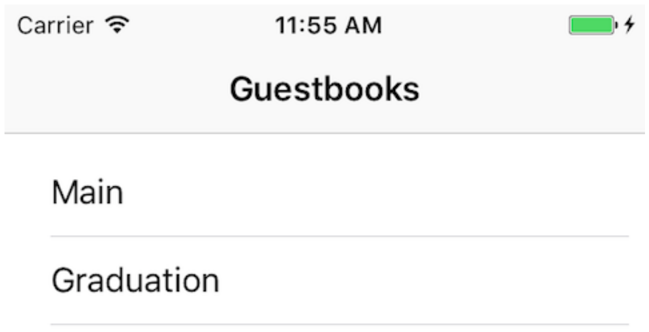


Figure 50.6: After login, Guestbook List Screenlet displays the list of guestbooks from the portlet.

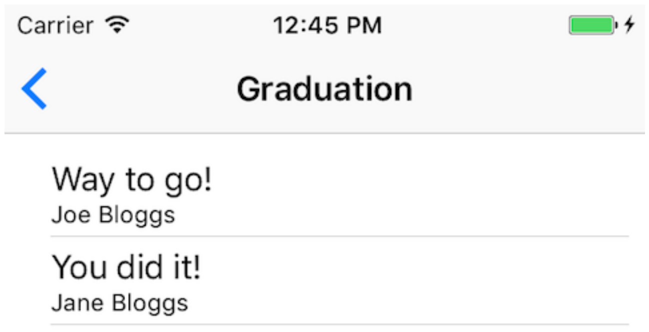


Figure 50.7: Selecting a guestbook displays a list of that guestbook's entries via Entry List Screenlet.

Congratulations! Now you know how to use Liferay Screens and create your own Screenlets. This opens up a world of possibilities for developing apps that leverage Liferay DXP. Although you learned a great deal in this Learning Path, there's still more. You can customize your Screenlet's appearance, package Screenlets and Themes for redistribution, and even add multiple actions to a Screenlet. These topics, and more, are covered in the tutorials on iOS apps with Liferay Screens.

Chapter 51

Tooling

Liferay is very flexible when it comes to supporting different development tooling. Instead of being pigeonholed into using a specific tool, Liferay works with whatever tools you like to use. This set of tutorials describes some of the more popular tools that are used to develop for Liferay. If you're a newbie looking for the best development tool for Liferay, or even a seasoned veteran looking for a tool you may like more than your current setup, this section will answer your tooling questions.

Chapter 52

Liferay @ide@

Liferay @ide@ provides an all-in-one, integrated development environment based on Eclipse that supports development for Liferay DXP. @ide@ includes Liferay IDE plugins and additional enterprise-only features like

- Kaleo Designer for Java
- WebSphere support
- Pre-installed Liferay Digital Enterprise server

@ide@ is also designed to work with build tools such as Gradle and Maven, and configuration tools like BndTools.

Liferay @ide@ makes Liferay development easier. There are editors for Service Builder files, workflow definitions, layout templates, and more. You'll find wizards for creating every kind of Liferay project there is, snippet for tag libraries, and auto-deploy of changes to plugins.

In this section of tutorials, you'll learn how to install Liferay @ide@ and develop/manage Liferay modules using Liferay Workspace and other technologies.

52.1 Installing Liferay @ide@

Liferay @ide@ is a plugin for Eclipse that provides many Liferay-specific features and additional enterprise only features. You can install it into your existing Eclipse environment, or Liferay provides a bundled version. In this tutorial, you'll learn the different methods available for installing Liferay @ide@. Before beginning the installation process, view @ide@'s Compatibility Matrix to get acquainted with its supported Liferay versions and application servers.

Important: If you're upgrading your Liferay @ide@ instance to version 3.1, you must install a new @ide@ bundle. You cannot install an update for this upgrade. Your Liferay Workspace instance and its contents are backwards compatible and can be copied to the new 3.1 version.

Install the Liferay @ide@ Bundle

1. Download and install Java. Liferay DXP runs on Java, so you'll need it to run everything else. Because you'll be developing apps for Liferay DXP in Liferay @ide@, the Java Development Kit (JDK) is required. It is an enhanced version of the Java Environment used for developing new Java technology. You can download the Java SE JDK from the Java Downloads page.

2. Download the Liferay @ide@ installer. Be sure to choose the installer appropriate for your operating system (e.g., Windows, MacOS, Linux).

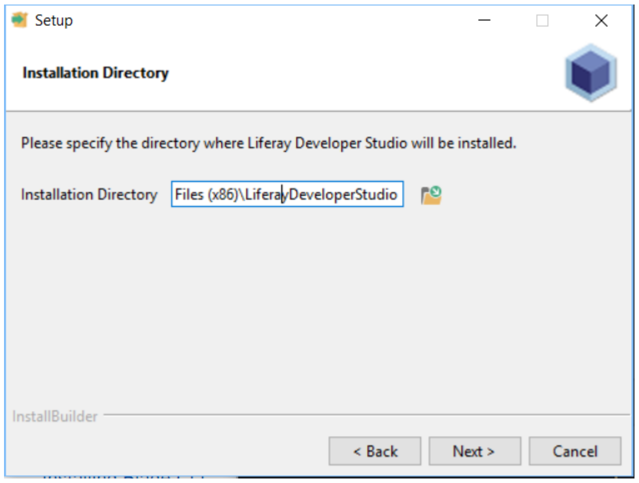
   You may be prompted for your liferay.com username and password before downloading the Liferay DXP installer. Since @ide@ includes access to Liferay DXP, you must verify that you have rights to use it.

   Your credentials are not saved locally; they're saved as a token in the `~/.liferay` folder. The token is used by your @ide@'s Liferay Workspace if you ever decide to redownload a Liferay DXP bundle. Furthermore, the Liferay DXP bundle that was downloaded in your workspace is also copied to your `~/.liferay/bundles` folder, so if you decide to initialize another Liferay DXP instance of the same version, the bundle is not re-downloaded. See the Adding a Liferay Bundle to a Workspace for more information on this topic.

   Important: The token generator sometimes has issues generating a token for workspaces built behind a proxy. If you're unable to automatically generate a workspace token, you can generate one manually.

3. Run the installer. You may need to allow permission for the installer to run, depending on your operating system and where you want to install it.

4. Click *Next* to begin the installation process. Select the installation folder for your Liferay @ide@ instance. Then click *Next*.



Setup

Installation Directory

Please specify the directory where Liferay Developer Studio will be installed.

Installation Directory: Files (x86)\LiferayDeveloperStudio

Figure 52.1: Choose the folder your @ide@ instance should reside.

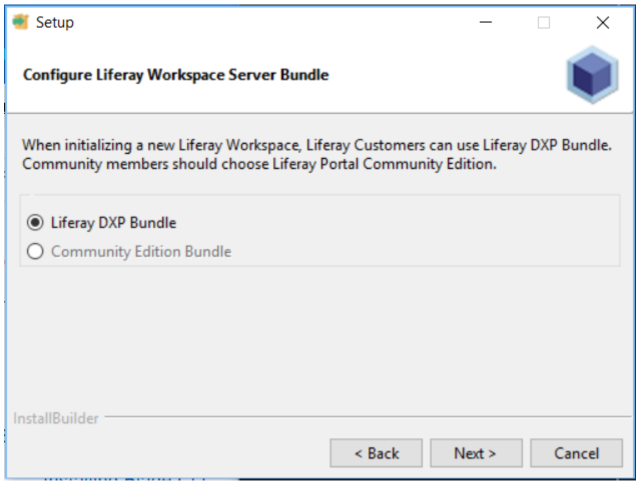
5. Liferay @ide@ provides Liferay Workspace by default, which is a developer environment used to build and manage Liferay DXP projects. The installer automatically installs Liferay Workspace and its dedicated command line tool (Blade CLI).

   You'll need to choose the Liferay bundle you plan to use in your Liferay Workspace: *Liferay DXP Bundle* or *Community Edition Bundle*. Then choose *Next*.

   If you selected *Liferay DXP Bundle*, you're also required to provide your liferay.com email and password.

6. Click *Next* to finish the installation process for your @ide@ instance.

Congratulations! You've installed Liferay @ide@! It's now available in the folder you specified. To run @ide@, execute the `DeveloperStudio` executable. A Liferay Workspace has also been initialized in that same folder. For more information on the Liferay Workspace installation related to this installation process, see the Installing Liferay Workspace section.



When initializing a new Liferay Workspace, Liferay Customers can use Liferay DXP Bundle. Community members should choose Liferay Portal Community Edition.

(●) Liferay DXP Bundle
( ) Community Edition Bundle

Figure 52.2: Choose the Liferay bundle you plan to use.

Install Liferay @ide@ into Eclipse Environment

To install @ide@ using an update URL, follow these steps:

1. In Eclipse, go to *Help* → *Install New Software...*.
2. In the *Work with* field, copy in the URL http://releases.liferay.com/tools/ide/latest/stable/.
3. You'll see the @ide@ components in the list below. Check them off and click *Next*.
4. Accept the terms of the agreements. Click *Next*, and @ide@ is installed. Like other Eclipse plugins, you must restart Eclipse to use them.

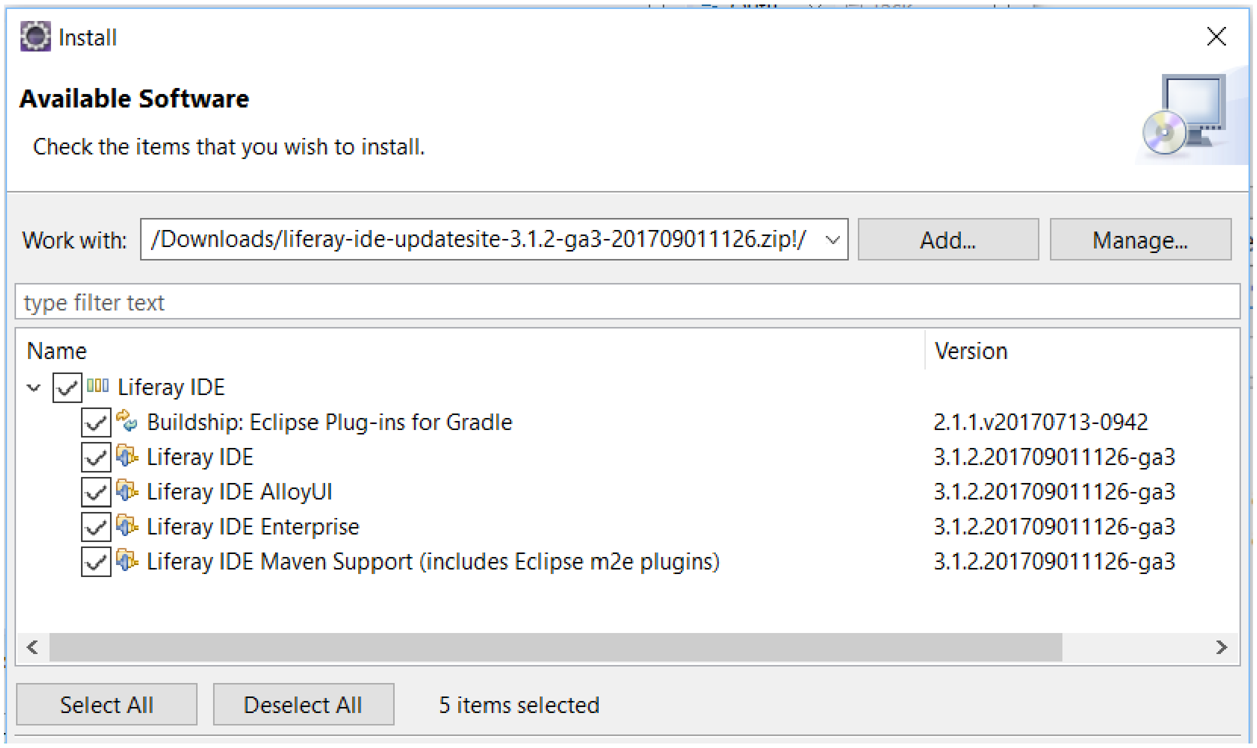
Liferay @ide@ is now installed in your existing Eclipse environment.

Install Liferay @ide@ into Eclipse from a ZIP File

To install @ide@ using a Zip file, follow these steps:

1. Go to the Liferay @ide@ downloads page. From the drop-down menu, select *Developer Studio Update Site Zip* and click *Download*.
2. In Eclipse, go to *Help* → *Install New Software...*.
3. In the *Add* dialog, click the *Archive* button and browse to the location of the downloaded Liferay @ide@ Update Site .zip file. Then press *OK*.
4. You'll see the @ide@ components in the list below. Check them off and click *Next*.
5. Accept the terms of the agreements and click *Next*, and Developer Studio is installed. Like other Eclipse plugins, you must restart Eclipse to use them.

Awesome! You've installed Liferay @ide@ in your existing Eclipse environment.


Install

Available Software

Check the items that you wish to install.

Work with: /Downloads/liferay-ide-updatesite-3.1.2-ga3-201709011126.zip!/

Name	Version
✓ Liferay IDE	
✓ Buildship: Eclipse Plug-ins for Gradle	2.1.1.v20170713-0942
✓ Liferay IDE	3.1.2.201709011126-ga3
✓ Liferay IDE AlloyUI	3.1.2.201709011126-ga3
✓ Liferay IDE Enterprise	3.1.2.201709011126-ga3
✓ Liferay IDE Maven Support (includes Eclipse m2e plugins)	3.1.2.201709011126-ga3

5 items selected

Figure 52.3: Make sure to check all the @ide@ components you wish to install.

Generating a Workspace Token Manually

If you run into any issues with generating your token automatically, you can follow the steps below to manually create one.

1. Navigate to www.liferay.com and log in to your account.
2. Click the Options button (☰) and select *Account Home*.
3. Select *Account Settings* from the left menu.
4. Click *Authorization Tokens* from the right menu under the Miscellaneous heading.
5. Select *Add Token*, give it a device name, and click *Generate*. The device name can be set to any string; it's for bookkeeping purposes only.
6. Create a file named `~/.liferay/token` and copy the generated token into that file.

   Make sure there are no new lines or white space in the file. It should only be one line.

You've successfully generated your token manually and it's now available for your installer to access. If you haven't run the installer, you can do so now. If you've already run the installer, you can set the DXP bundle to download in the `gradle.properties` file of your workspace. See the Adding a Liferay Bundle to a Workspace tutorial for details.

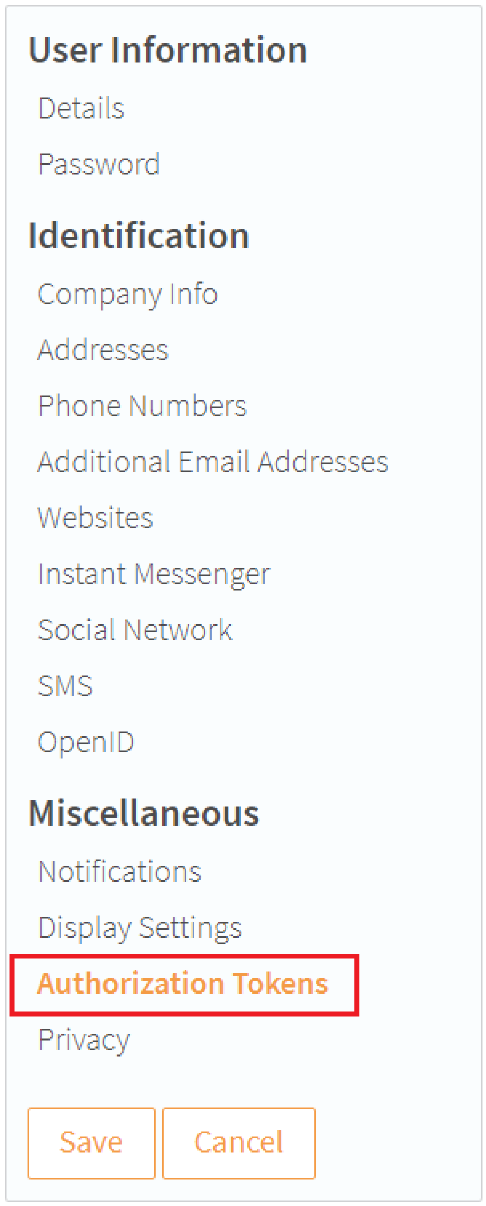
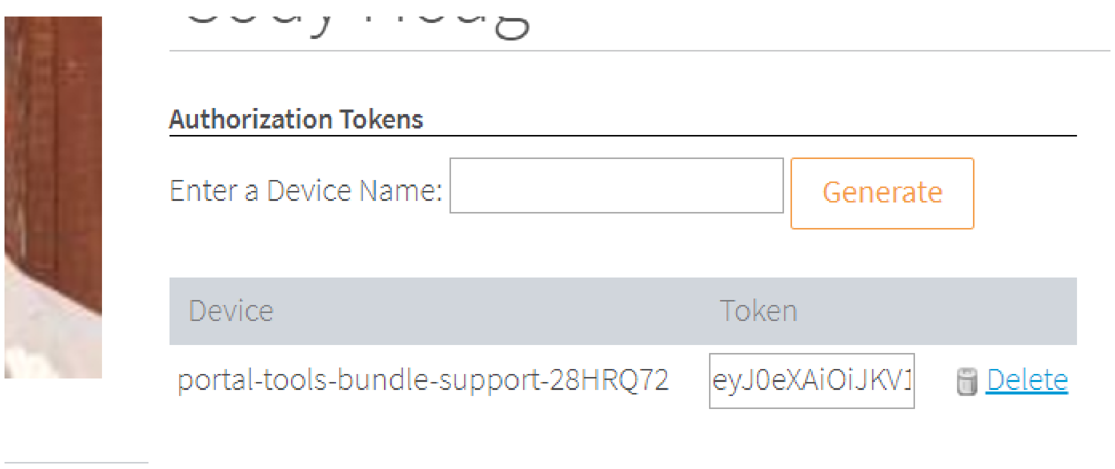


Figure 52.4: You can manually create your workspace token in the Authorization Tokens menu.


Authorization Tokens

Enter a Device Name:

Generate

Device	Token
portal-tools-bundle-support-28HRQ72	eyJ0eXAiOiJKV1

Delete

Figure 52.5: The generated token is available to copy.

52.2 Creating a Liferay Workspace with Liferay @ide@

In this tutorial, you'll learn how to generate a Liferay Workspace using Liferay @ide@, which runs on the Blade CLI behind the scenes. Liferay @ide@ gives you a graphical interface instead of the command prompt, which can streamline your workflow. To learn more about Liferay Workspaces, visit its dedicated tutorial section.

!PVideo Thumbnail

Before creating your Liferay Workspace, you should understand the available Liferay DXP perspectives specifically designed for Liferay DXP development. You'll notice in the Perspectives view the *Liferay Workspace* and *Liferay* perspectives. If you plan on using a Liferay Workspace for your Liferay DXP development, you should select the *Liferay Workspace* perspective. This offers Gradle related development tools that are helpful when using a Liferay workspace. The *Liferay* perspective is geared towards developers who are using Ant-based development tools such as the Plugins SDK.

To create a Liferay Workspace in @ide@, select *File* → *New* → *Liferay Workspace Project*.

Note: Creating or importing a Liferay Workspace in IDE leverages Gradle scripts provided by the Buildship plugin. When using @ide@ 3.1.x, you should be using Buildship 2.1.x. If you leverage higher versions of Buildship (e.g., 2.2.x), @ide@ cannot successfully create or import a workspace.

A New Liferay Workspace dialog appears, presenting several configuration options. Follow the instructions below to create your workspace.

1. Give your workspace a name.

2. Choose the location where you'd like your workspace to reside. Checking the *Use default location* checkbox places your Liferay Workspace in the Eclipse workspace you're working in.

3. Check the *Download Liferay bundle* checkbox if you'd like to auto-generate a Liferay instance in your workspace. You'll be prompted to name the server, if selected. This Liferay bundle is generated the same way as described in the previous section.

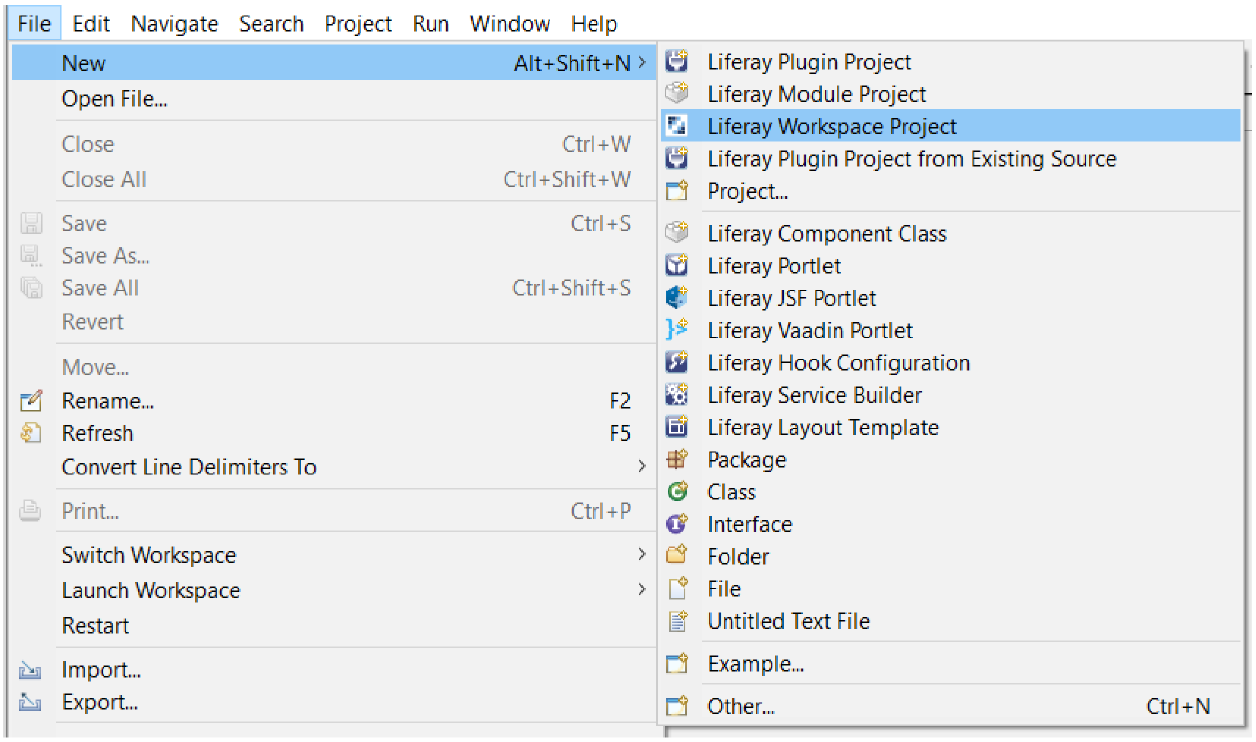


Figure 52.6: By selecting *Liferay Workspace*, you begin the process of creating a new workspace for your Liferay projects.

```
**Note:** If you'd like to configure a pre-existing Liferay bundle to your
workspace, you can create a directory for the bundle in your workspace and
configure it in the workspace's `gradle.properties` file by setting the
`liferay.workspace.home.dir` property.
```

---

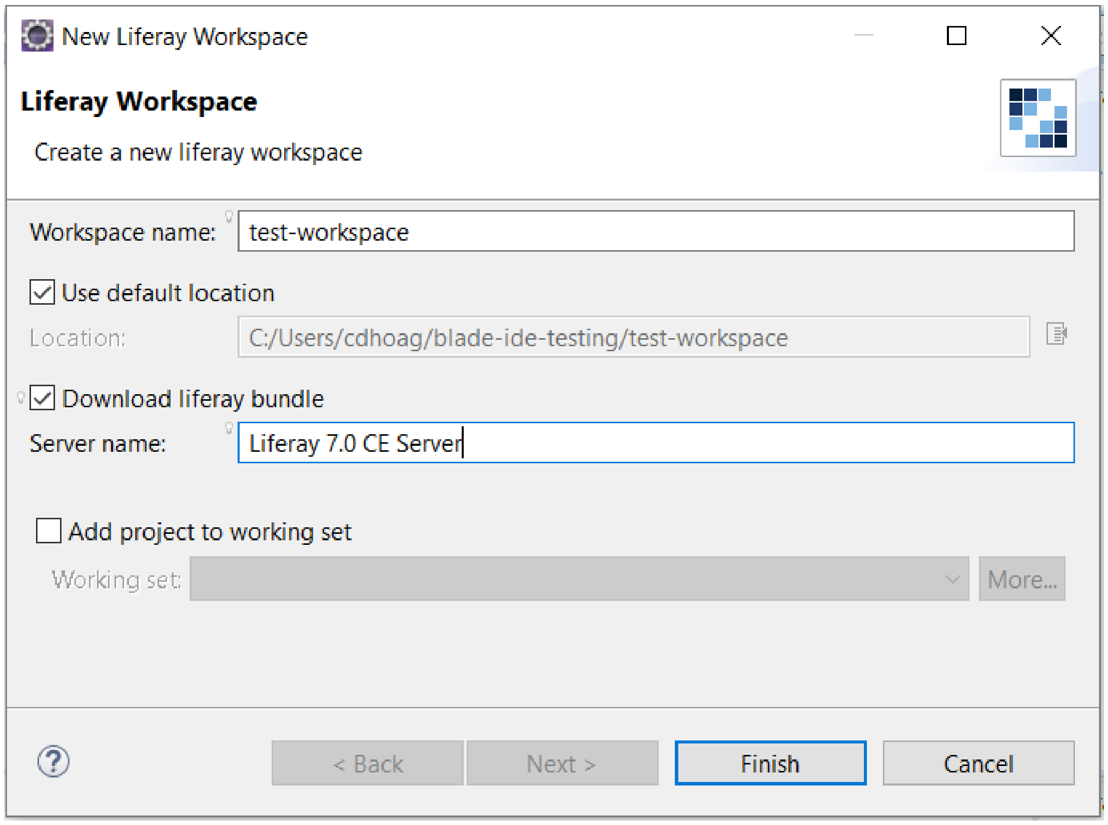
4. Check the *Add project to working set* checkbox if you'd like the workspace to be a part of a larger working set you've already created in @ide@. For more information on working sets, visit Eclipse Help.

5. Click *Finish* to create your Liferay Workspace.

A dialog appears prompting you to open the Liferay Workspace perspective. Click *Yes*, and your perspective will switch to Liferay Workspace.

Note: You can also create a Liferay Workspace during the initial start-up of a Liferay Developer Studio instance.

Awesome! You've successfully created a Liferay Workspace in Liferay @ide@! If you're using Liferay Developer Studio, you can also create a workspace during initial start-up.


New Liferay Workspace

Liferay Workspace

Create a new liferay workspace

Workspace name: test-workspace

☑ Use default location

Location: C:/Users/cdhoag/blade-ide-testing/test-workspace

☑ Download liferay bundle

Server name: Liferay 7.0 CE Server

☐ Add project to working set

Working set:

Figure 52.7: Liferay @ide@ provides an easy-to-follow menu to create your Liferay Workspace.

Liferay Workspace Settings in @ide@

The Liferay Workspace perspective is intended for Gradle development for 7.0 modules. Since Liferay Workspaces are used for Gradle based development and the Liferay Plugins perspective is intended for the Plugins SDK and Ant based development, the two perspectives are independent of each other.

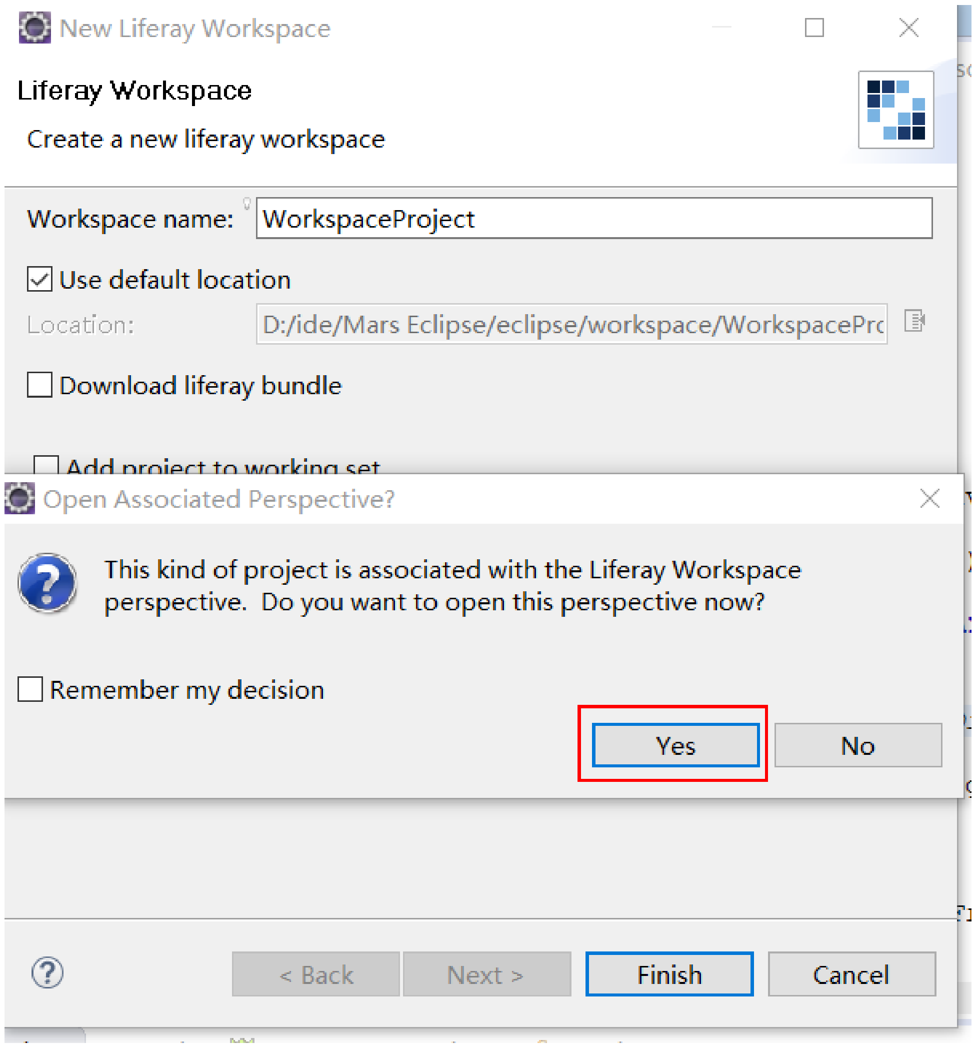
You'll find your new workspace in the Project Explorer and your Liferay server (if you created it) in the Servers menu. It's important to note that an Eclipse workspace can only have one Liferay Workspace project.

You can configure your workspace's module presentation by switching between the default *Hierarchical* or *Flat* views. To do this, navigate to the Project Explorer's *View Menu* (▽), select *Projects Presentation* and then select the presentation mode you'd like to display. The Hierarchical view displays subfolders and subprojects under the workspace project, whereas the Flat view displays the workspace's modules separately from the workspace.

If you've already created a Liferay Workspace and you'd like to import it into your existing @ide@, you can do so by navigating to *File* → *Import* → *Liferay* → *Liferay Workspace Project*. Then click *Next* and browse for your workspace project. Once you've selected you workspace, click *Finish*.

Congratulations! You've learned how to create and configure a Liferay Workspace using Liferay @ide@. Now that your workspace is created, you can begin creating Liferay plugins.

!VVideo Tutorial


New Liferay Workspace

Liferay Workspace

Create a new liferay workspace

Workspace name: WorkspaceProject

[✓] Use default location

Location: D:/ide/Mars Eclipse/eclipse/workspace/WorkspacePrc

[ ] Download liferay bundle

[ ] Add project to working set

Open Associated Perspective?

This kind of project is associated with the Liferay Workspace perspective. Do you want to open this perspective now?

[ ] Remember my decision

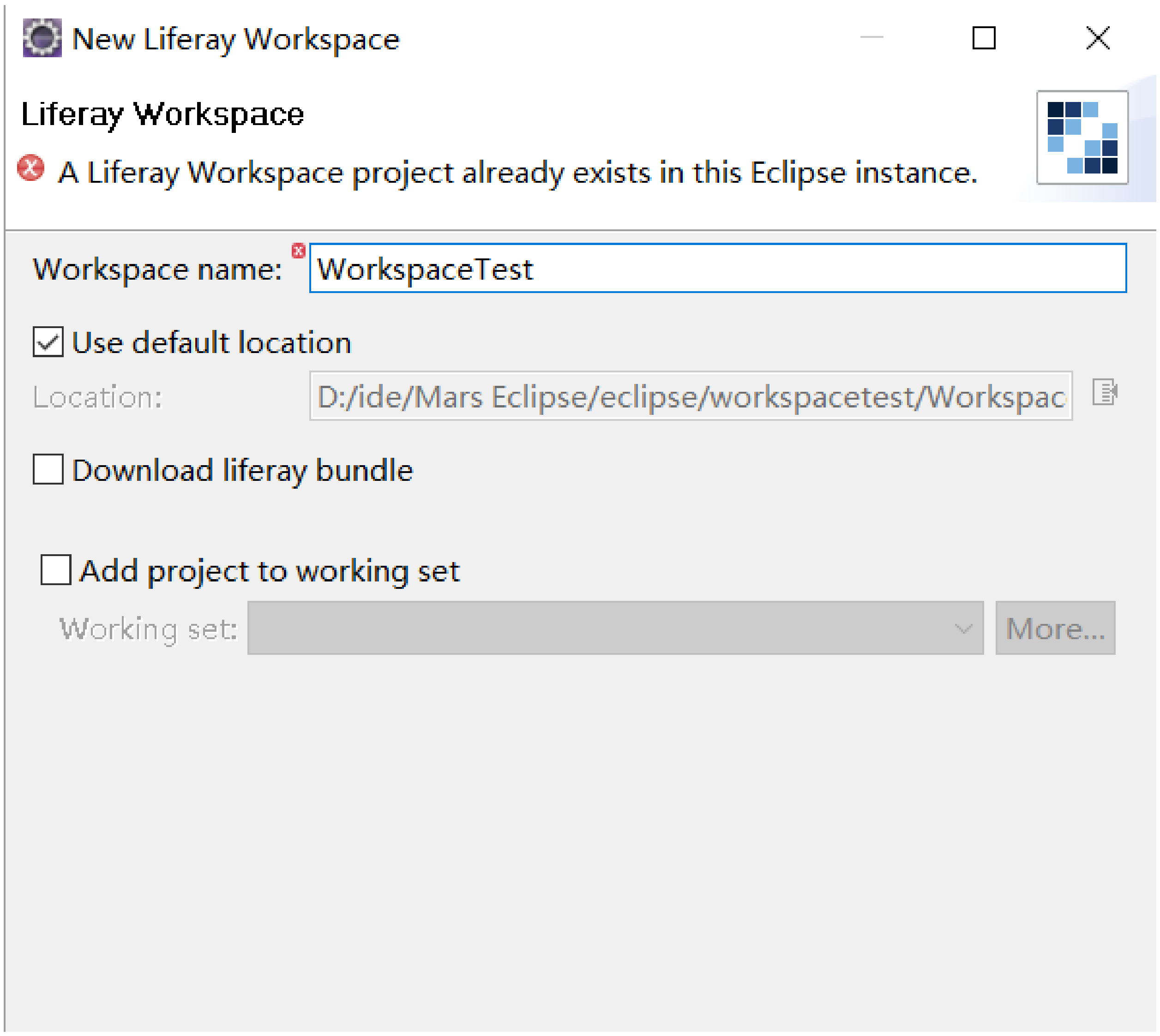
Figure 52.8: The Liferay Workspace perspective is preferred for 7.0 and OSGi module development.

52.3 Setting Proxy Requirements for Liferay @ide@

If you have proxy server requirements and want to configure your http(s) proxy
to work with Liferay @ide@, follow the instructions below.

1. Navigate to Eclipse's *Window* → *Preferences* → *General* → *Network Connections* menu.
2. Set the *Active Provider* drop-down selector to *Manual*.
3. Under *Proxy entries*, configure both proxy HTTP and HTTPS by clicking the field and selecting the *Edit* button.
4. For each schema (HTTP and HTTPS), enter your proxy server's host, port, and authentication settings (if necessary).

   Note: Do not leave whitespace at the end of your proxy host or port settings.
5. Once you've configured your proxy entry, click *OK* → *OK*.



New Liferay Workspace

Liferay Workspace

A Liferay Workspace project already exists in this Eclipse instance.

Workspace name: WorkspaceTest

[✓] Use default location

Location: D:/ide/Mars Eclipse/eclipse/workspacetest/Workspac

[ ] Download liferay bundle

[ ] Add project to working set

Working set:

Figure 52.9: An @ide@ workspace only supports one Liferay Workspace project. If you create another, you'll be given an error message.

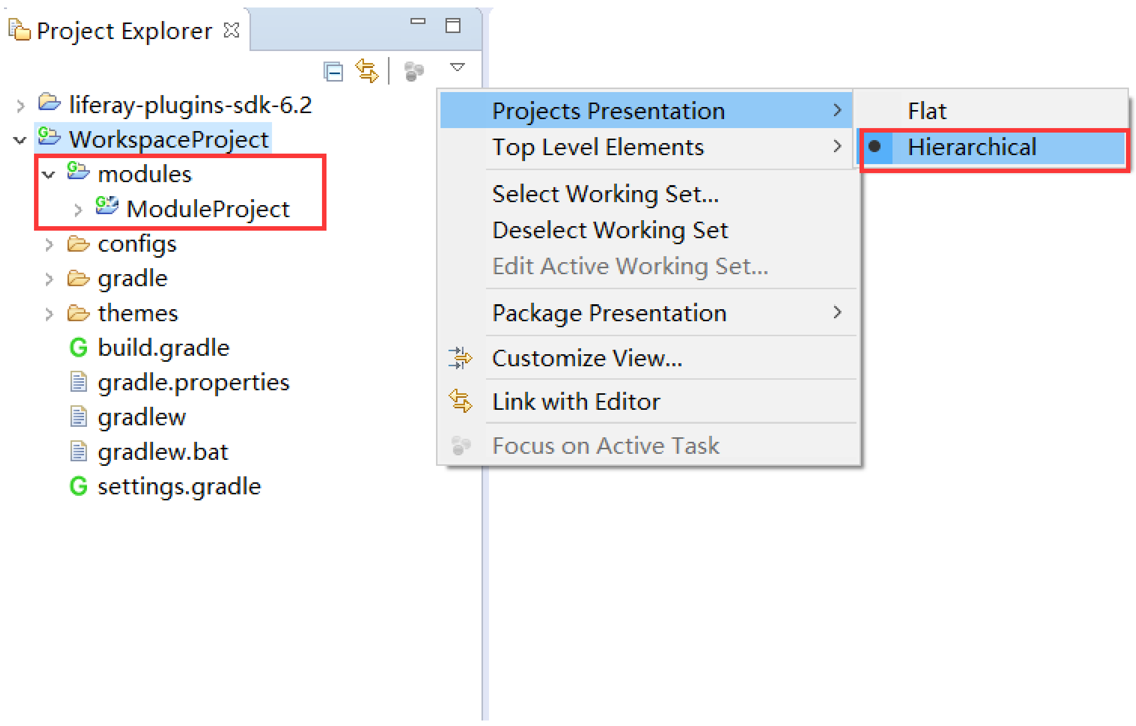


Figure 52.10: The Hierarchical project presentation mode is set, by default.

If you're working with a Liferay Workspace in @ide@, you'll need to configure your proxy settings for that environment too. See the Setting Proxy Requirements for Liferay Workspace for more details.

Awesome! You've successfully configured Liferay @ide@'s proxy settings!

Additional Proxy Settings

Some Eclipse plugins do not properly check the `core.net` proxy infrastructure when setting proxy settings via *Window* → *Preferences* → *General* → *Network Connections*. Therefore, you may need to configure additional proxy settings.

To do so, open the `eclipse.ini` file associated with your Eclipse installation and add the following entries:

```
-vmargs
-Dhttp.proxyHost=www.somehost.com
-Dhttp.proxyPort=1080
-Dhttp.proxyUser=userId
-Dhttp.proxyPassword=somePassword
-Dhttps.proxyHost=www.somehost.com
-Dhttps.proxyPort=1080
-Dhttps.proxyUser=userId
-Dhttps.proxyPassword=somePassword
```

After saving the file, restart Eclipse. Now your additional proxy settings are applied!

52.4 Updating Liferay @ide@

If you're already using Liferay @ide@ but need to update your environment, follow the steps below:

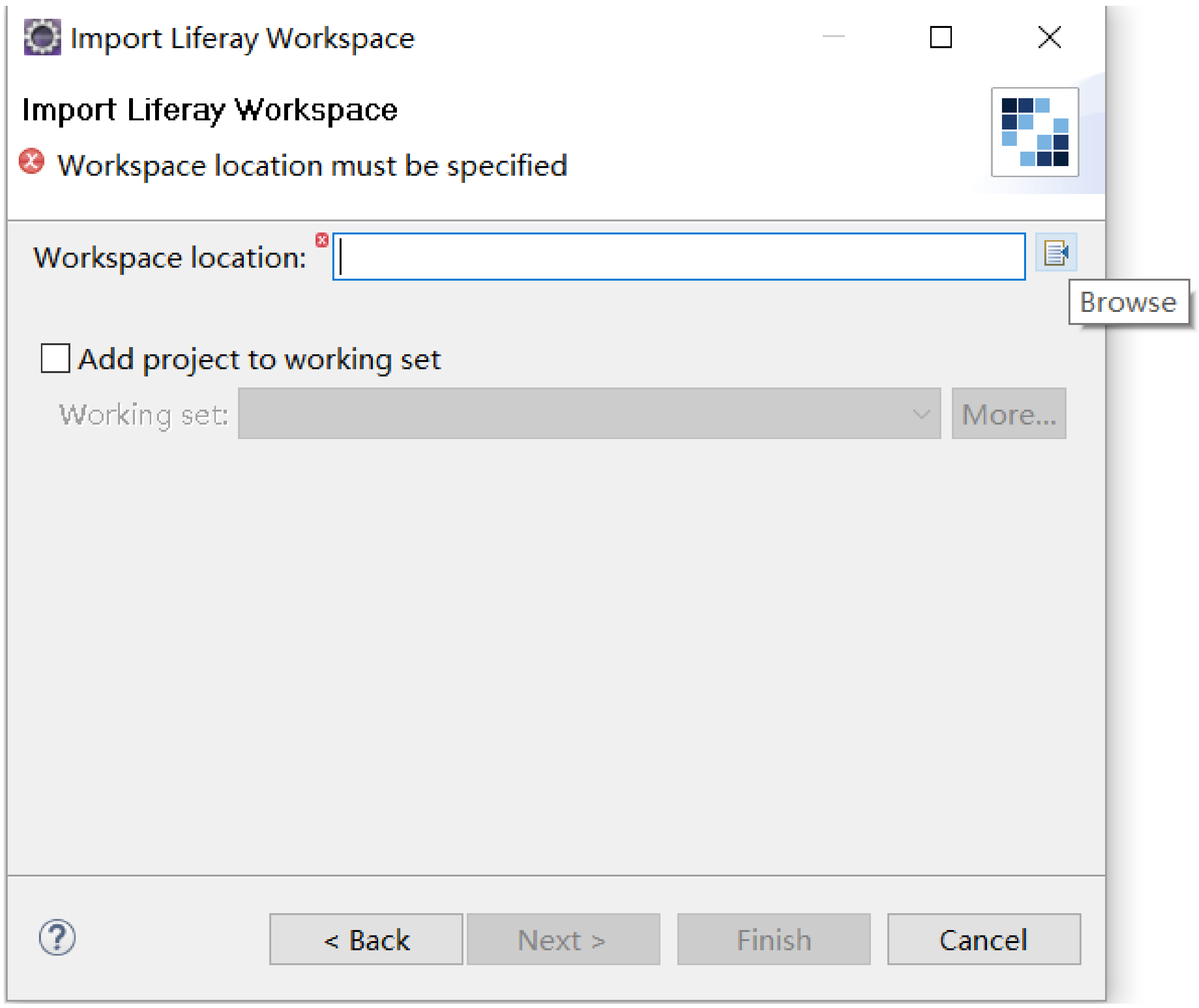


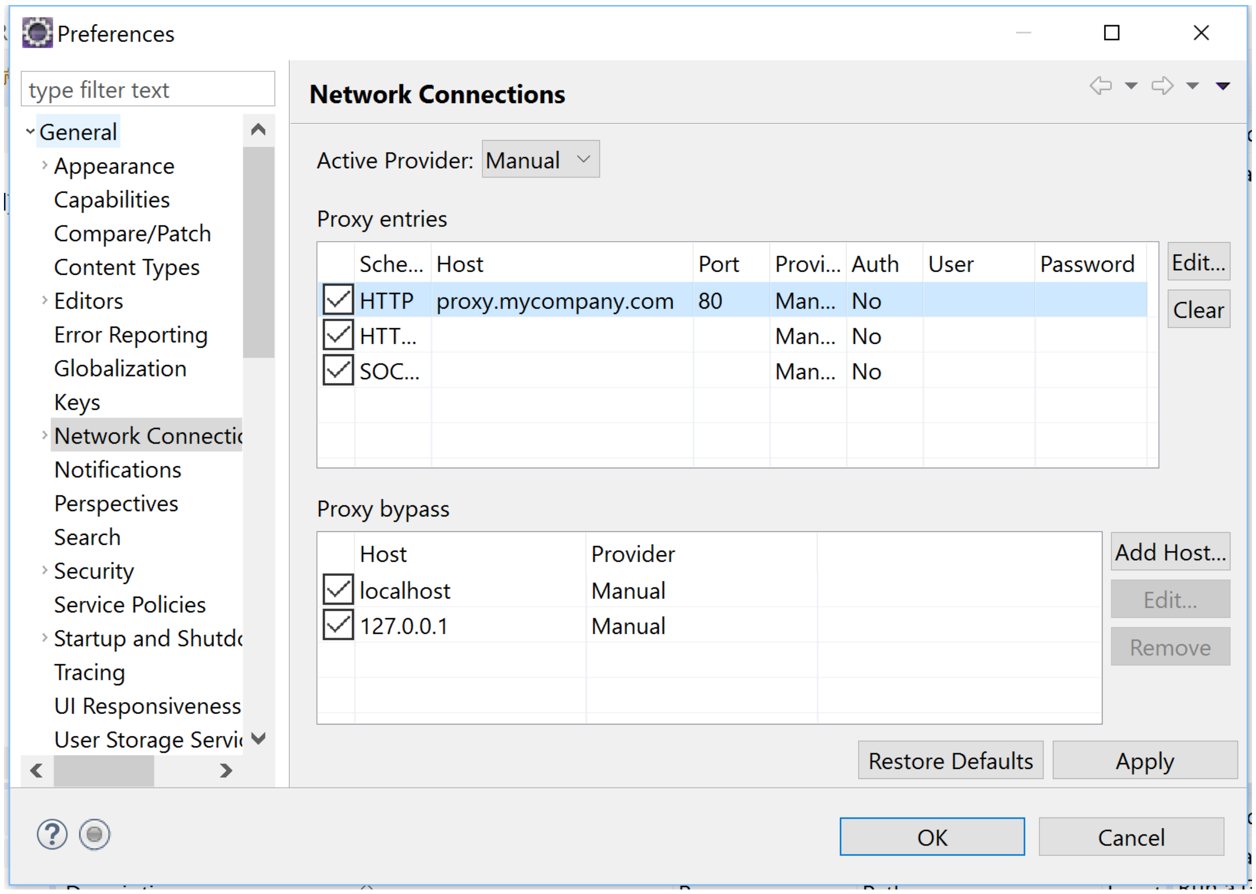
Figure 52.11: You can import an existing Liferay Workspace into your current @ide@ session.

1. In Liferay @ide@, go to *Help* → *Install New Software...*.
2. In the *Work with* field, copy in the URL http://releases.liferay.com/tools/ide/latest/stable/.
3. You'll see the @ide@ components in the list below. Check them off and click *Next*.
4. Accept the terms of the agreements. Click *Next*, and @ide@ is updated. You must restart @ide@ for the updates to take effect.

You're now on the latest version of Liferay @ide@!

52.5 Creating Modules with Liferay @ide@

@ide@ provides a Module Project Wizard for users to create a variety of different module projects. You can create a new Liferay module project by navigating to *File* → *New* → *Liferay Module Project*.



Network Connections

Active Provider: Manual

Proxy entries

	Sche...	Host	Port	Provi...	Auth	User	Password
✓	HTTP	proxy.mycompany.com	80	Man...	No		
✓	HTT...			Man...	No		
✓	SOC...			Man...	No		

Proxy bypass

	Host	Provider
✓	localhost	Manual
✓	127.0.0.1	Manual

Figure 52.12: You can configure your proxy settings in @ide@'s Network Connections menu.

You're given options for project name, location, build type, and template type. You can build your project using Gradle or Maven. If you're unsure for which template type to choose, see the Project Templates reference section. Click *Next* and you're given additional configuration options for a component class.

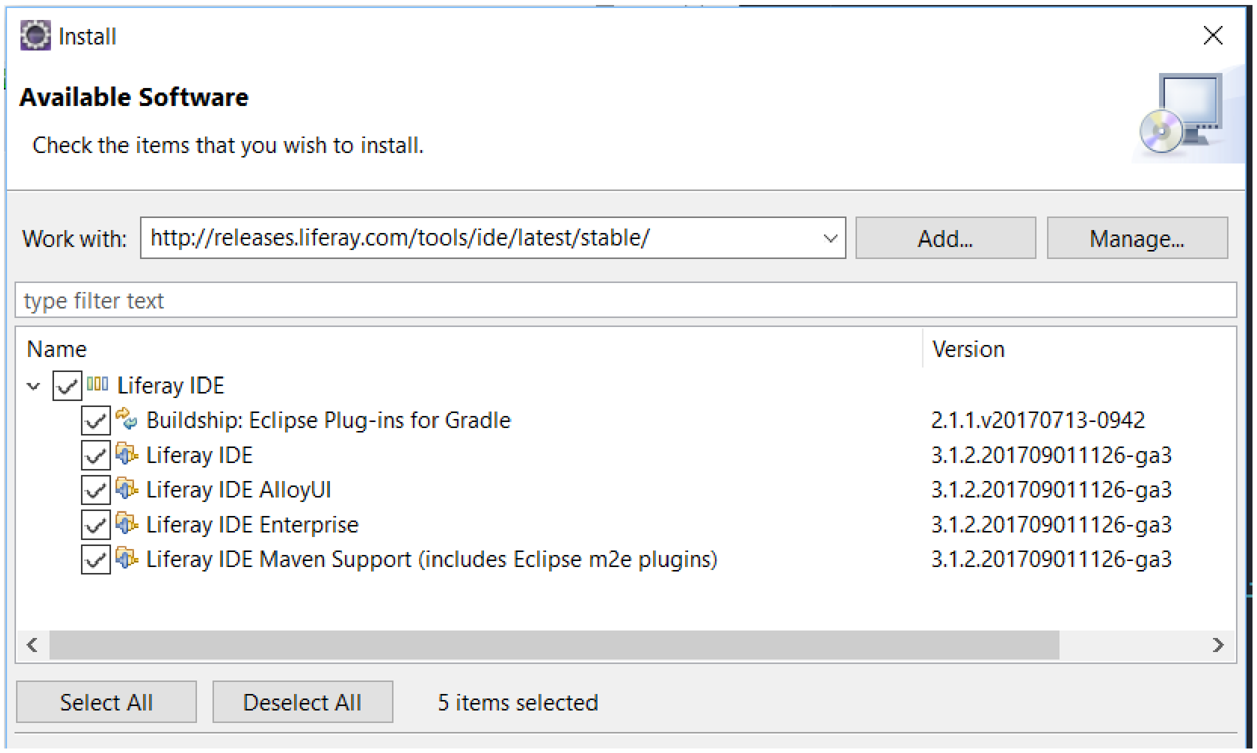
You can specify your component class's name, package name, and its properties. The properties you assign are the ones found in the `@Component` annotation's `property = {...}` assignment.

Once you've configured your module project's component class, click *Finish* to create your project.

Creating Component Classes

You can also create a new component class for a pre-existing module project. Navigate to *File* → *New* → *Liferay Component Class*. This is a similar wizard to the previous component class wizard, except you can select a component class template. There are many templates in the `Component Class Template` list:

- *Auth Failure*: processes a verify login failure
- *Auth Max Failure*: processes maximum number of login failures
- *Authenticator*: authenticates processing
- *Friendly URL Mapper*: processes Friendly URLs
- *GOGO Command*: creates a custom Gogo command
- *Indexer Post Processor*: creates a new Indexer Post Processor
- *Login Pre Action*: creates a login pre action


Install

Available Software

Check the items that you wish to install.

Work with: http://releases.liferay.com/tools/ide/latest/stable/

Name	Version
✓ Liferay IDE	
✓ Buildship: Eclipse Plug-ins for Gradle	2.1.1.v20170713-0942
✓ Liferay IDE	3.1.2.201709011126-ga3
✓ Liferay IDE AlloyUI	3.1.2.201709011126-ga3
✓ Liferay IDE Enterprise	3.1.2.201709011126-ga3
✓ Liferay IDE Maven Support (includes Eclipse m2e plugins)	3.1.2.201709011126-ga3

5 items selected

Figure 52.13: Make sure to check all the @ide@ components you wish to install.

- *Model Listener*: sets a model listener
- *Poller Processor*: creates a new poller processor
- *Portlet*: creates a new portlet class file
- *Portlet Action Command*: creates a new portlet action command
- *Portlet Filter*: creates a new portlet filter
- *Rest*: calls and wraps inner service in the way of Rest
- *Service Wrapper*: creates a new service wrapper
- *Struts in Action*: creates a new struts action
- *Struts Portlet Action*: creates a new struts portlet action

Possible Dependency Issues

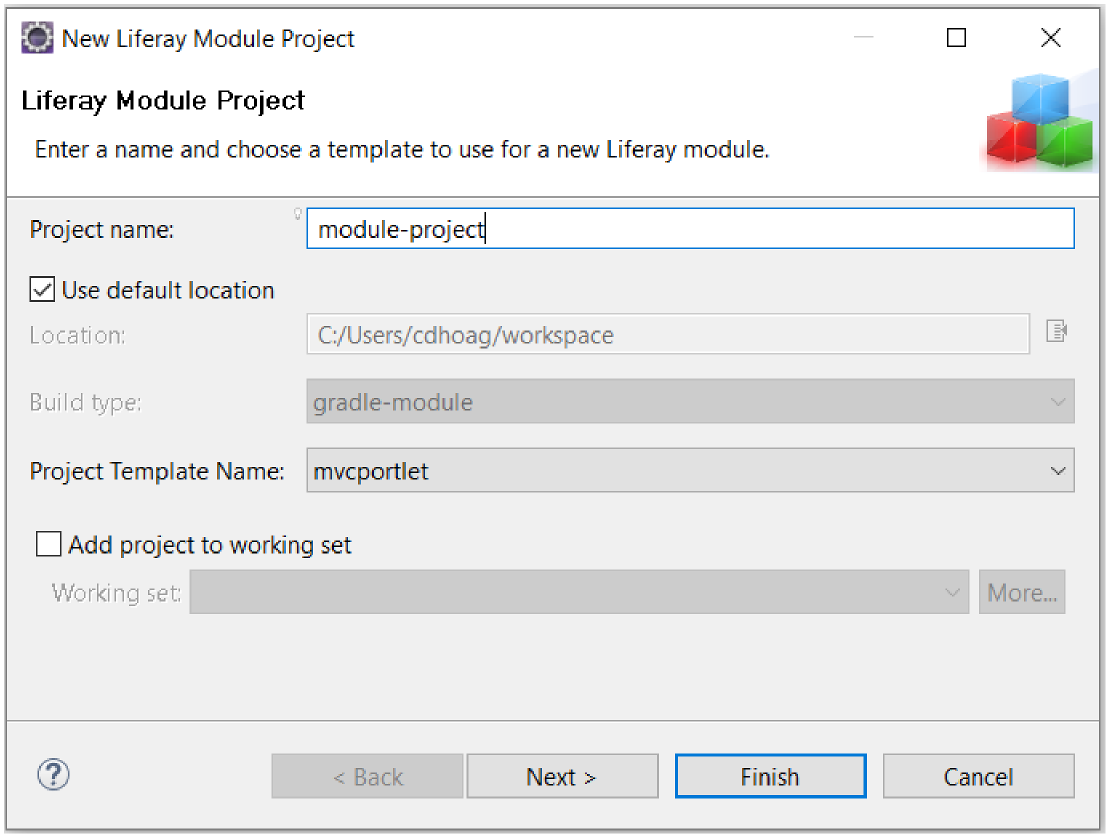
When selecting the `Authenticator`, `Portlet Action Command`, `Rest`, or `Service Wrapper` templates, you may run into some dependency issues that could cause errors in your project. There is a set of steps outlined below that you should follow, with sub-steps for each of the four templates that could cause problems.

1. Open the module project's `build.gradle` file.

2. Check whether the appropriate dependencies exist. These are outlined below.

3. Right-click your project and select *Gradle* → *Refresh Gradle Project*.

4. If you're using the *Portlet Action Command* template, you'll also need to change the component class declaration from implementing the `FreeMarkerPortlet` class to extending it. For instance, your `*Portlet` component class should have the following declared:


New Liferay Module Project

Liferay Module Project

Enter a name and choose a template to use for a new Liferay module.

Project name: module-project

☑ Use default location

Location: C:/Users/cdhoag/workspace

Build type: gradle-module

Project Template Name: mvcportlet

☐ Add project to working set

Working set:

Figure 52.14: When selecting *New → Liferay Module Project*, a Module Project Wizard appears.

YourPortletClass extends FreeMarkerPortlet

The dependencies to check for when using each template are outlined below:
Authenticator

- `compile com.liferay.portal:com.liferay.portal.kernel:VERSION`
- `compile org.osgi:org.osgi.service.component.annotations:VERSION`

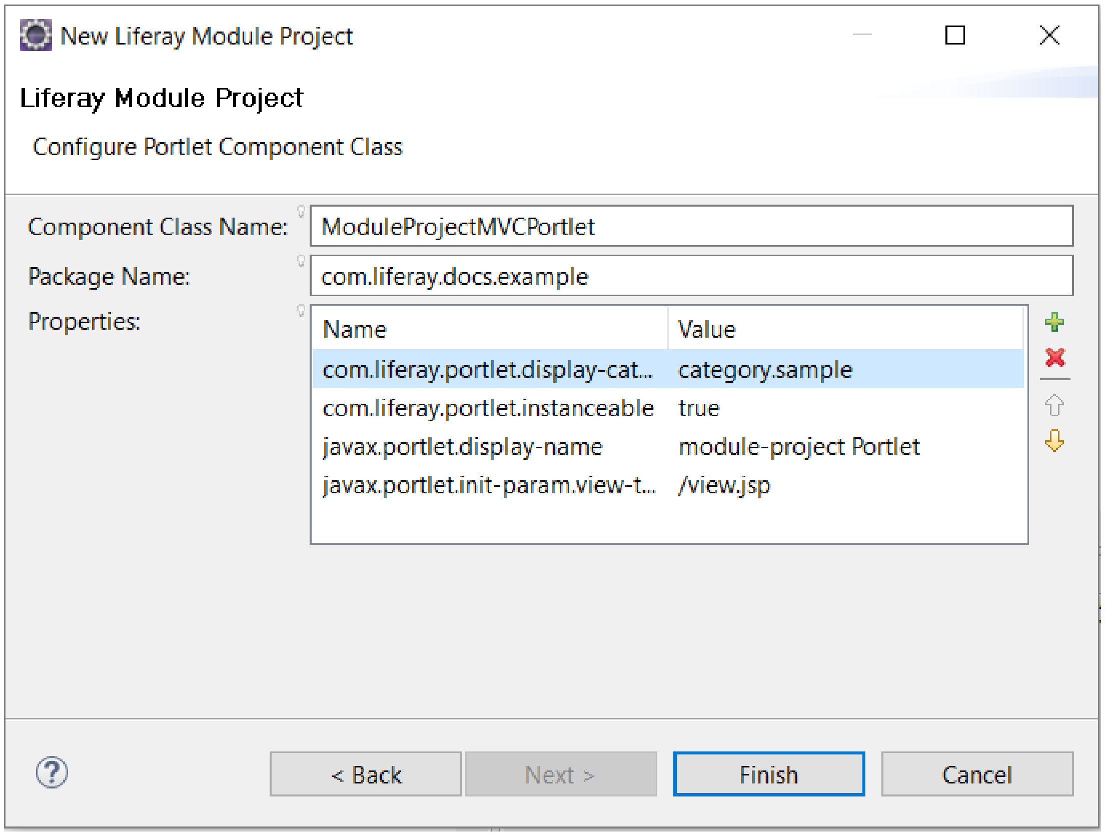
Portlet Action Command

- `compile javax.portlet:portlet-api:VERSION`
- `compile javax.servlet:javax.servlet-api:VERSION`
- `compile org.osgi:org.osgi.service.component.annotations:VERSION`
- `compile com.liferay.portal:com.liferay.portal.kernel:VERSION`
- `compile com.liferay.portal:com.liferay.util.bridges:VERSION`
- `compile com.liferay.portal:com.liferay.util.taglib:VERSION`

Rest

- `compile javax.ws.rs:javax.ws.rs-api:VERSION`

Service Wrapper


New Liferay Module Project

Liferay Module Project

Configure Portlet Component Class

Component Class Name: ModuleProjectMVCPortlet

Package Name: com.liferay.docs.example

Properties:

Name	Value
com.liferay.portlet.display-cat...	category.sample
com.liferay.portlet.instanceable	true
javax.portlet.display-name	module-project Portlet
javax.portlet.init-param.view-t...	/view.jsp

Figure 52.15: Specify your component class's details in the Portlet Component Class Wizard.

- The service wrapper class being used. For example, if you're using the `BookmarksEntryLocalServiceWrapper`, the following dependency would be required:

```
compile com.liferay:com.liferay.bookmarks.api:VERSION
```

Make sure the replace the `VERSION` text with the appropriate version for each specified dependency.

Once you've created your module project, you can configure your project's presentation in the @ide@'s Project Explorer. To change the project's presentation, select the default *Hierarchical* or *Flat* views. To do this, navigate to the Project Explorer's *View Menu* (▽), select *Projects Presentation* and then select the presentation mode you'd like to display. The Hierarchical view displays subfolders and subprojects under the project, whereas the Flat view displays the modules separately from their project.

@ide@ also provides a method to import existing module projects. You can import a module project by navigating to *File* → *Import* → *Liferay* → *Liferay Module Project(s)*. Then point to the project location and click *Finish*.

You now have the knowledge to create a Liferay module project from Liferay @ide@. Now go out there and get stuff done!

52.6 Creating Themes with Liferay @ide@

Liferay @ide@ lets you create and configure Liferay theme projects. You can create a standalone theme or in a Liferay Workspace. You can even create a Gradle or Maven based theme! Read on to learn more about creating themes in @ide@.

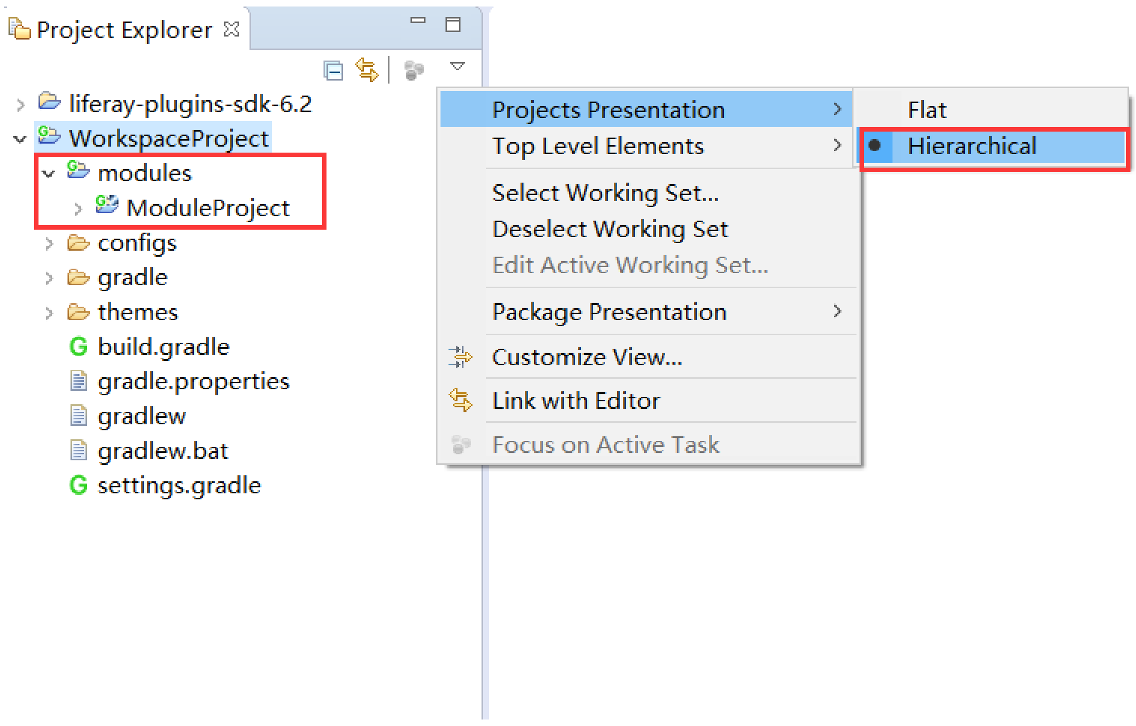


Figure 52.16: The Hierarchical project presentation mode is set, by default.

1. In @ide@, navigate to *File* → *New* → *Liferay Module Project*.

2. In the New Liferay Module Project wizard, give your project a name and select the *theme* project template. Also choose your theme's build type by selecting either *gradle-module* or *maven-module*.

3. Select *Finish*.

That's it! You've created a theme project in @ide@!

If you've configured a Liferay Workspace in your @ide@ instance, your theme is available in the workspace's `wars` folder by default. If you don't have a workspace configured in @ide@, it's available in the root of @ide@'s Project Explorer.

Note that themes created in @ide@ follow a WAR-style layout. This is the default layout of themes in 7.0. Although the wizard can be misleading by calling the theme a new module project, it is a WAR.

To modify a theme created in @ide@, mirror the folder structure of the files you wish to change and copy them into your theme's `webapp` folder.

Under the hood, @ide@ is using the theme project template. If you're interested in creating Liferay themes using the Liferay Theme Generator, see its dedicated tutorial. For more general information on Liferay themes, visit their dedicated tutorial section Themes and Layout Templates.

52.7 Deploying Modules with Liferay @ide@

Deploying modules in Liferay @ide@ is a cinch. Before deploying your module, make sure you have a Liferay server configured in @ide@. To see how to do this, see the Installing a Server in Liferay @ide@

There are two ways to deploy a module to your Liferay server. You should start your Liferay server before attempting to deploy to it.

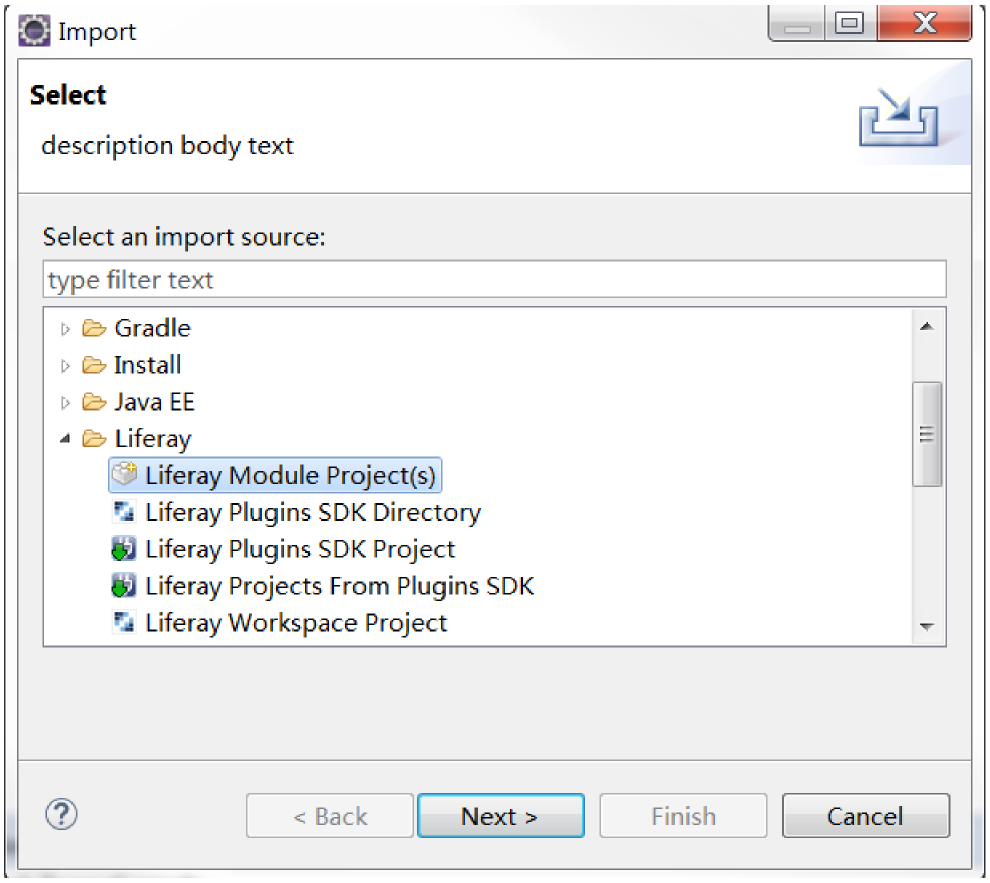

Import
Select
description body text
Select an import source:
type filter text
Gradle
Install
Java EE
Liferay
Liferay Module Project(s)
Liferay Plugins SDK Directory
Liferay Plugins SDK Project
Liferay Projects From Plugins SDK
Liferay Workspace Project
< Back
Next >
Finish
Cancel

Figure 52.17: Select the *Liferay Module Project(s)* to import a module project.

1. Select the module from the Package Explorer window and drag it to your Liferay server in the Servers window.

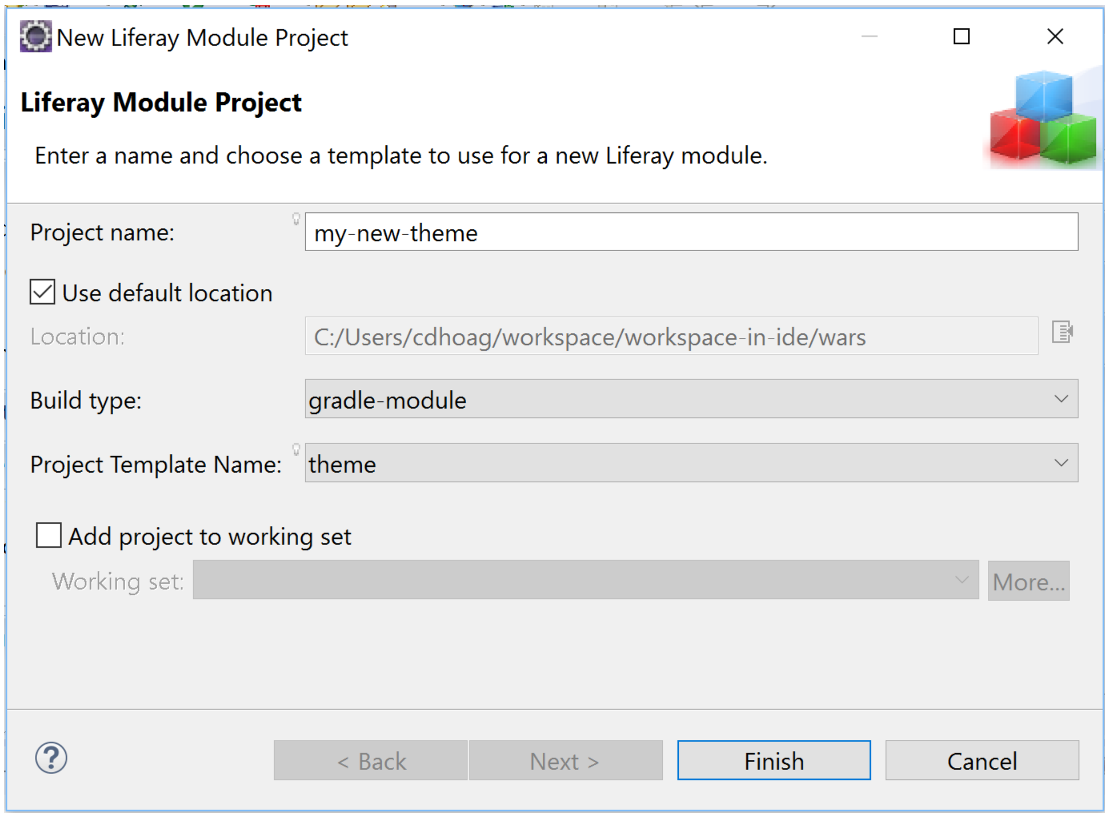
2. Right-click the server from the Servers window and select *Add and Remove....* Add the module(s) you'd like to deploy from the Available window to the Configured window. Then click *Finish*.

Note: For a legacy Maven application, you were able to deploy it by right-clicking it in the Package Explorer and selecting *Liferay* → *Maven* → *liferay:deploy*. This is no longer possible because Liferay's Maven archetypes no longer rely on the legacy `liferay-maven-plugin`. To deploy Maven projects in @ide@, make sure to follow the methods described above.

That's it! Once your module is deployed to the Liferay server, you can verify its installation in @ide@'s Console window.

52.8 Managing Module Projects with Liferay @ide@

Liferay @ide@ provides the ability to manage Liferay module projects from a GUI. Before you begin learning about managing your modules from Liferay @ide@, you should make sure a Liferay server is configured in your Eclipse workspace so you can deploy and run your projects. You can learn how to create a Liferay bundle and link it to your Liferay workspace in the Creating a Liferay Workspace with Liferay @ide@ tutorial.


New Liferay Module Project

Liferay Module Project

Enter a name and choose a template to use for a new Liferay module.

Project name: my-new-theme

[✓] Use default location

Location: C:/Users/cdhoag/workspace/workspace-in-ide/wars

Build type: gradle-module

Project Template Name: theme

[ ] Add project to working set

Working set:

Figure 52.18: Use the theme project template to create a Liferay theme in @ide@.

Once you've created modules, you can deploy them using Liferay @ide@. First, make sure your Liferay server is started by clicking the *Start Server* button (▶). Then navigate to your module project from the Project Explorer and drag-and-drop the project onto the configured Liferay bundle in the *Servers* menu. If at any time you'd like to stop your Liferay server, click the *Stop Server* button (■). Awesome! You've deployed a module to your running Liferay instance!

For the deployed module project, you can check if it has been deployed successfully by using Gogo Shell. Right-click the started portal in server view and select *Open Gogo Shell*.

A Gogo shell terminal appears, allowing you to enter Gogo commands to inspect your Liferay instance and the modules deployed to it. Enter the `lb` command to view a list of deployed modules. If the project status is active, then it deployed successfully.

Since the Liferay Workspace perspective in @ide@ is Gradle-based, you have some additional Gradle features you can take advantage of. The Gradle Tasks toolbar presents Gradle commands for your workspace that you can execute with a click of the mouse.

You can also access various Gradle build operations intended for Liferay module projects. Right-click your module project and select *Liferay* → *Gradle* and then the build command you want to execute.

To learn more about Gradle development in Liferay @ide@, see the Using Gradle in Liferay @ide@ tutorial.

Excellent! You've learned how to manage your Gradle-based Liferay Workspace using Liferay @ide@.

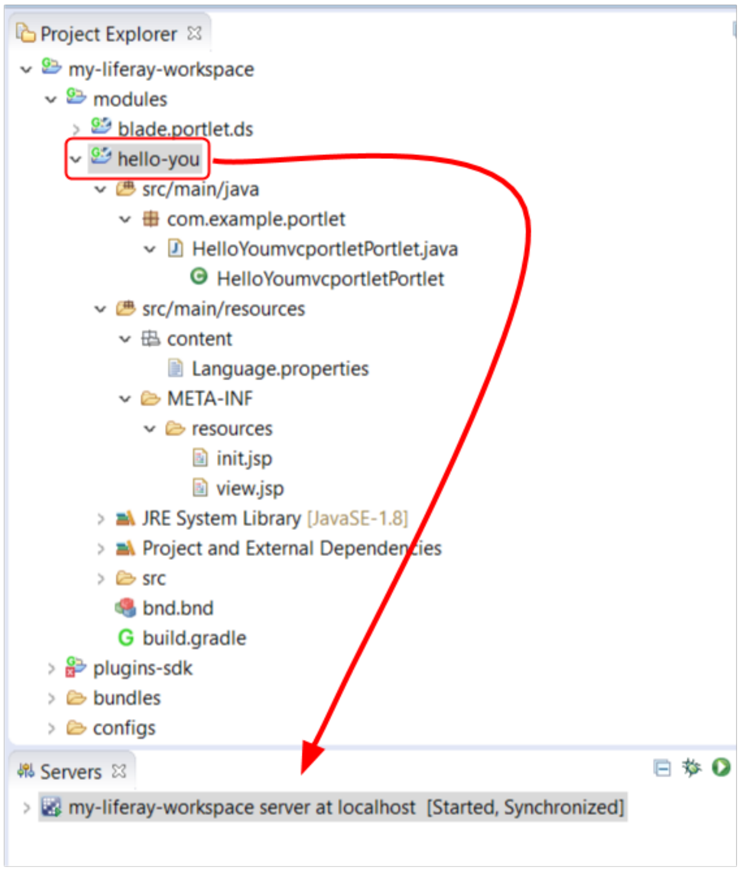


Figure 52.19: You can use the drag-and-drop method to deploy your module to Liferay DXP.

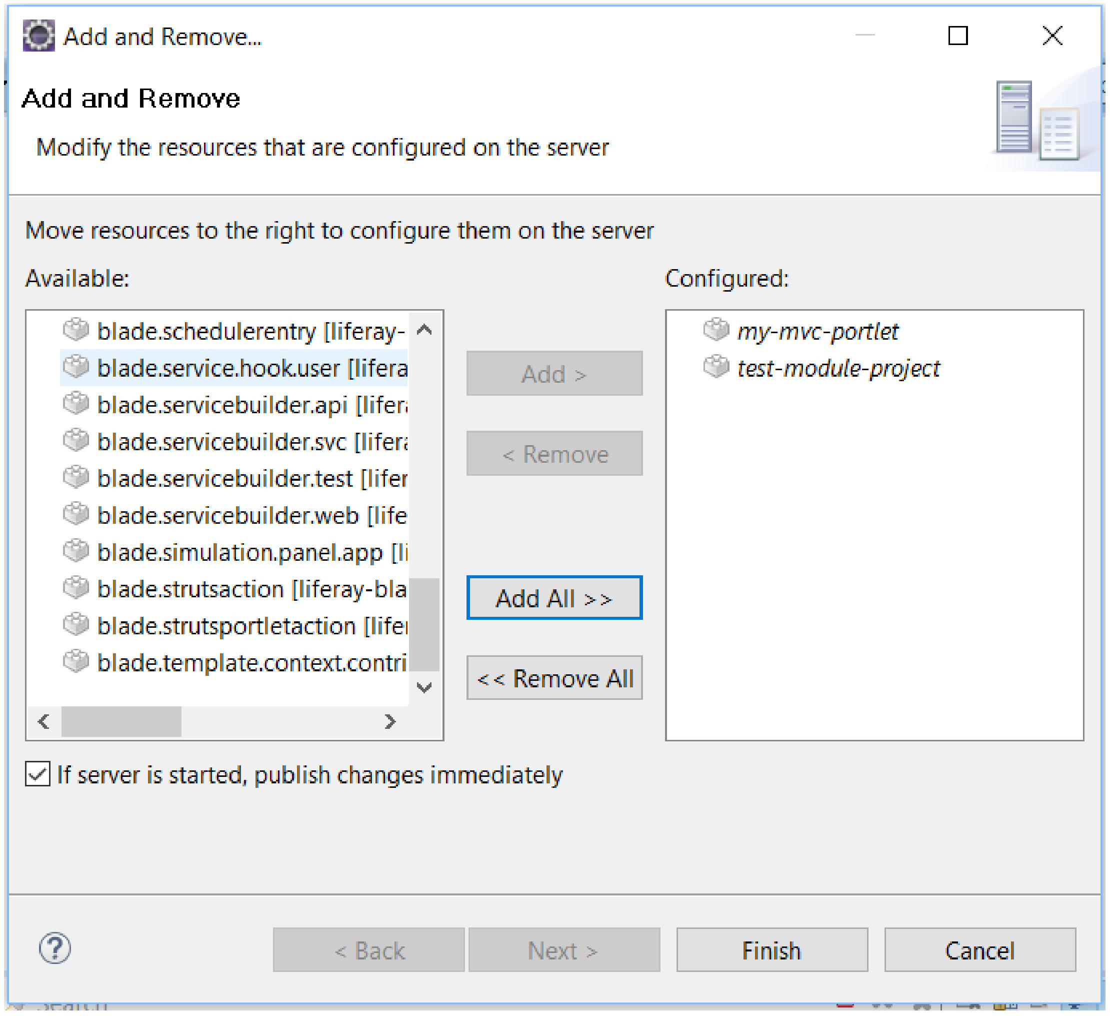
52.9 Installing a Server in Liferay @ide@

Installing a server in Liferay @ide@ is easy. In just a few steps you'll have your server up and running. Follow these steps to install your server:

1. In the Servers view, click the *No Servers are available* link. If you already have a server installed, you can install a new server by right-clicking in the Servers view and selecting *New* → *Server*. This brings up a wizard that walks you through the process of defining a new server.

2. Select the type of server you would like to create from the list of available options. For a standard server, open the *Liferay, Inc.* folder and select the *Liferay 7.x* option. You can change the server name to something more unique too; this is the name displayed in the Servers view. Then click *Next*. If you're creating a server for the first time, continue to the next step.

   Note: If you've already configured previous Liferay servers, you'll be provided the *Server runtime environment* field, which lets you choose previously configured runtime environments. If you want to re-add an existing server, select one from the dropdown menu. You can also add a new server by selecting *Add*, or you can edit existing servers by selecting *Configure runtime environments*. Once you've configured the


Add and Remove...

Add and Remove

Modify the resources that are configured on the server

Move resources to the right to configure them on the server

Available:

- blade.schedulerentry [liferay-
- blade.service.hook.user [lifera
- blade.servicebuilder.api [lifera
- blade.servicebuilder.svc [lifera
- blade.servicebuilder.test [lifer
- blade.servicebuilder.web [life
- blade.simulation.panel.app [li
- blade.strutsaction [liferay-bla
- blade.strutsportletaction [lifer
- blade.template.context.contri

Configured:

- *my-mvc-portlet*
- *test-module-project*

☑ If server is started, publish changes immediately

Figure 52.20: Using the this deployment method is convenient when deploying multiple module projects.

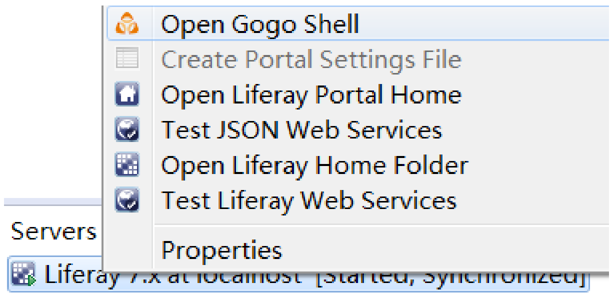
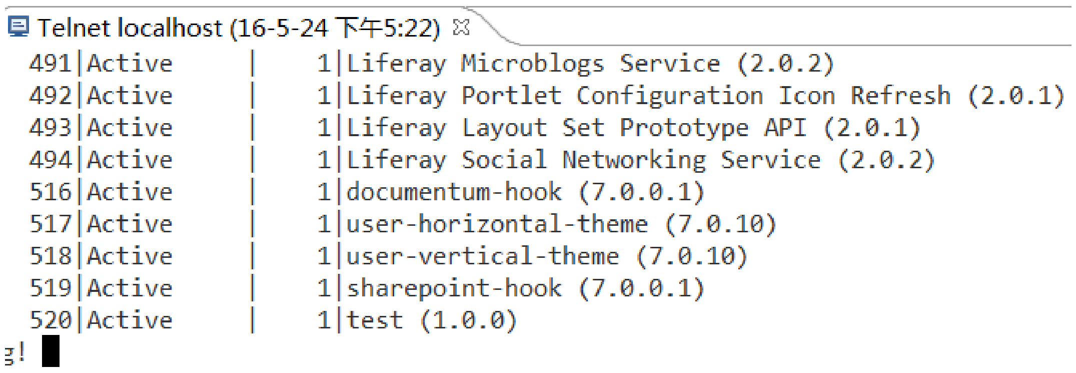
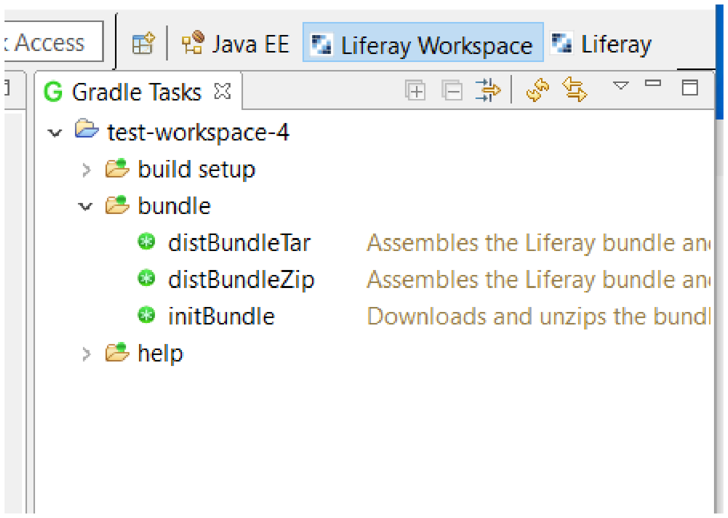


Figure 52.21: Select *Open Gogo Shell* to open a terminal window in @ide@ using Gogo shell.



```
Telnet localhost (16-5-24 下午5:22)
 491|Active      |    1|Liferay Microblogs Service (2.0.2)
 492|Active      |    1|Liferay Portlet Configuration Icon Refresh (2.0.1)
 493|Active      |    1|Liferay Layout Set Prototype API (2.0.1)
 494|Active      |    1|Liferay Social Networking Service (2.0.2)
 516|Active      |    1|documentum-hook (7.0.0.1)
 517|Active      |    1|user-horizontal-theme (7.0.10)
 518|Active      |    1|user-vertical-theme (7.0.10)
 519|Active      |    1|sharepoint-hook (7.0.0.1)
 520|Active      |    1|test (1.0.0)
g!
```

Figure 52.22: You can check to see if your module deployed successfully to Liferay using the Gogo shell.



Assembles the Liferay bundle an
Assembles the Liferay bundle an
Downloads and unzips the bundl

Figure 52.23: The Gradle Task toolbar offers Gradle tasks and their descriptions, which can be executed by double-clicking them.

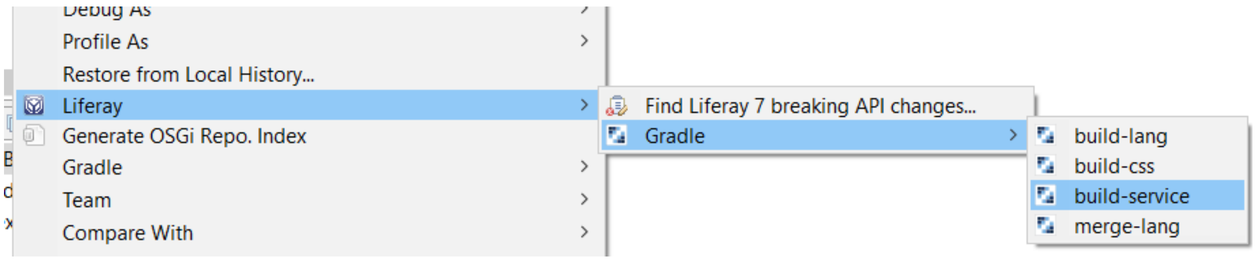
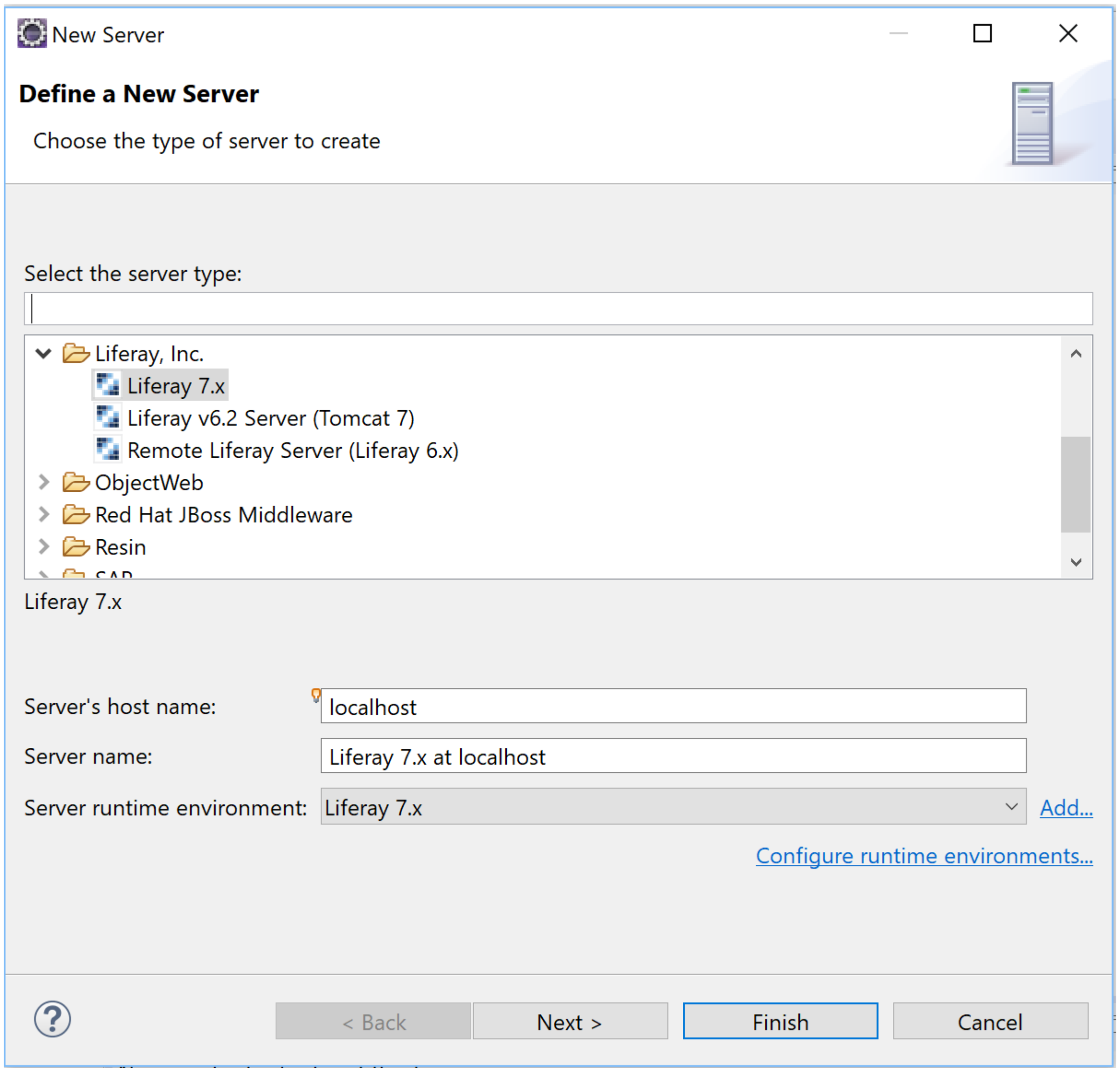


Figure 52.24: You can execute build operations by right-clicking the Gradle project in the Project Explorer.



New Server

Define a New Server

Choose the type of server to create

Select the server type:

- Liferay, Inc.
  - Liferay 7.x
  - Liferay v6.2 Server (Tomcat 7)
  - Remote Liferay Server (Liferay 6.x)
- ObjectWeb
- Red Hat JBoss Middleware
- Resin

Liferay 7.x

Server's host name: localhost

Server name: Liferay 7.x at localhost

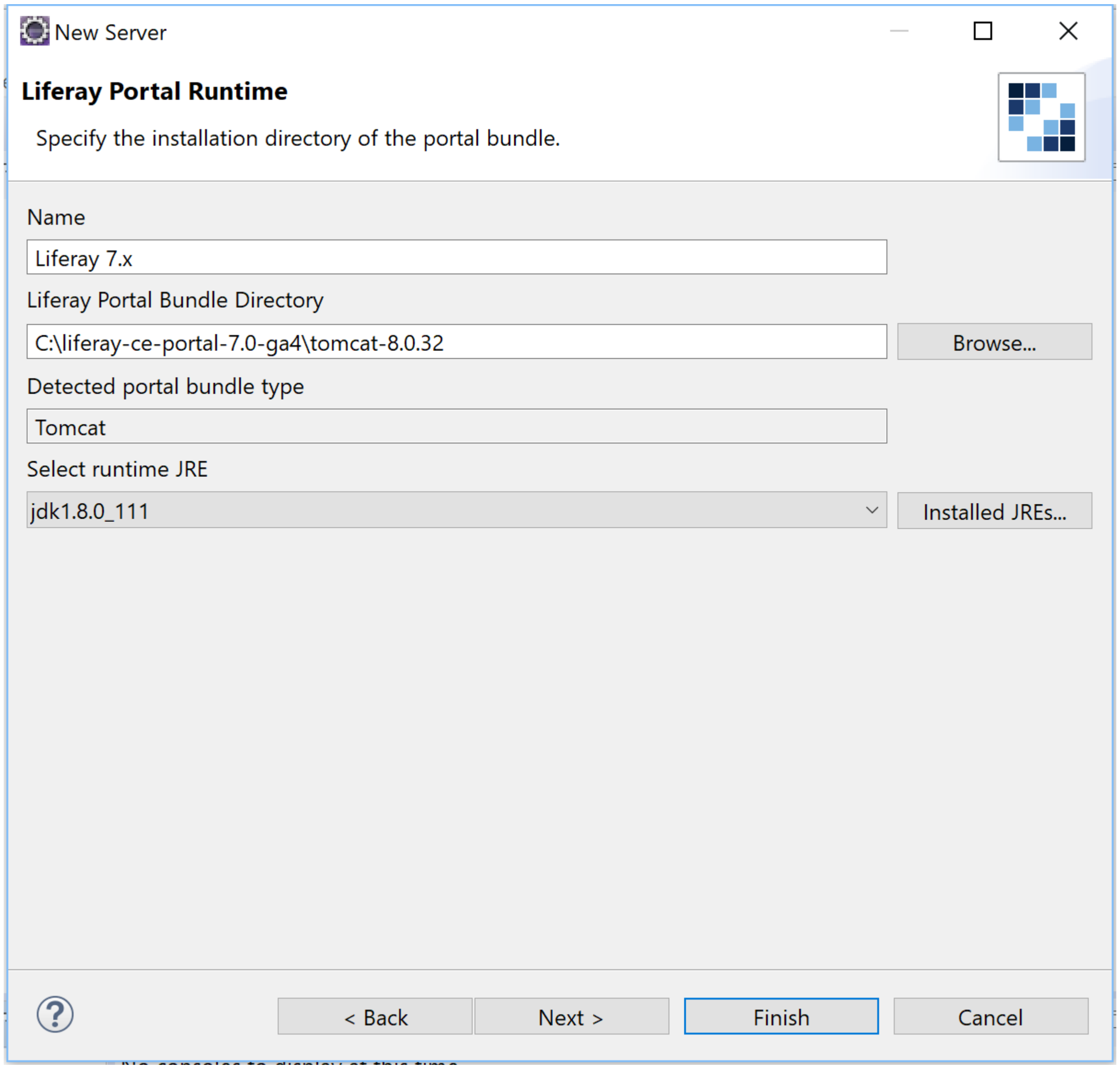
Server runtime environment: Liferay 7.x

Figure 52.25: Choose the type of server you want to create.

server runtime environment, select *Finish*. If you selected an existing server, your server installation is finished; you can skip steps 3-5.

3. Enter a name for your server. This is the name for the Liferay DXP runtime configuration used by @ide@. This is not the display name used in the Servers tab.

4. Browse to the installation folder of the Liferay DXP bundle. For example, `C:\liferay-ce-portal-7.0-ga4\tomcat-8.0.32`.


New Server

Liferay Portal Runtime

Specify the installation directory of the portal bundle.

Name: Liferay 7.x

Liferay Portal Bundle Directory: C:\liferay-ce-portal-7.0-ga4\tomcat-8.0.32

Detected portal bundle type: Tomcat

Select runtime JRE: jdk1.8.0_111

Figure 52.26: Specify the installation folder of the bundle.

5. Select a runtime JRE and click *Finish*. Your new server appears under the Servers view.

Your server is now available in Liferay @ide@!
For reference, here's how the Liferay DXP server buttons work with your Liferay DXP instance:

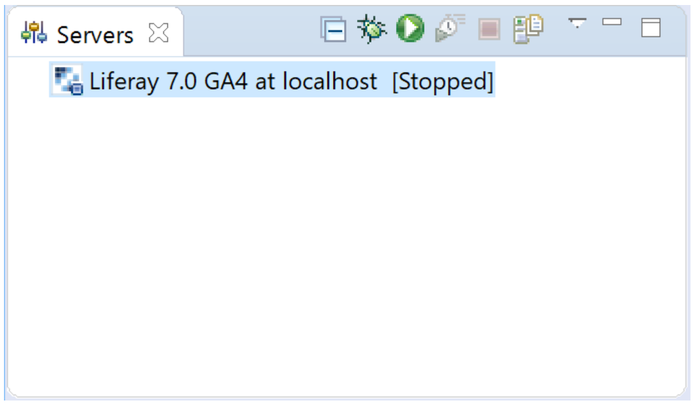


Figure 52.27: Your new server appears under the *Servers* view.

- *Start* (▶): Starts the server.
- *Stop* (■): Stops the the server.
- *Debug* (✱): Starts the server in debug mode. For more information on debugging in Eclipse, see the Eclipse Debugging article.

Now you're ready to use your server in Liferay @ide@!

52.10 Searching Liferay DXP Source in Liferay @ide@

In Liferay Liferay DXP, you can search through Liferay DXP's source code to aid in the development of your project. Liferay provides great resources to help with development (e.g., official documentation, docs.liferay.com, sample projects, etc.), but sometimes, searching through Liferay's codebase (i.e., platform and official apps) for patterns is just as useful. For example, if you're creating a custom app that extends a class provided in Liferay's `portal-kernel` JAR, you can inspect that class and research how it's used in other areas of Liferay DXP's codebase.

To do this, you must be developing in a Liferay Workspace. Liferay Workspace is able to provide this functionality by targeting a specific Liferay DXP version, which indexes the configured Liferay DXP source code to provide advanced search. See the Managing the Target Platform in Liferay Workspace tutorial for more information on how this works.

Workspace does not perform portal source indexing by default. You must enable this functionality by adding the following property to your workspace's `gradle.properties` file:

```
target.platform.index.sources=true
```

Note: Portal source indexing is disabled in Gradle workspace version 2.0.3+ (Target Platform plugin version 2.0.0+).

In this tutorial, you'll explore three use cases where advanced search would be useful.

- Search class hierarchy
- Search declarations
- Search references

These examples are just a small subset of what you can search in Liferay @ide@. See Eclipse's documentation on Java Search for a comprehensive guide.

Search Class Hierarchy

Inspecting classes that extend a similar superclass can help you find useful patterns and examples for how you can develop your own app. For example, suppose your app extends the MVCPortlet class. You an search classes that extend that same class in @ide@ by right-clicking the `MVCPortlet` declaration and selecting *Open Type Hierarchy*. This opens a window that lets you inspect all classes residing in the target platform that extend `MVCPortlet`.

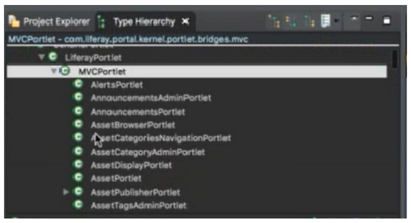


Figure 52.28: Browse the Type Hierarchy window and open the provided classes for examples on how to extend a class.

Great! Now you can search for all extensions and implementations of a class/interface to aid in your quest for developing the perfect app.

Search Method Declarations

Sometimes you want a search to be more granular, exploring the declarations of a specific method provided by a class/interface. Liferay @ide@'s advanced search has no limits; Liferay Workspace's target platform indexing provides method exploration too!

Suppose in the MVCPortlet class you're extending, you'd like to search for declarations of its `doView` method you're overriding. You can do this by right-clicking the `doView` method declaration in your custom app's class and selecting *Declarations → Workspace*.

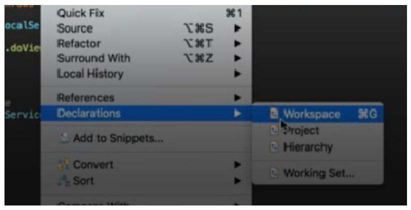


Figure 52.29: All declarations of the method are returned in the Search window.

The rendered Search window displays the other occurrences in the target platform where that method was overridden.

Search Annotation References

Annotations used in Liferay DXP's source code can sometimes be cryptic. With the ability to search where these types of annotations reside in Liferay's target platform, you can find how they could be used in your own app.

For example, you may find some official documentation on using the `@Reference` annotation in an OSGi module and implement it in your custom app. It could be useful to reference real world examples in Liferay

DXP's apps to check how it was used elsewhere. You could search for this by right-clicking the annotation in a class and selecting *References → Workspace*.

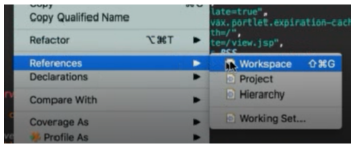


Figure 52.30: All matching annotations are displayed in the Search window.

The rendered Search window displays the other occurrences in the target platform where that annotation was used.

Excellent! You now have the tools to search the configured target platform specified in your Liferay Workspace!

52.11 Debugging Liferay DXP Source in Liferay @ide@

You can use Liferay Liferay DXP to debug Liferay DXP source code to help resolve errors. Debugging Liferay DXP code follows most of the same techniques associated with debugging in Eclipse. If you need some help with general debugging, you can visit Eclipse's documentation. Here's some helpful Eclipse links to visit:

- Debugger
- Local Debugging
- Remote Debugging

There are a couple Liferay-specific configurations to know before debugging Liferay DXP code:

- Configure your target platform.
- Configure a Liferay server and start it in debug mode.

Let's explore these Liferay-specific debugging configurations.

Configure Your Target Platform

To configure your target platform, you must be developing in a Liferay Workspace. Liferay Workspace is able to provide debugging capabilities by targeting a specific Liferay DXP version, which indexes the configured Liferay DXP source code. You must enable this functionality by adding the following property to your workspace's `gradle.properties` file:

```
target.platform.index.sources=true
```

Note: Portal source indexing is disabled in Gradle workspace version 2.0.3+ (Target Platform plugin version 2.0.0+).

Without specifying a target platform, Liferay DXP's source code cannot be accessed by @ide@. See the Managing the Target Platform for Liferay Workspace tutorial for more information on how this works.

Important: The target platform should match the Liferay server version you configure in the next section.

Once the target platform is configured in your workspace, Eclipse has access to all of Liferay DXP's source code. Next, you'll configure a Liferay server and learn how to start it in Debug mode.

Configure a Liferay Server and Start It in Debug Mode

Configuring your target platform gives Eclipse Liferay DXP's source code to reference. Now you must configure a Liferay server matching the target platform version so you can deploy the custom code you wish to debug.

1. Set up your Liferay DXP server to run in @ide@. See the Installing a Server in Liferay @ide@ for more details.

2. Start the server in debug mode. To do this, click the debug button in the Servers pane of Liferay @ide@.

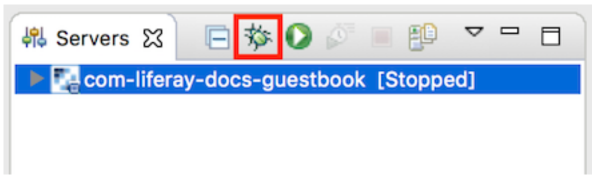


Figure 52.31: The red box in this screenshot highlights the debug button. Click this button to start the server in debug mode.

Awesome! You're now equipped to begin debugging in Liferay @ide@!

52.12 Enabling Code Assist Features in Your Project

Liferay @ide@'s integration of Tern provides many valuable front-end and back-end development tools for code inference and completion. This tutorial covers how to enable Tern features for your projects.

Before beginning this tutorial, make sure your @ide@ instance has the necessary development tooling and Tern integration installed. To to this, go to *Help* → *Installation Details* and search for *Liferay IDE AlloyUI* under *Installed Software*. If you have it installed, you can continue to the *Setting Up Tern Features* section; if you do not, you'll need to install it by following the instructions below.

Navigate to *Help* → *Install New Software...* and paste the following link into the *Work with* field:

```
http://releases.liferay.com/tools/ide/latest/stable/
```

Make sure the *Liferay IDE AlloyUI* option is checked and finish the installation process.

Now that the necessary features are installed, follow the steps below to learn how to enable Tern's code assist features in your project.

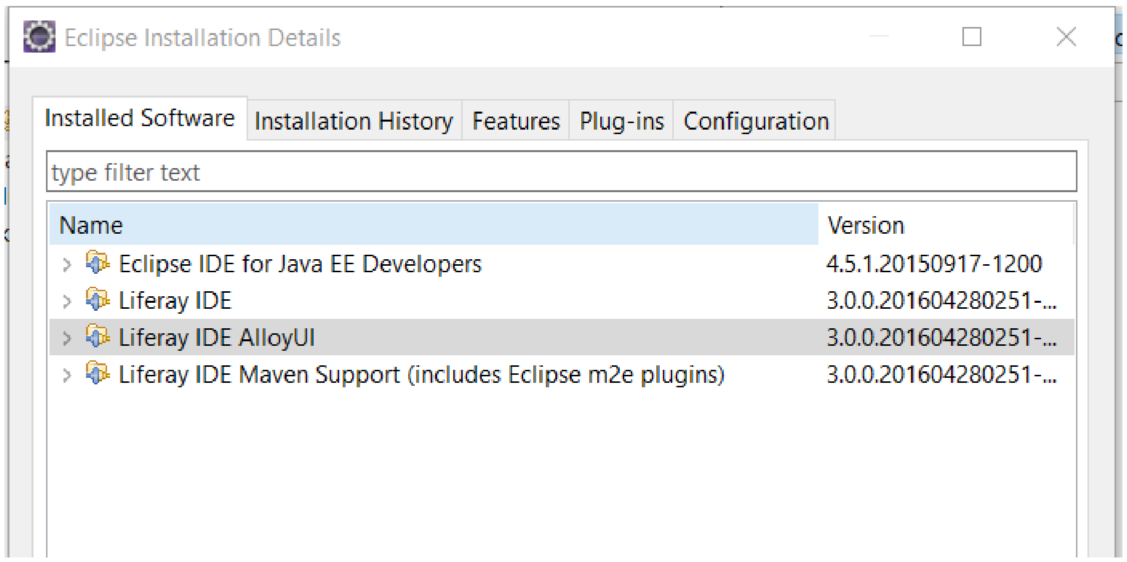
Setting Up Tern Features

Tern features are enabled on a project-by-project basis. By default, Tern is already enabled for Liferay portlet plugins. For all other project types, you'll need to follow the steps below:

1. Right-click on your project and select *Configure* → *Convert to Tern Project*.

   Your project is now configured to use Tern. Now that you have your project configured, you need to enable the modules you want to use for your project.

2. Right-click your project and select *Properties*.


Eclipse Installation Details

Name	Version
Eclipse IDE for Java EE Developers	4.5.1.20150917-1200
Liferay IDE	3.0.0.201604280251-...
Liferay IDE AlloyUI	3.0.0.201604280251-...
Liferay IDE Maven Support (includes Eclipse m2e plugins)	3.0.0.201604280251-...

Figure 52.32: The *Liferay IDE AlloyUI* option is actually a sub-option listed within the *Liferay IDE* option.

3. Select *Tern → Modules*.

   Here you'll find a list of all the available Tern modules you currently have installed. To use AlloyUI features, you'll need the *AlloyUI*, *Browser*, *JSCS*, *Liferay*, and *YUI Library* modules enabled. The figure below shows the Tern Modules menu.

4. Check any additional modules you wish to use in your project and click *OK*.

Your project is now set up to use @ide@'s Tern features.

Related Topics

Using Front-End Code Assist Features in @ide@

Creating Modules with Liferay @ide@

Blade CLI

52.13 Using Gradle in Liferay @ide@

Gradle is a popular open source build automation system. You can take full advantage of Gradle in Liferay @ide@ by utilizing Buildship, which is a collection of Eclipse plugin-ins that provide support for building software using Gradle with Liferay @ide@. Buildship is bundled with Liferay @ide@ versions 3.0 and higher.

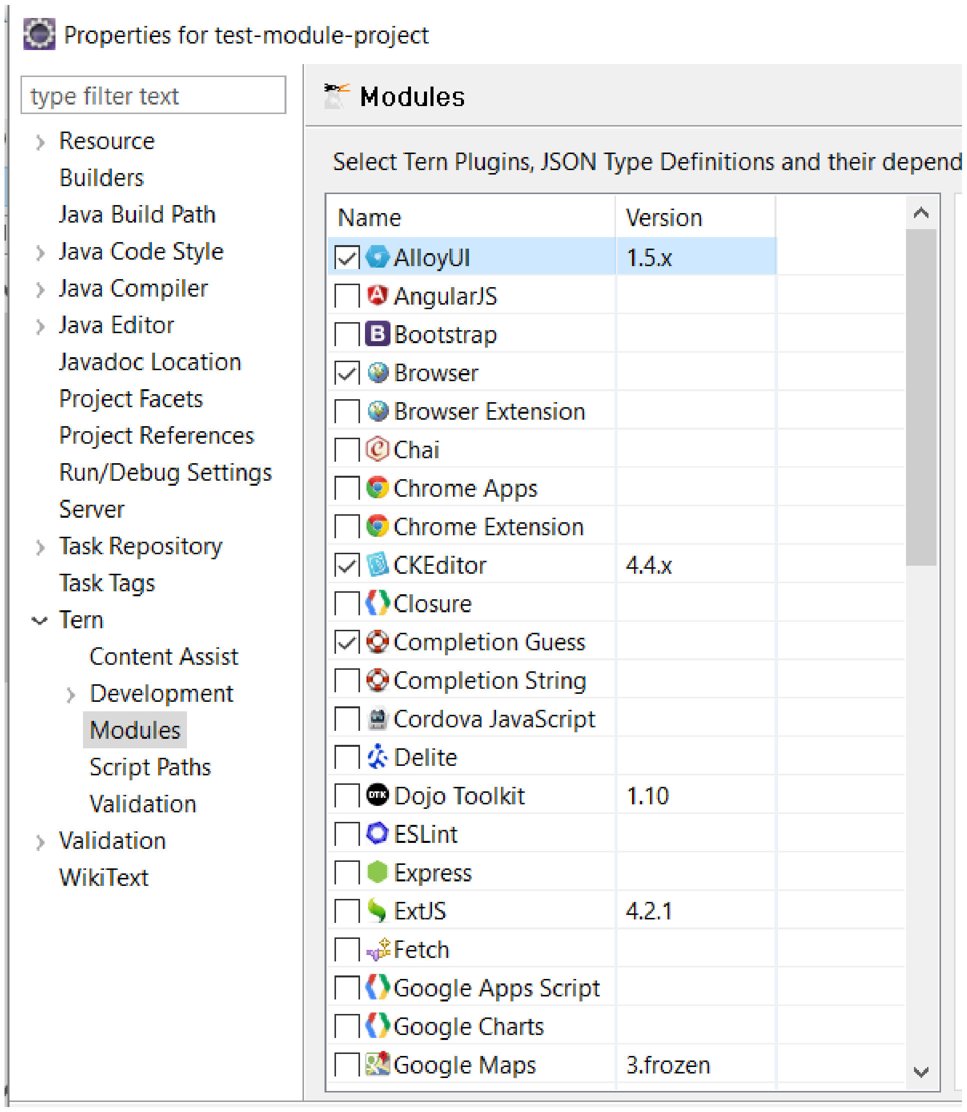
The first thing you'll learn about in this tutorial is creating Gradle projects in @ide@.

Creating and Importing Gradle Projects

You can create a Gradle project by using the Gradle Project wizard.

1. Navigate to *File → New → Project...* and select *Gradle → Gradle Project*. Finally, click *Next → Next*.

2. Enter a valid project name. You can also specify your project location and working sets.


Properties for test-module-project

type filter text

- Resource
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- Project Facets
- Project References
- Run/Debug Settings
- Server
- Task Repository
- Task Tags
- Tern
  - Content Assist
  - Development
  - Modules
  - Script Paths
  - Validation
- Validation
- WikiText

Modules

Select Tern Plugins, JSON Type Definitions and their depend

Name	Version
✓ AlloyUI	1.5.x
☐ AngularJS	
☐ Bootstrap	
✓ Browser	
☐ Browser Extension	
☐ Chai	
☐ Chrome Apps	
☐ Chrome Extension	
✓ CKEditor	4.4.x
☐ Closure	
✓ Completion Guess	
☐ Completion String	
☐ Cordova JavaScript	
☐ Delite	
☐ Dojo Toolkit	1.10
☐ ESLint	
☐ Express	
☐ ExtJS	4.2.1
☐ Fetch	
☐ Google Apps Script	
☐ Google Charts	
☐ Google Maps	3.frozen

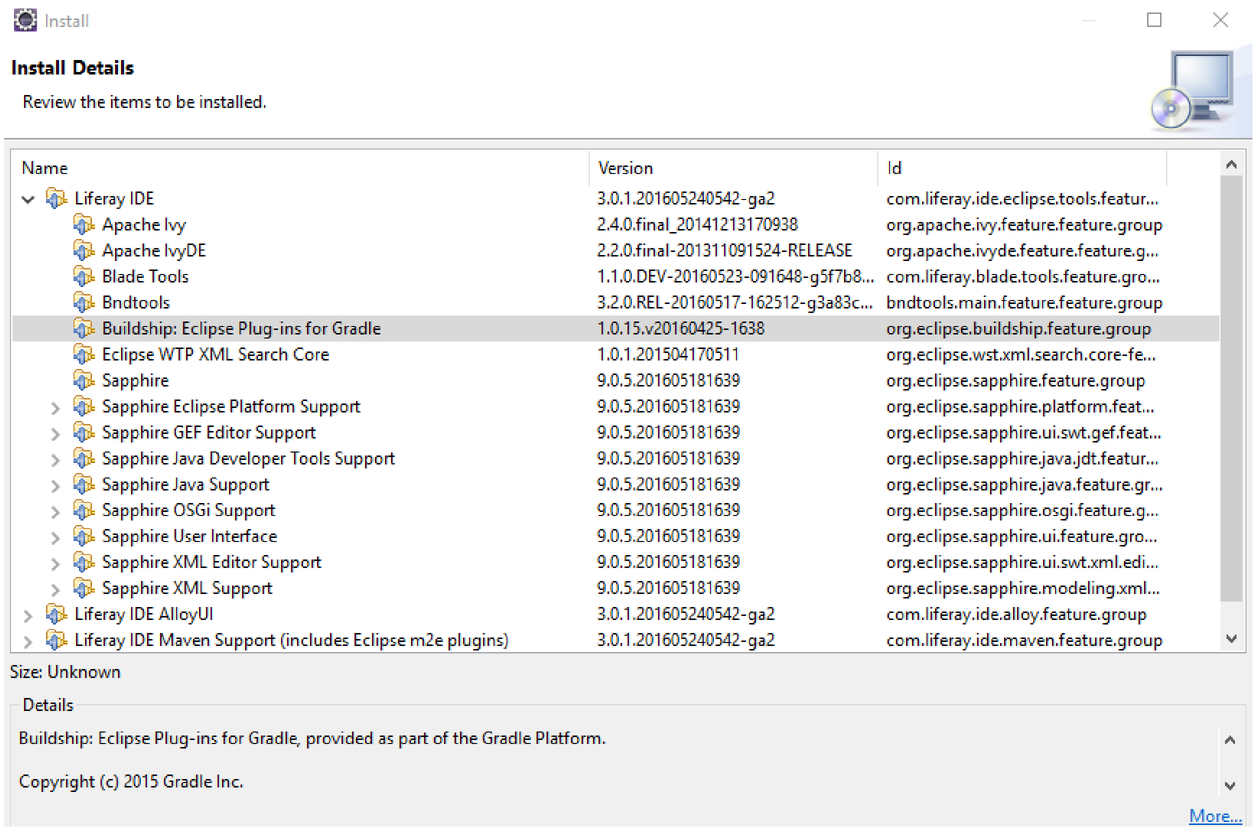
Figure 52.33: By selecting these Tern modules, you can use AlloyUI code assist features in your project.

3. Optionally, you can navigate to the next page and specify your Gradle distribution and other advanced options. Once you're finished, select *Finish*.

You can also import existing Gradle projects in Liferay @ide@.

1. Go to *File* → *Import...* → *Gradle* → *Gradle Project* → *Next* → *Next*.
2. Click the *Browse...* button to choose a Gradle project.
3. Optionally, you can navigate to the next page and specify your Gradle distribution and other advanced options. Once you're finished, click *Next* again to review the import configuration. Select *Finish* once you've confirmed your Gradle project import.

Next you'll learn about Gradle tasks and executions, and learn how to run and display them in Liferay @ide@.


Install

Install Details

Review the items to be installed.

Name	Version	Id
Liferay IDE	3.0.1.201605240542-ga2	com.liferay.ide.eclipse.tools.featur...
Apache Ivy	2.4.0.final_20141213170938	org.apache.ivy.feature.feature.group
Apache IvyDE	2.2.0.final-201311091524-RELEASE	org.apache.ivyde.feature.feature.g...
Blade Tools	1.1.0.DEV-20160523-091648-g5f7b8...	com.liferay.blade.tools.feature.gro...
Bndtools	3.2.0.REL-20160517-162512-g3a83c...	bndtools.main.feature.feature.group
Buildship: Eclipse Plug-ins for Gradle	1.0.15.v20160425-1638	org.eclipse.buildship.feature.group
Eclipse WTP XML Search Core	1.0.1.201504170511	org.eclipse.wst.xml.search.core-fe...
Sapphire	9.0.5.201605181639	org.eclipse.sapphire.feature.group
Sapphire Eclipse Platform Support	9.0.5.201605181639	org.eclipse.sapphire.platform.feat...
Sapphire GEF Editor Support	9.0.5.201605181639	org.eclipse.sapphire.ui.swt.gef.feat...
Sapphire Java Developer Tools Support	9.0.5.201605181639	org.eclipse.sapphire.java.jdt.featur...
Sapphire Java Support	9.0.5.201605181639	org.eclipse.sapphire.java.feature.gr...
Sapphire OSGi Support	9.0.5.201605181639	org.eclipse.sapphire.osgi.feature.g...
Sapphire User Interface	9.0.5.201605181639	org.eclipse.sapphire.ui.feature.gro...
Sapphire XML Editor Support	9.0.5.201605181639	org.eclipse.sapphire.ui.swt.xml.edi...
Sapphire XML Support	9.0.5.201605181639	org.eclipse.sapphire.modeling.xml...
Liferay IDE AlloyUI	3.0.1.201605240542-ga2	com.liferay.ide.alloy.feature.group
Liferay IDE Maven Support (includes Eclipse m2e plugins)	3.0.1.201605240542-ga2	com.liferay.ide.maven.feature.group

Size: Unknown

Details

Buildship: Eclipse Plug-ins for Gradle, provided as part of the Gradle Platform.

Copyright (c) 2015 Gradle Inc.

More...

Figure 52.34: Navigate to *Help* → *Installation Details* to view plugins included in Liferay @ide@.

Gradle Tasks and Executions

Liferay @ide@ provides two views to enhance your developing experience using Gradle: Gradle Tasks and Gradle Executions. You can open these views by following the instructions below.

1. Go to *Window* → *Show View* → *Other...*.
2. Navigate to the *Gradle* folder and open *Gradle Tasks* and *Gradle Executions*.

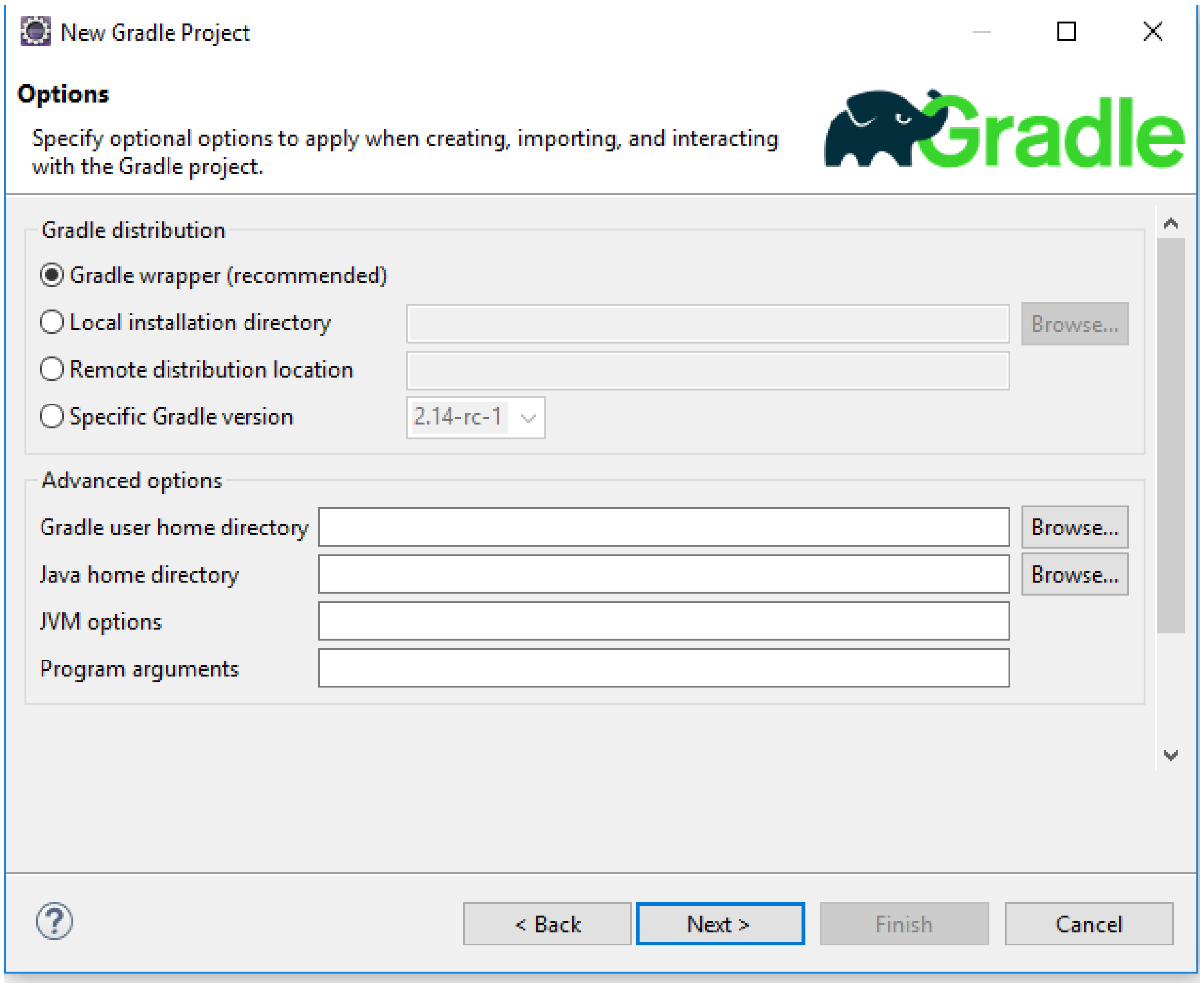
Gradle tasks and executions views open automatically once you create or import a Gradle project.

The Gradle Tasks view allows you to display the Gradle tasks available for you to use in your Gradle project. Users can execute a task listed under the Gradle project by double-clicking it.

Once you've executed a Gradle task, you can open the Gradle Executions view to inspect its output.

Keep in mind that if you change the Gradle build scripts inside your Gradle projects (e.g., `build.gradle` or `settings.gradle`), you must refresh the project so Liferay @ide@ can account for the change and display it properly in your views. To refresh a Gradle project, right-click on the project and select *Gradle* → *Refresh Gradle Project*.

If you prefer Eclipse refresh your Gradle projects automatically, navigate to *Window* → *Preferences* → *Gradle* and enable the *Automatic Project Synchronization* checkbox. If you'd like to enable Gradle's automatic synchronization for just one Gradle project, you can right-click a Gradle project and select *Properties* → *Gradle* and enable auto sync that way. This feature is available in Buildship version 2.2+, so make sure you have the required version.



New Gradle Project

Options

Specify optional options to apply when creating, importing, and interacting with the Gradle project.

Gradle distribution

- [✓ Gradle wrapper (recommended)]
- Local installation directory
- Remote distribution location
- Specific Gradle version: 2.14-rc-1

Advanced options

- Gradle user home directory
- Java home directory
- JVM options
- Program arguments

Figure 52.35: You can specify your Gradle distribution and advanced options such as home directories, JVM options, and program arguments.

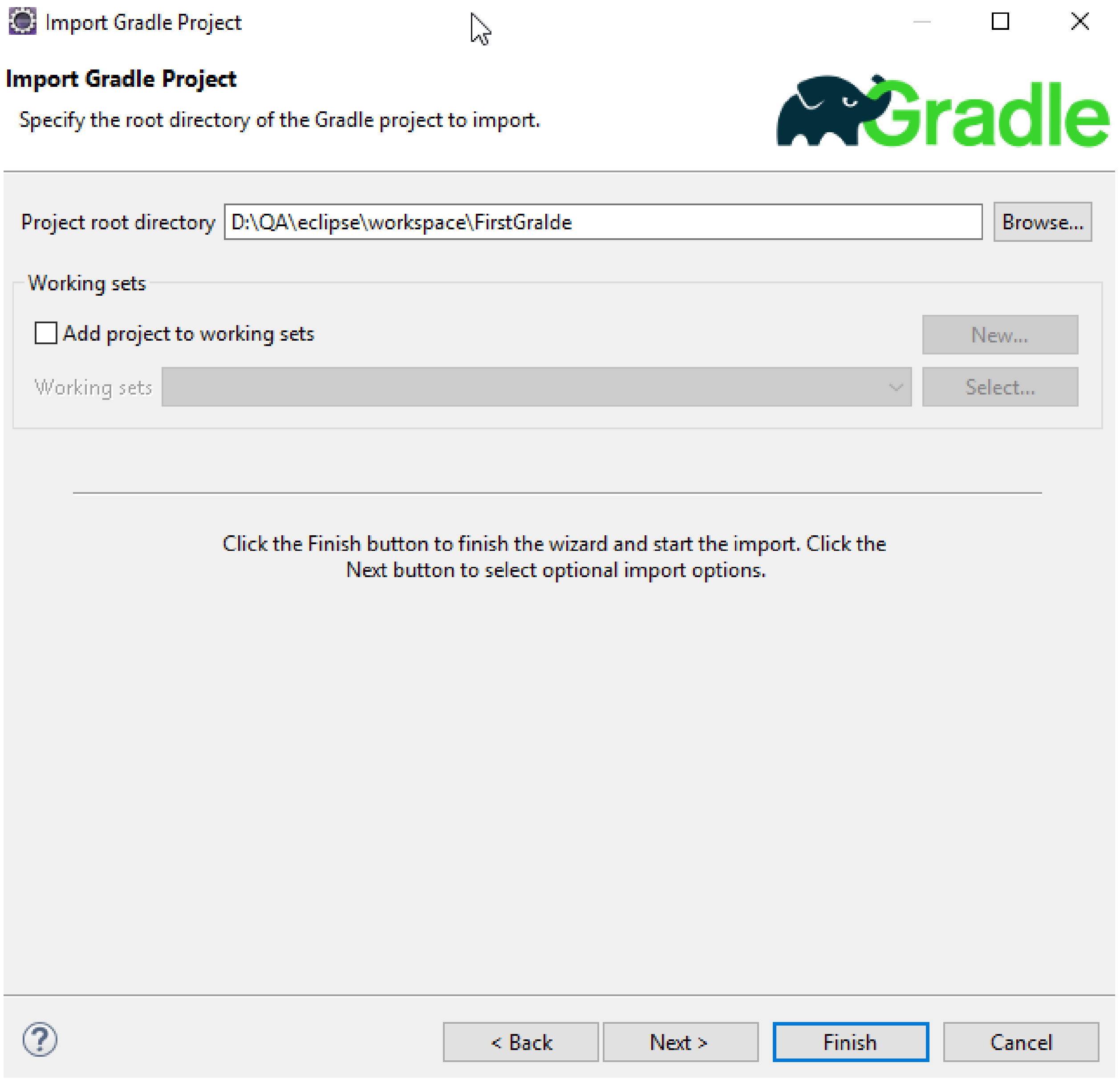
Excellent! You're now equipped with the knowledge to add, import, and build your Gradle projects in Liferay @ide@!

52.14 Using Maven in Liferay @ide@

You can take full advantage of Maven in Liferay @ide@ with its built-in Maven support. In this tutorial, you'll learn about the following topics:

- Installing Maven Plugins for Liferay @ide@
- Creating Maven Projects
- Importing Maven Projects
- Using the POM Graphic Editor

First you'll install the necessary Maven plugins for Liferay @ide@.



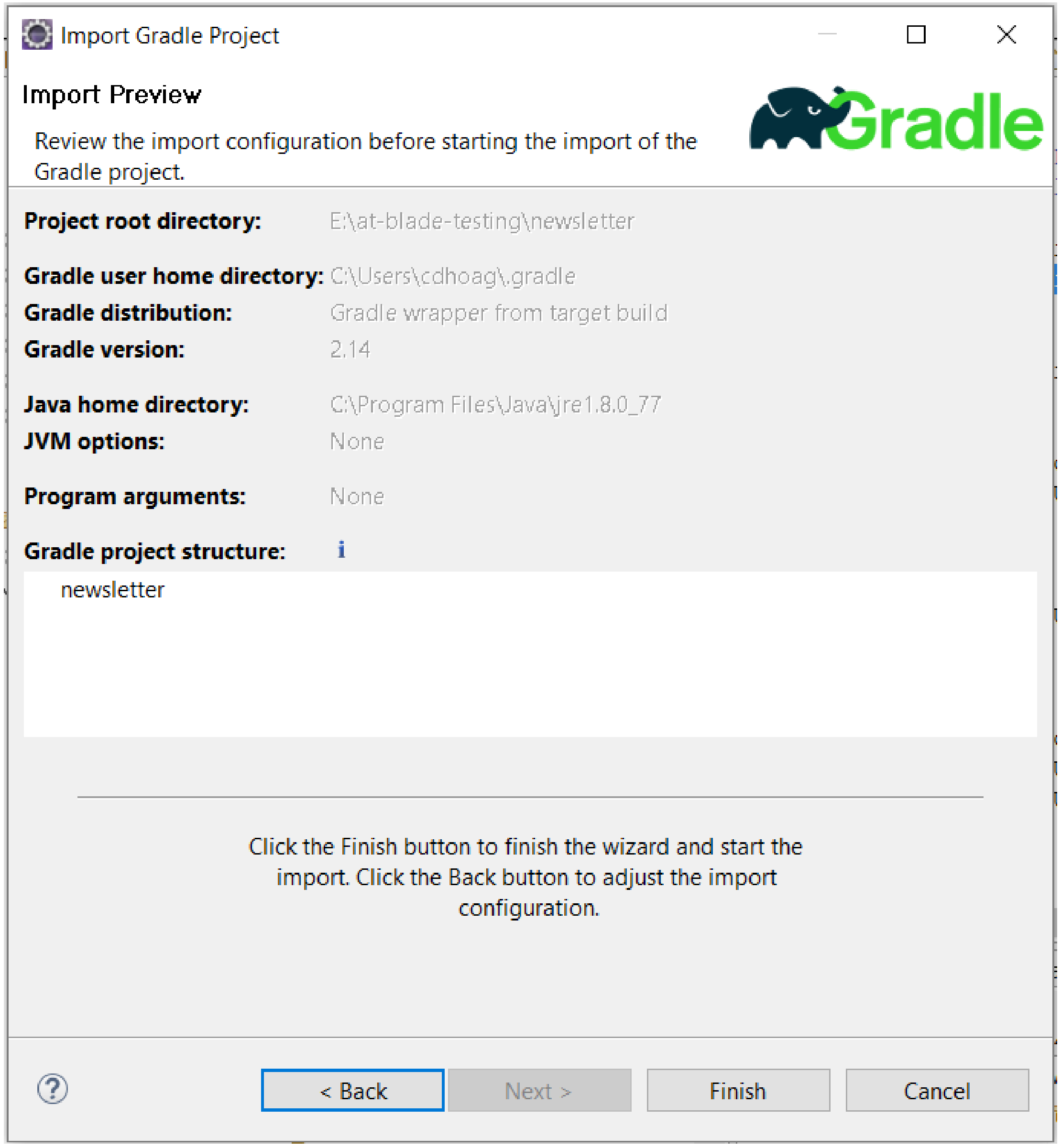
Import Gradle Project

Specify the root directory of the Gradle project to import.

Project root directory: D:\QA\eclipse\workspace\FirstGralde

Click the Finish button to finish the wizard and start the import. Click the Next button to select optional import options.

Figure 52.36: You can specify what Gradle project to import from the *Import Gradle Project* wizard.


Import Gradle Project

Import Preview

Review the import configuration before starting the import of the Gradle project.

Project root directory: E:\at-blade-testing\newsletter

Gradle user home directory: C:\Users\cdhoag\.gradle

Gradle distribution: Gradle wrapper from target build

Gradle version: 2.14

Java home directory: C:\Program Files\Java\jre1.8.0_77

JVM options: None

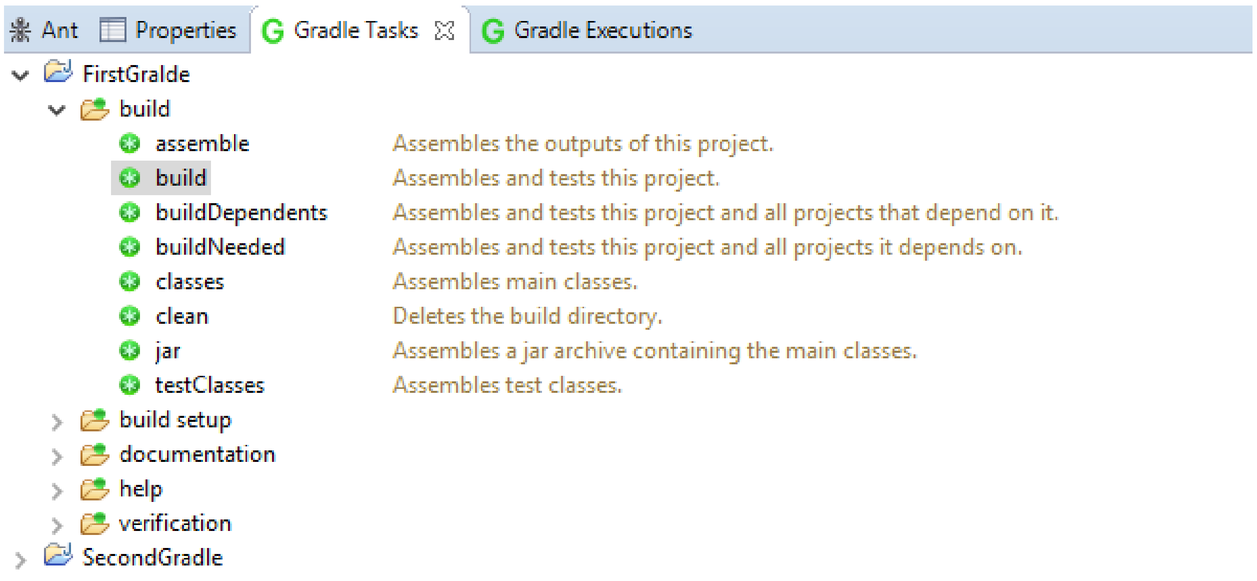
Program arguments: None

Gradle project structure:

newsletter

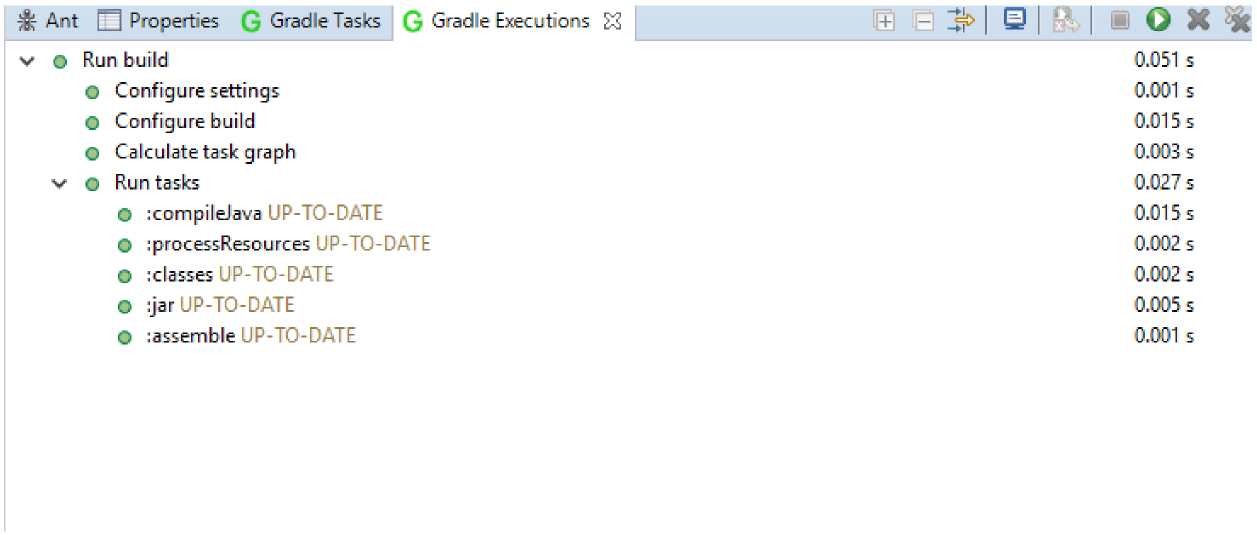
Click the Finish button to finish the wizard and start the import. Click the Back button to adjust the import configuration.

Figure 52.37: You can preview your Gradle project's import information.



```
FirstGralde
  build
    assemble          Assembles the outputs of this project.
    build             Assembles and tests this project.
    buildDependents   Assembles and tests this project and all projects that depend on it.
    buildNeeded       Assembles and tests this project and all projects it depends on.
    classes           Assembles main classes.
    clean             Deletes the build directory.
    jar               Assembles a jar archive containing the main classes.
    testClasses       Assembles test classes.
  build setup
  documentation
  help
  verification
SecondGradle
```

Figure 52.38: Navigate into your preferred Gradle project to view its available Gradle tasks.



```
Run build                                0.051 s
  Configure settings                     0.001 s
  Configure build                        0.015 s
  Calculate task graph                   0.003 s
  Run tasks                              0.027 s
    :compileJava UP-TO-DATE              0.015 s
    :processResources UP-TO-DATE         0.002 s
    :classes UP-TO-DATE                  0.002 s
    :jar UP-TO-DATE                      0.005 s
    :assemble UP-TO-DATE                 0.001 s
```

Figure 52.39: The Gradle Executions view helps you visualize the Gradle build process.

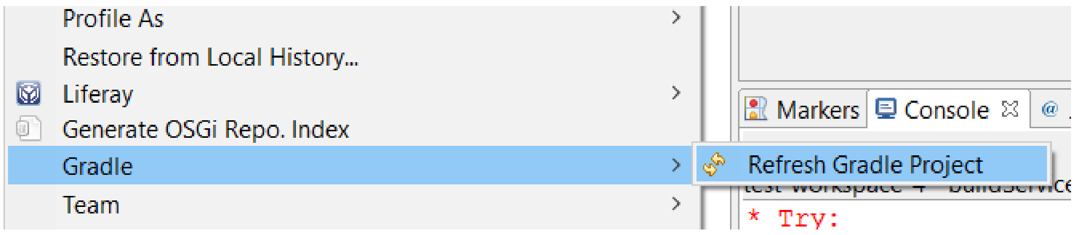


Figure 52.40: Make sure to always refresh your Gradle project in Liferay @ide@ after build script edits.

Installing Maven Plugins for Liferay @ide@

In order to support Maven projects in @ide@ properly, you first need a mechanism to recognize Maven projects as Liferay @ide@ projects. @ide@ projects are recognized in Eclipse as faceted web projects that include the appropriate Liferay plugin facet. Therefore, all @ide@ projects are also Eclipse web projects (faceted projects with the web facet installed). In order for @ide@ to recognize the Maven project and for it to be able to leverage Java EE tooling features (e.g., the Servers view) with the project, the project must be a flexible web project. Liferay @ide@ relies on the following Eclipse plugins to provide this capability:

- `m2e-core` (Maven integration for Eclipse)
- `m2e-wtp` (Maven integration for WTP)

All you have to do is install them so you can begin developing Maven projects for Liferay DXP.

When first installing Liferay @ide@, the installation startup screen lets you select whether you'd like to install the Maven plugins automatically. Don't worry if you missed this during setup. You'll learn how to install the required Maven plugins for @ide@ manually below.

1. Navigate to *Help* → *Install New Software*. In the *Work with* field, insert the following value:

   ```
   Liferay IDE repository - http://releases.liferay.com/tools/ide/latest/milestone/
   ```

2. Check the *Liferay IDE Maven Support* option. This bundles all the required Maven plugins you need to begin developing Maven projects for Liferay DXP.

   If the *Liferay IDE Maven Support* option does not appear, then it's already installed. To verify that it's installed, uncheck the *Hide items that are already installed* checkbox and look for *Liferay IDE Maven Support* in the list of installed plugins. Also, if you'd like to view everything that is bundled with the *Liferay IDE Maven Support* option, uncheck the *Group items by category* checkbox.

3. Click *Next*, review the install details, accept the term and license agreements, and select *Finish*.

Awesome! Your Liferay DXP is ready to develop Maven projects for Liferay DXP!

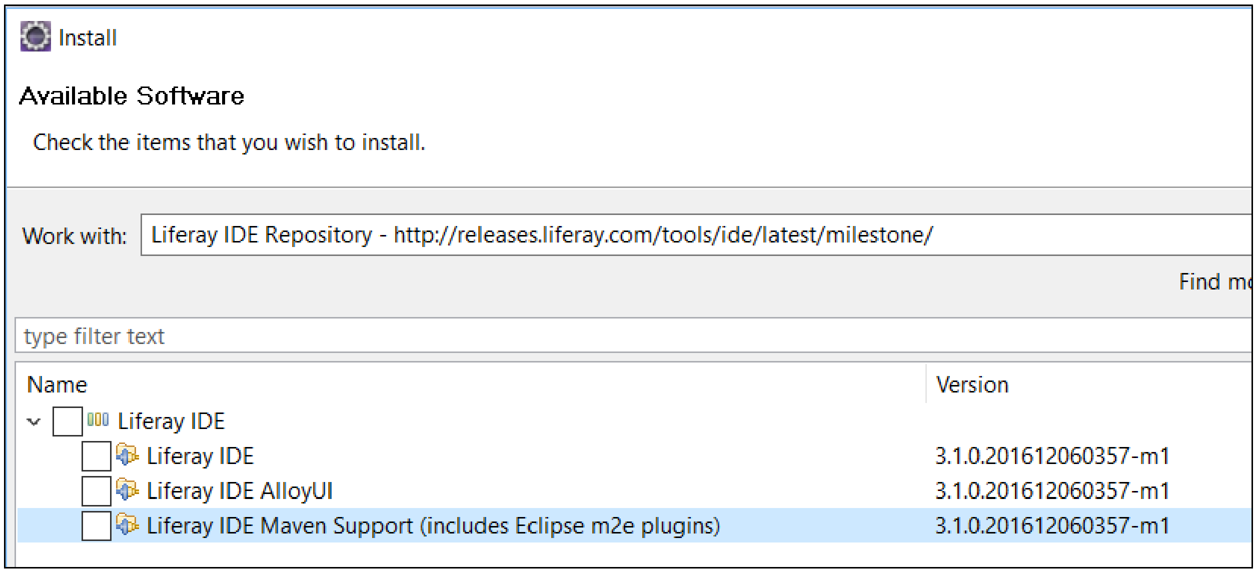
You'll learn about creating Maven projects in @ide@ next.

Creating Maven Projects

You can create a Maven project based on Liferay's provided Maven archetypes.

1. Navigate to *File* → *New* → *Liferay Module Project*.

2. Give your project a name, select the `maven-module` build type, and choose the project template (archetype) you'd like to use.


Install

Available Software

Check the items that you wish to install.

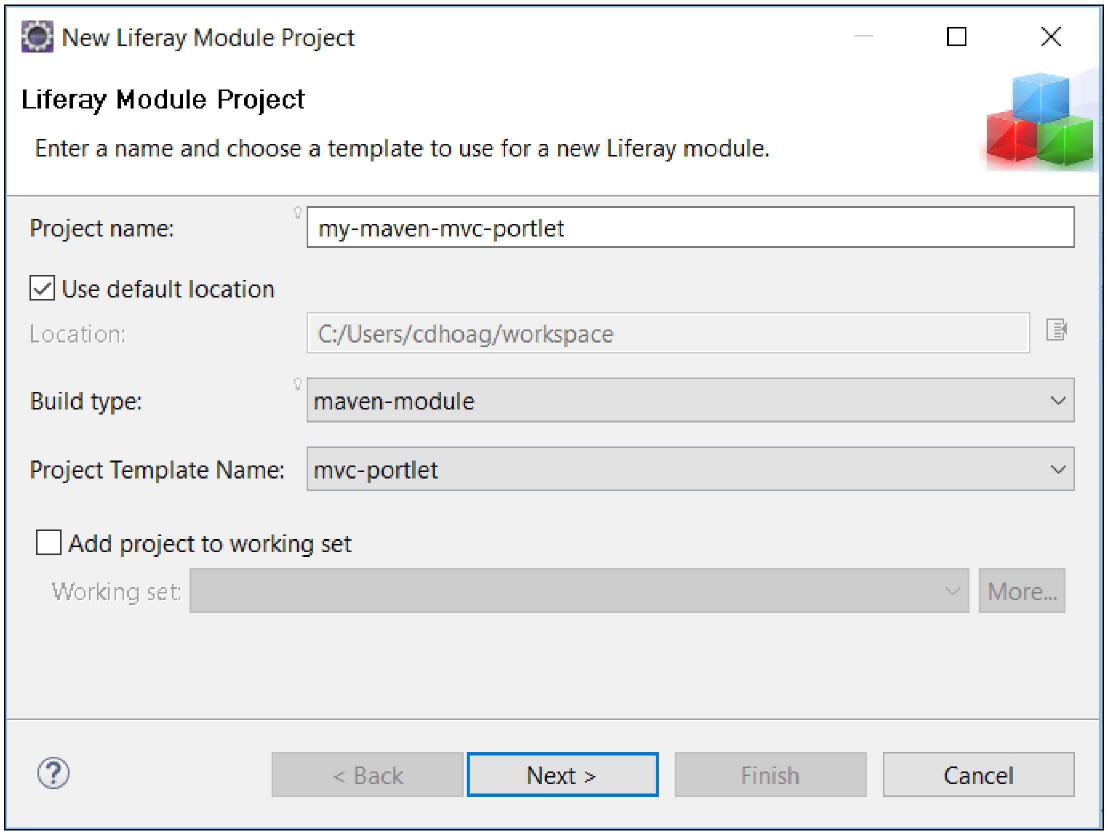
Work with: Liferay IDE Repository - http://releases.liferay.com/tools/ide/latest/milestone/

Find mo

type filter text

Name	Version
Liferay IDE	
Liferay IDE	3.1.0.201612060357-m1
Liferay IDE AlloyUI	3.1.0.201612060357-m1
Liferay IDE Maven Support (includes Eclipse m2e plugins)	3.1.0.201612060357-m1

Figure 52.41: You can install all the necessary Maven plugins for @ide@ by installing the *Liferay IDE Maven Support* option.


New Liferay Module Project

Liferay Module Project

Enter a name and choose a template to use for a new Liferay module.

Project name: my-maven-mvc-portlet

[✓] Use default location

Location: C:/Users/cdhoag/workspace

Build type: maven-module

Project Template Name: mvc-portlet

[ ] Add project to working set

Working set:

Figure 52.42: The New Liferay Module Project wizard lets you generate a Maven module project.

3. (Optional) Click *Next* and name your component class name and package. You can also specify your component class's properties in the Properties menu.

4. Click *Finish*.

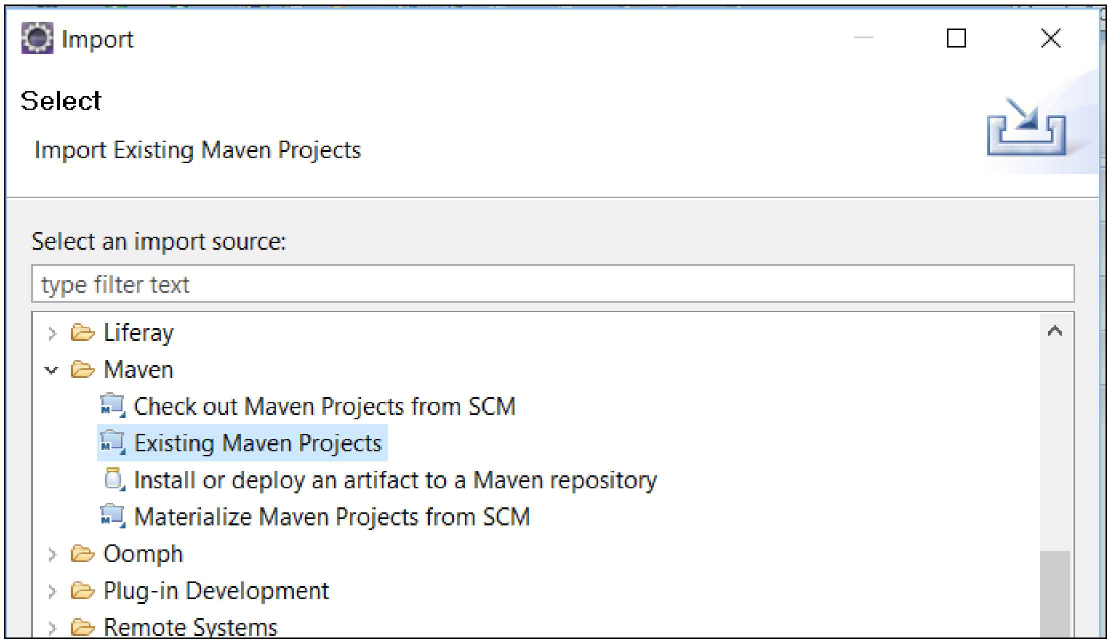
That's it! You've created a Liferay module project using Maven!

If you created your Maven project outside of @ide@ with another tool, you can still manage that project in @ide@, but you must first import it. You'll learn how to do this next.

Importing Maven Projects

To import a pre-existing Maven project into Liferay @ide@, follow the steps outlined below:

1. Navigate to *File* → *Import* → *Maven* → *Existing Maven Projects* and click *Next*.



Import

Select

Import Existing Maven Projects

Select an import source:

type filter text

- Liferay
- Maven
  - Check out Maven Projects from SCM
  - Existing Maven Projects
  - Install or deploy an artifact to a Maven repository
  - Materialize Maven Projects from SCM
- Oomph
- Plug-in Development
- Remote Systems

Figure 52.43: @ide@ offers the Maven folder in the Import wizard.

2. Click *Browse...* and select the root folder for your Maven project. Once you've selected it, the `pom.xml` for that project should be visible in the Projects menu.

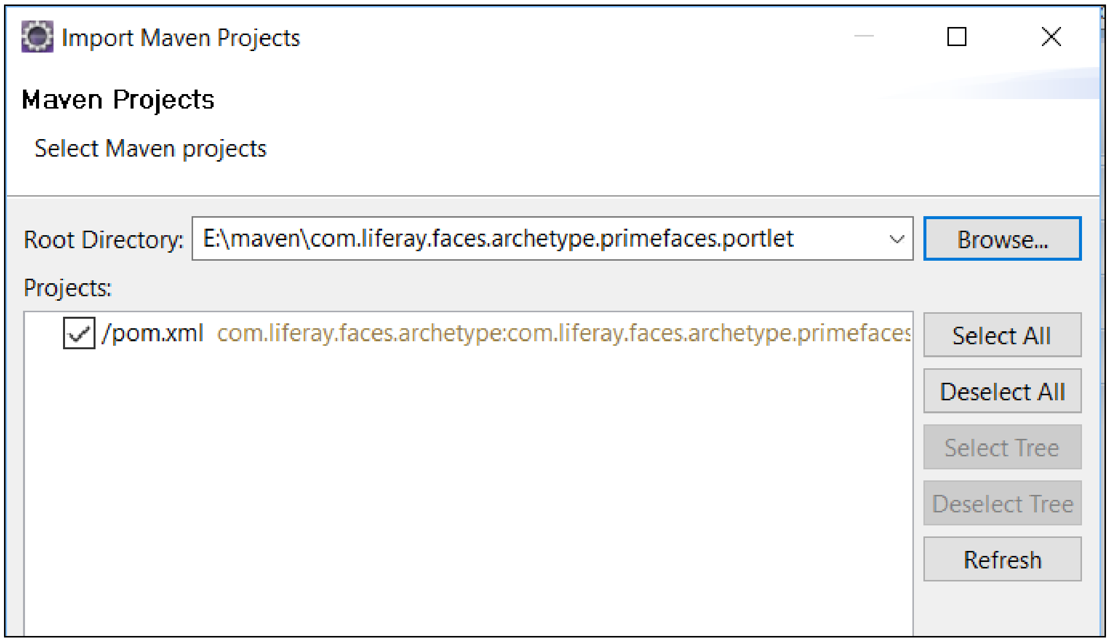
3. Click *Finish*.

Now your Maven project is available from the Package Explorer. Next you'll learn about Liferay @ide@'s POM graphical editor.

Using the POM Graphic Editor

You're provided a nifty POM graphic editor when opening your Maven project's `pom.xml` in Liferay @ide@. This gives you several different ways to leverage the power of Maven in your project:

- **Overview:** provides a graphical interface where you can add to and edit the `pom.xml` file.



E:\maven\com.liferay.faces.archetype.primefaces.portlet

☑ /pom.xml com.liferay.faces.archetype:com.liferay.faces.archetype.primefaces

Figure 52.44: Use the Import Maven Projects wizard to import your pre-existing project.

- **Dependencies:** provides a graphical interface for adding and editing dependencies in your project, as well as modifying the `dependencyManagement` section of the `pom.xml` file.
- **Effective POM:** provides a read-only version of your project POM merged with its parent POM(s), `settings.xml`, and the settings in Eclipse for Maven.
- **Dependency Hierarchy:** provides a hierarchical view of project dependencies and an interactive listing of resolved dependencies.
- **pom.xml:** provides an editor for your POM's source XML.

The figure below shows the `pom.xml` file editor and its modes.

By taking advantage of these interactive modes, Liferay @ide@ makes modifying and organizing your POM and its dependencies a snap!

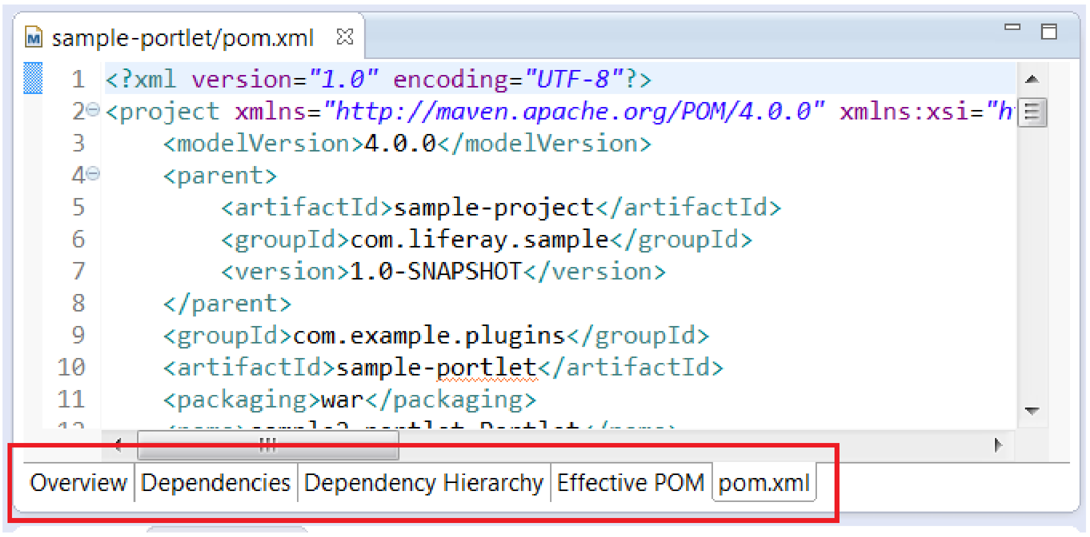
52.15 Using Front-End Code Assist Features in @ide@

Liferay @ide@ provides extended front-end development tools to assist in Liferay development. You now have access to code inferencing and code completion features for AlloyUI, JavaScript, CSS, and jQuery.

This tutorial covers how to use the code assist features for AlloyUI, JavaScript, CSS, and jQuery in @ide@. Each language is covered in its own section, so you can navigate to the language you're most interested in. Continue reading to find out how to use @ide@'s code assist features in your project.

Using Code Assist Features

@ide@'s integration of Tern gives you access to code assist in JavaScript, AlloyUI, and CSS. To access these features, you must be working in a JavaScript, JSP, HTML, or CSS file.


sample-portlet/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>sample-project</artifactId>
        <groupId>com.liferay.sample</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <groupId>com.example.plugins</groupId>
    <artifactId>sample-portlet</artifactId>
    <packaging>war</packaging>
```

Figure 52.45: Liferay @ide@ provides five interactive modes to help you edit and organize your POM..

You must have Tern features enabled in your project in order to use them. By default, Liferay portlet plugins already have Tern features enabled. Visit the Enabling Code Assist Features in your Project tutorial to learn how to enable Tern features for non-Liferay specific projects.

Note: For those developing with the Plugins SDK, the taglib descriptions that @ide@ makes available to users are dependent upon the Plugins SDK version. @ide@ uses taglib descriptions from the current SDK's `util-taglib.jar` file, so a more up-to-date Plugins SDK means more up-to-date taglib descriptions.

You'll begin testing the AlloyUI code assist features next.

AlloyUI Code Assist Features

There are several helpful code assist features that can improve your productivity when writing code for AlloyUI. The example below shows how to access the AlloyUI code assist features in the `main.js` of your project:

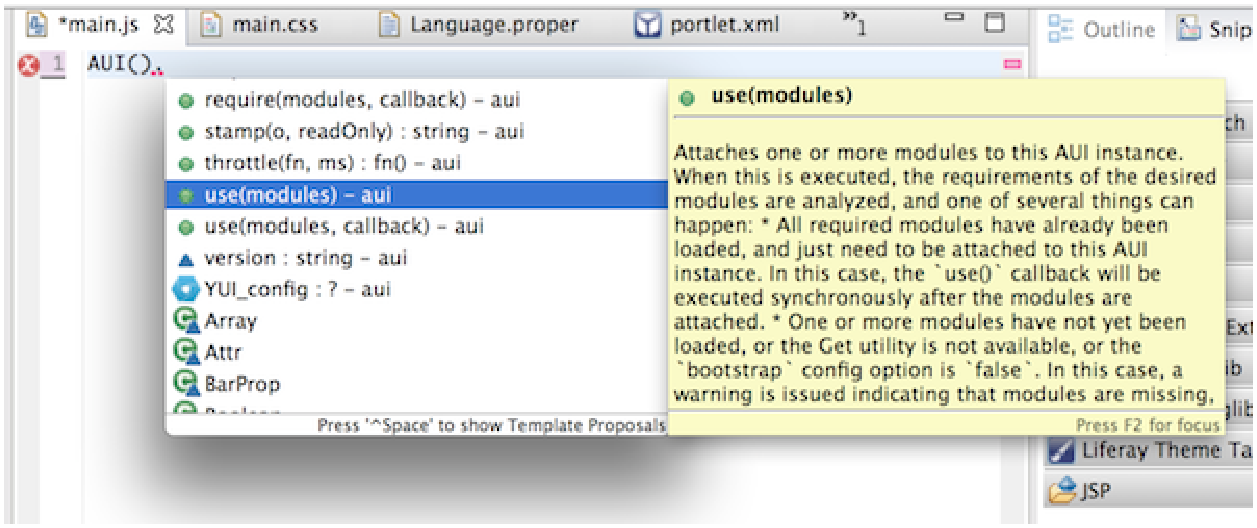
1. Open your project's `main.js` file and type the following code:

   ```
   AUI().
   ```

2. Press *Ctrl+Space* with your cursor to the right of `AUI().`. This brings up the code inference for the `AUI()` global object. Notice the AlloyUI framework's own API documentation is also displayed. Press *Enter* to use code completion.

Note: Code assist not only works for methods of an object, but also works for AUI-specific Tern completions for objects. For instance, you could type `AU` and press *Ctrl+Space* to see a list of objects to choose from.

By default, code inference is triggered by a keystroke combination; however, you can enable auto activation in @ide@'s Preferences menu. Follow the steps below to enable auto activation:



use(modules)

Attaches one or more modules to this AUI instance. When this is executed, the requirements of the desired modules are analyzed, and one of several things can happen: * All required modules have already been loaded, and just need to be attached to this AUI instance. In this case, the `use()` callback will be executed synchronously after the modules are attached. * One or more modules have not yet been loaded, or the Get utility is not available, or the `bootstrap` config option is `false`. In this case, a warning is issued indicating that modules are missing,

Figure 52.46: This figure demonstrates code inference in a JS file.

1. Navigate to *Window* → *Preferences* → *JavaScript* → *Editor* → *Content Assist*.
2. Check the *Enable auto activation* box and click *Apply*. Then click *OK*.

The figure below shows how to enable auto activation:

Now, if you follow the previous example, code inference activates as soon as you press the trigger key, which in this case is the . (period) key.

In addition to general code inference for AlloyUI, you have access to code templates. AUI JavaScript templates are available in Eclipse's JavaScript editor as well as in the HTML/JSP editor when working with `<script>` and `<aui-script>` tags. Follow the steps below to use AUI code templates:

1. Type the following code in your `main.js`:

   ```
   AUI
   ```

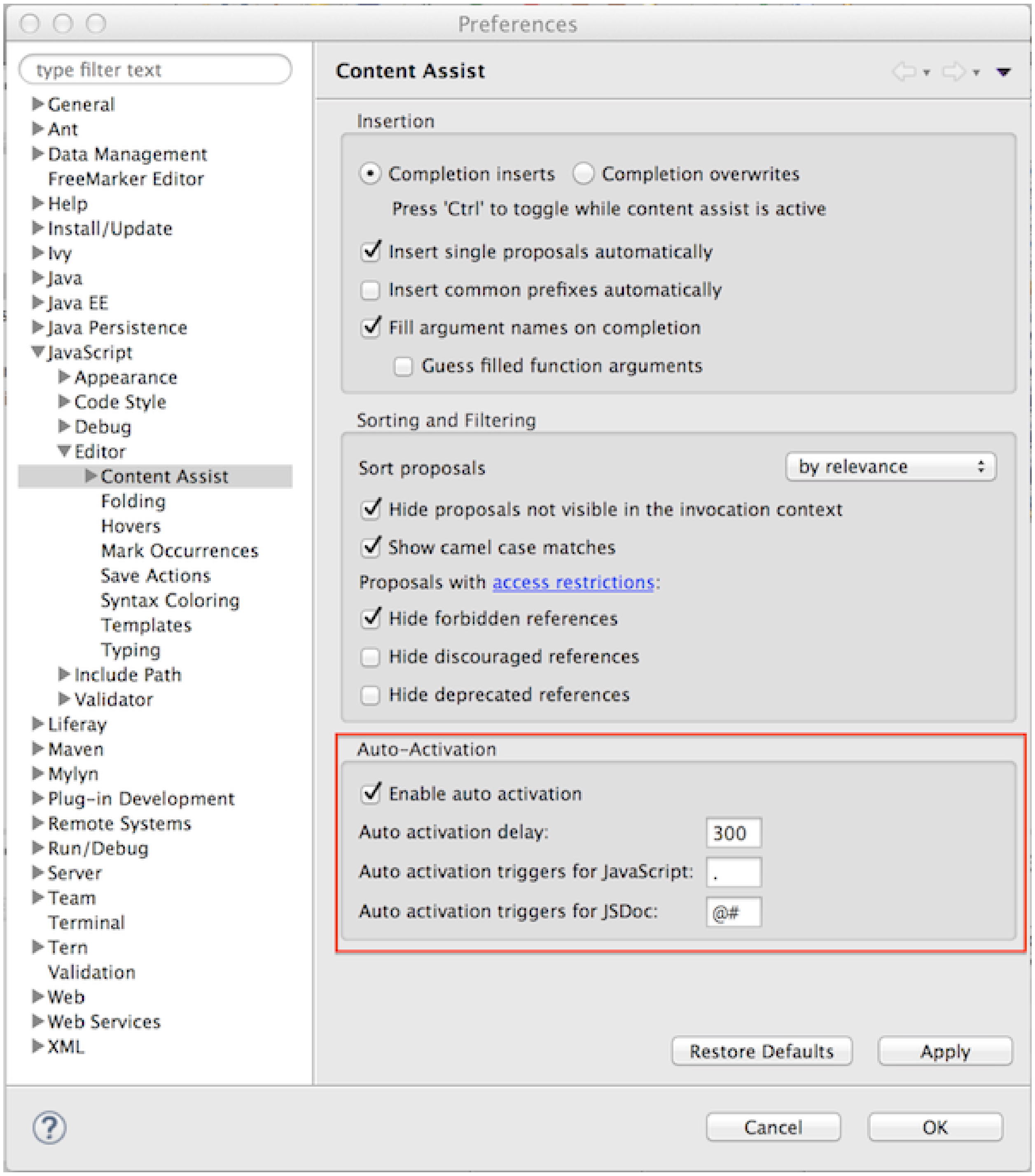
2. Press *Ctrl+Space* to bring up the code inference for `AUI`, and you'll see a list of all the available AlloyUI code templates, along with documentation.
3. Select your template and hit *Enter* to paste its contents into your `main.js`.

```
**Note:** You can view all the AlloyUI code templates you have
installed by going to @ide@'s Preferences menu and selecting *JavaScript*
&rarr; *Editor* &rarr; *Templates*.
```

In addition to code inference in your JS files, you can also use code inference in your JSP/HTML files using `<aui:script>` tags.

Open one of your project's JSPs and add the AUI taglib directive if it is not already in your JSP:

```
<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
```



Content Assist

Insertion

[● Completion inserts] ○ Completion overwrites

Press 'Ctrl' to toggle while content assist is active

✓ Insert single proposals automatically

☐ Insert common prefixes automatically

✓ Fill argument names on completion

☐ Guess filled function arguments

Sorting and Filtering

Sort proposals: by relevance

✓ Hide proposals not visible in the invocation context

✓ Show camel case matches

Proposals with access restrictions:

✓ Hide forbidden references

☐ Hide discouraged references

☐ Hide deprecated references

Auto-Activation

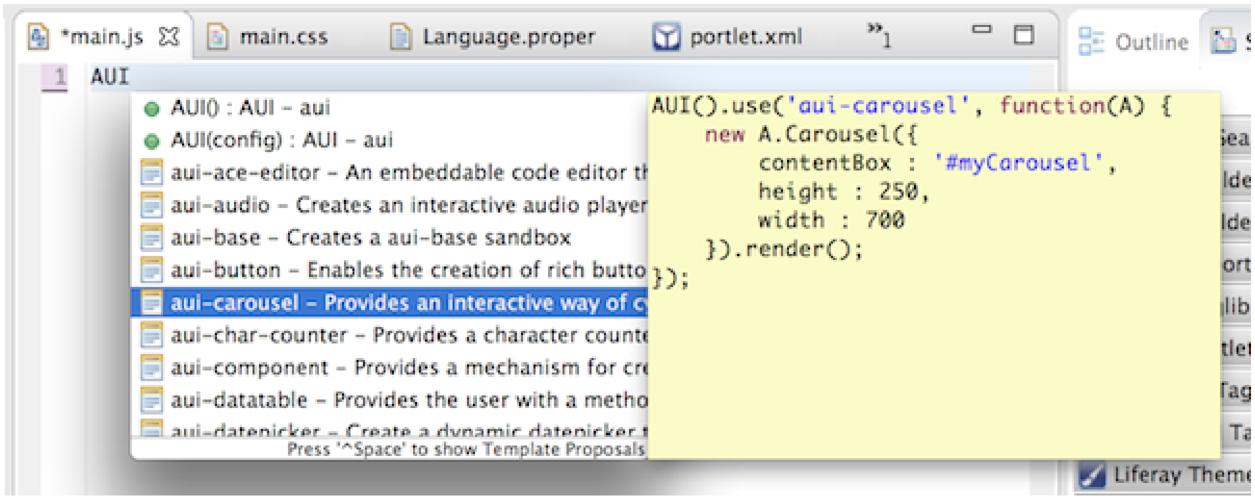
✓ Enable auto activation

Auto activation delay: 300

Auto activation triggers for JavaScript: .

Auto activation triggers for JSDoc: @#

Figure 52.47: The *Enable auto activation* checkbox is listed below the *Auto-Activation* heading.


*main.js
main.css
Language.proper
portlet.xml
Outline
AUI
AUI() : AUI - aui
AUI(config) : AUI - aui
aui-ace-editor - An embeddable code editor t
aui-audio - Creates an interactive audio player
aui-base - Creates a aui-base sandbox
aui-button - Enables the creation of rich butto
aui-carousel - Provides an interactive way of c
aui-char-counter - Provides a character counte
aui-component - Provides a mechanism for cr
aui-datatable - Provides the user with a metho
Press '^Space' to show Template Proposals
Liferay Theme

```
AUI().use('aui-carousel', function(A) {
    new A.Carousel({
        contentBox : '#myCarousel',
        height : 250,
        width : 700
    }).render();
});
```

Figure 52.48: @ide@ gives you access to AUI code templates in the JS and JSP editors.

You can also add the import from the Snippets menu under *Taglib imports → Liferay AUI Taglib Import v6.1*.

1. Add an `<aui:script>` tag inside your JSP and configure it to look like the following code:

```
<aui:script>
    aui
</aui:script>
```

2. Press *Ctrl+Space* with your cursor placed to the right of `aui` to bring up code inference.

There you go! Whether in a JavaScript file or inside a JSP, you now have access to code assist features that improve your workflow.

Next, you'll examine the JavaScript code assist features for @ide@.

JavaScript Code Assist Features

In addition to AlloyUI code assist features, you also have access to code inference and completion using raw JavaScript. This code assist feature is available in your project because the Tern module `Liferay` is enabled. This plugin provides code completions for the static JavaScript object APIs available to portlets when running in Liferay Portal. To learn more about enabling Tern modules in Eclipse, refer to the Enabling Code Assist Features in Your Project tutorial.

The example below shows how you can use code assist features to easily access functions in your portlet project.

1. Open the `main.js` of your portlet and add the following function:

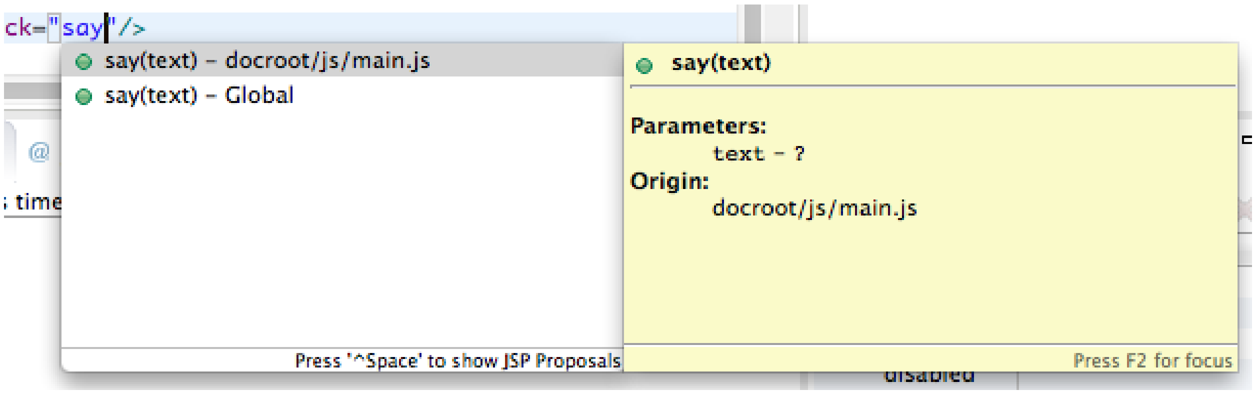
```
function say(text){
    alert(text);
}
```

2. Add the following button to the `view.jsp` of your portlet:

```
<aui:button onClick=""/>
```

3. Place your cursor within the quotation marks of the `onClick` attribute and press *Ctrl+Space*. The code inference dialog pops up with a list of possible JavaScript functions available for you to use.

4. Type *say* and you'll notice the list is narrowed down to your new `say(text)` function.


ck="say"/>
say(text) - docroot/js/main.js
say(text) - Global
Press '^Space' to show JSP Proposals

say(text)
Parameters:
text - ?
Origin:
docroot/js/main.js
Press F2 for focus

Figure 52.49: JavaScript code assist features give easy access to your functions.

5. Select the `say(text)` function, and you'll notice that it's accompanied by documentation that provides the parameter for the function, as well as the file path where the function is located.

6. Press *Enter* to use code completion and add the function to your button.

As you can see, JavaScript development is a breeze using @ide@'s code assist features. Now that you know how to use the AlloyUI and JavaScript code assist features, you can learn how to use the CSS code assist features next.

CSS Code Assist Features

@ide@ offers code inference and completion tools for CSS. In order to use these tools, you'll need to install an additional plugin.

Note: The plugin described below is planned to be bundled with Liferay @ide@ in the near future. Initial tests of the plugin revealed performance issues in some cases, which is why it is not yet a part of Liferay @ide@. Problems were not consistent, so you may have no issues installing the plugin, but we wanted to give full disclosure about it.

Follow the steps below to install the plugin in @ide@:

1. Go to *Help* → *Install New Software...*.

2. Paste the following link into the `Work with:` input field:

```
http://oss.opensagres.fr/eclipse-wtp-webresources/1.1.0/
```

3. Click *Add...* and check the box next to *WTP HTML - Web Resources*.

4. Click *Next* and follow the installation instructions.

Now that your plugin is installed, you'll need to enable the CSS features in your project. Right-click your project and go to *Properties* → *Web Resources* → *CSS*. Check both boxes to enable CSS features in your project.

You have successfully installed and enabled the new CSS features in your project!

Now that you have the CSS features enabled, you'll find out how to use them next. Follow the steps below to use the CSS code assist features to locate a CSS class. Note that the process below can also be used to locate an ID.

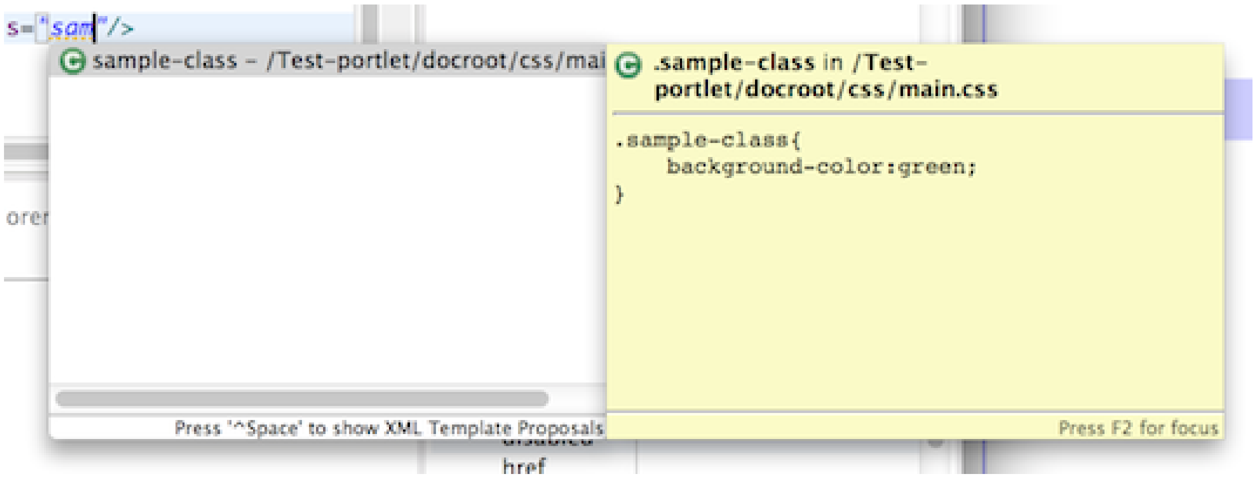
1. Open your `main.css` file and add the following class to it:

   ```
   .sample-class {
       background-color:green;
   }
   ```

2. Inside your `view.jsp` add an `<aui:button/>` tag and configure it to match the following code:

   ```
   <aui:button name="test" value="test" cssClass=""/>
   ```

3. Within the quotations of the `cssClass`×attribute, press *Ctrl+Space* to bring up the code inference for CSS. Begin typing *sample-class* to narrow down the classes to the one you're looking for.



```
s="sam"/>
```

sample-class - /Test-portlet/docroot/css/mai

.sample-class in /Test-portlet/docroot/css/main.css

```
.sample-class{
    background-color:green;
}
```

Figure 52.50: CSS code inference improves your workflow when developing in CSS.

   Notice, along with code inference, you can also view the styling you have for the class, as well as the file in which it is located.

4. Press *Enter* to use code completion and add the CSS class to the JSP.

If you look at the code inference dialog for CSS classes, you'll also notice that in addition to your own CSS classes, you also have access to Bootstrap CSS classes found in Liferay Portal.

Note: You can go to the file that the class, ID, or function is located in by hovering over top of it in your JSP and holding down the `Ctrl` (Windows) or `command` (Mac) key, and clicking the hyperlink that appears.

Lastly, you'll learn about the code assist features for jQuery.

jQuery Code Assist Features

You can also use code assist with jQuery. To do this, you must enable the `jQuery` Tern module. Follow the instructions in the Enabling Code Assist Features in Your Project tutorial to learn how to enable Tern modules in your project.

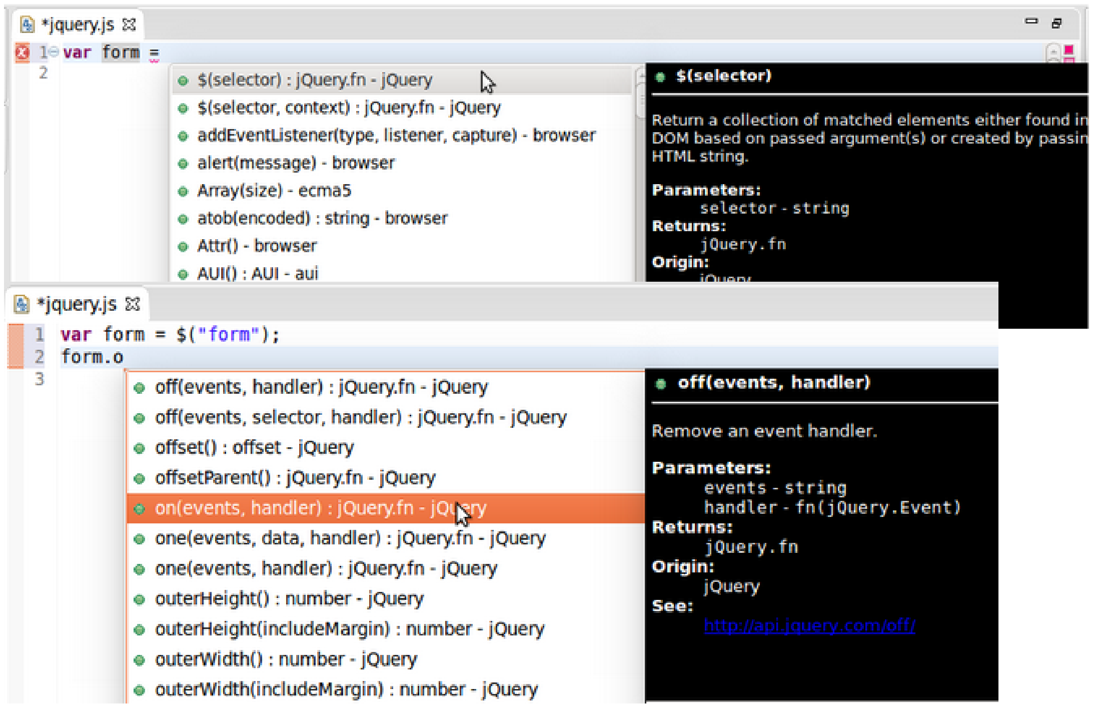
The jQuery Tern plugin gives type information for the jQuery framework. In the example below, you'll test the jQuery code assist feature.

1. Open your project's `jquery.js` file.

2. In the file, type the following sample variable:

```
var form =
```

3. Press *Ctrl+Space* to bring up the code inference for the variable you're declaring, and you'll see a list of everything that is available. Also notice jQuery documentation is available for each method. Take a look at the figure below for an example of using code assit in jQuery.



*jquery.js

```
var form =
```

- $(selector) : jQuery.fn - jQuery
- $(selector, context) : jQuery.fn - jQuery
- addEventListener(type, listener, capture) - browser
- alert(message) - browser
- Array(size) - ecma5
- atob(encoded) : string - browser
- Attr() - browser
- AUI() : AUI - aui

$(selector)

Return a collection of matched elements either found in DOM based on passed argument(s) or created by passin HTML string.

Parameters:
`selector - string`
Returns:
`jQuery.fn`
Origin:

*jquery.js

```
var form = $("form");
form.o
```

- off(events, handler) : jQuery.fn - jQuery
- off(events, selector, handler) : jQuery.fn - jQuery
- offset() : offset - jQuery
- offsetParent() : jQuery.fn - jQuery
- on(events, handler) : jQuery.fn - jQuery
- one(events, data, handler) : jQuery.fn - jQuery
- one(events, handler) : jQuery.fn - jQuery
- outerHeight() : number - jQuery
- outerHeight(includeMargin) : number - jQuery
- outerWidth() : number - jQuery
- outerWidth(includeMargin) : number - jQuery

off(events, handler)

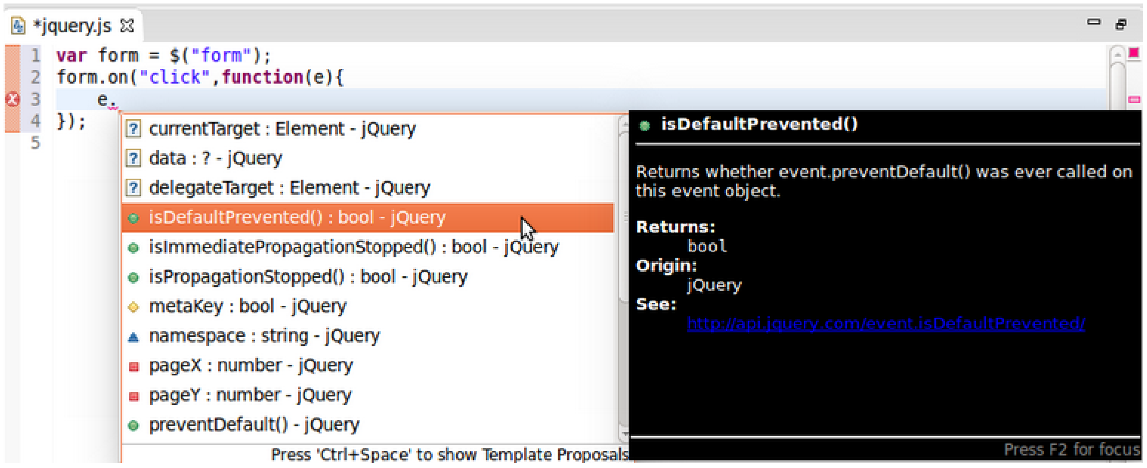
Remove an event handler.

Parameters:
`events - string`
`handler - fn(jQuery.Event)`
Returns:
`jQuery.fn`
Origin:
jQuery
See:
http://api.jquery.com/off/

Figure 52.51: Using the jQuery code assist features gives you the convenience of showing you what's available, and the documentation behind each option.

Furthermore, for jQuery callback handlers, the type information for parameters is also made available.

Excellent! You now know how to use @ide@'s front-end development code assist features to improve your workflow.


*jquery.js

```
var form = $("form");
form.on("click",function(e){
    e.
});
```

- currentTarget : Element - jQuery
- data : ? - jQuery
- delegateTarget : Element - jQuery
- isDefaultPrevented() : bool - jQuery
- isImmediatePropagationStopped() : bool - jQuery
- isPropagationStopped() : bool - jQuery
- metaKey : bool - jQuery
- namespace : string - jQuery
- pageX : number - jQuery
- pageY : number - jQuery
- preventDefault() - jQuery

Press 'Ctrl+Space' to show Template Proposals

isDefaultPrevented()

Returns whether event.preventDefault() was ever called on this event object.

Returns:
bool
Origin:
jQuery
See:
http://api.jquery.com/event.isDefaultPrevented/

Press F2 for focus

Figure 52.52: jQuery code assist also displays type information for parameters.

Related Topics

Enabling Code Assist Features in your Project

Liferay Workspace

From Liferay 6 to 7.0

Chapter 53

Blade CLI

The Blade CLI is the easiest way for Liferay developers to create new Liferay modules. Although the Plugins SDK is also supported, Blade CLI lets you create projects that can be used with any IDE or development environment. Blade CLI is a command line tool bootstrapped on to a Gradle based environment that is used to build Liferay 7.0 modules. This tool set provides a host of sub-commands that help Liferay developers create and deploy modules to a Liferay instance. The following sub-commands are callable in the Blade CLI environment:

- *convert*: Converts a Plugins SDK plugin project to a Gradle Workspace project.
- *create*: Creates a new Liferay module project from available templates.
- *deploy*: Builds and deploys bundles to the Liferay module framework.
- *gw*: Executes Gradle command using the Gradle Wrapper, if detected.
- *help*: Gives help on a specific command.
- *init*: Initializes a new Liferay Workspace.
- *install*: Installs a bundle into Liferay's module framework.
- *open*: Opens or imports a file or project in Liferay @ide@.
- *samples*: Generates a sample project.
- *server*: Starts or stops server defined by your Liferay project.
- *sh*: Connects to Liferay and executes Gogo command and returns output.
- *update*: Updates Blade CLI to latest version.
- *upgradeProps:* Analyzes your old `portal-ext.properties` and your newly installed 7.x server to show you properties moved to OSGi configuration files or removed from the product.
- *version*: Displays version information about Blade CLI.

In this set of tutorials, you'll learn how to use these commands to create and test Liferay modules.

53.1 Installing Blade CLI

You can install Blade CLI using the Liferay Project SDK installer. This installs JPM and Blade CLI into your user home folder and optionally initializes a Liferay Workspace folder.

Note: In the past, if you've installed Blade CLI globally (e.g., using `sudo`), you should not run the installer to *update* your Blade CLI version. Since the installer only installs Blade CLI to your user home folder, your previous global installation would always override the installer's installation. Therefore, always follow the Updating Blade CLI tutorial to update your Blade CLI instance.

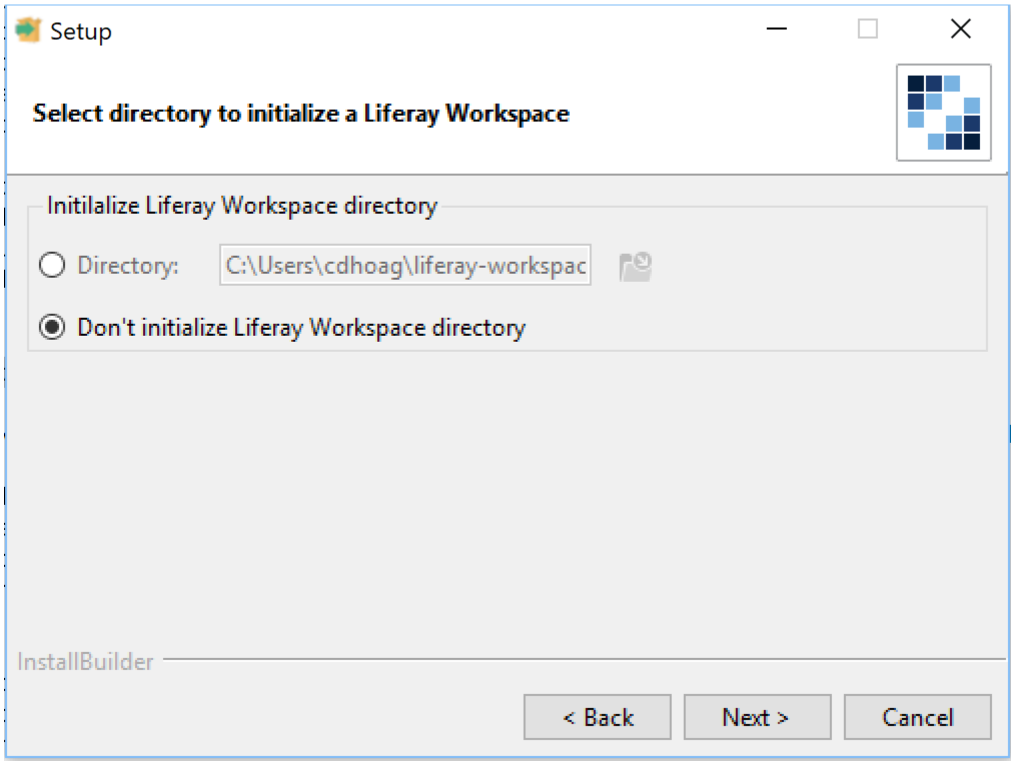
If you need to configure proxy settings for Blade CLI, follow the Installing Blade CLI with Proxy Requirements

Follow the steps below to download and install Blade CLI:

1. Download the latest Liferay Project SDK installer that corresponds with your operating system (e.g., Windows, MacOS, or Linux). The Project SDK installer is listed under *Liferay IDE*, so the folder versions are based on IDE releases. You can select an installer that does not include @ide@, if you don't intend to use it. The Project SDK installer is available for versions 3.2.0+. Do **not** select the large green download button; this downloads Liferay Portal instead.

2. Run the installer. Click *Next* to step through the installer's introduction.

3. If you'd like to initialize a Liferay Workspace, you can set the directory where it should go.



Setup

Select directory to initialize a Liferay Workspace

Initilalize Liferay Workspace directory

Directory: C:\Users\cdhoag\liferay-workspac

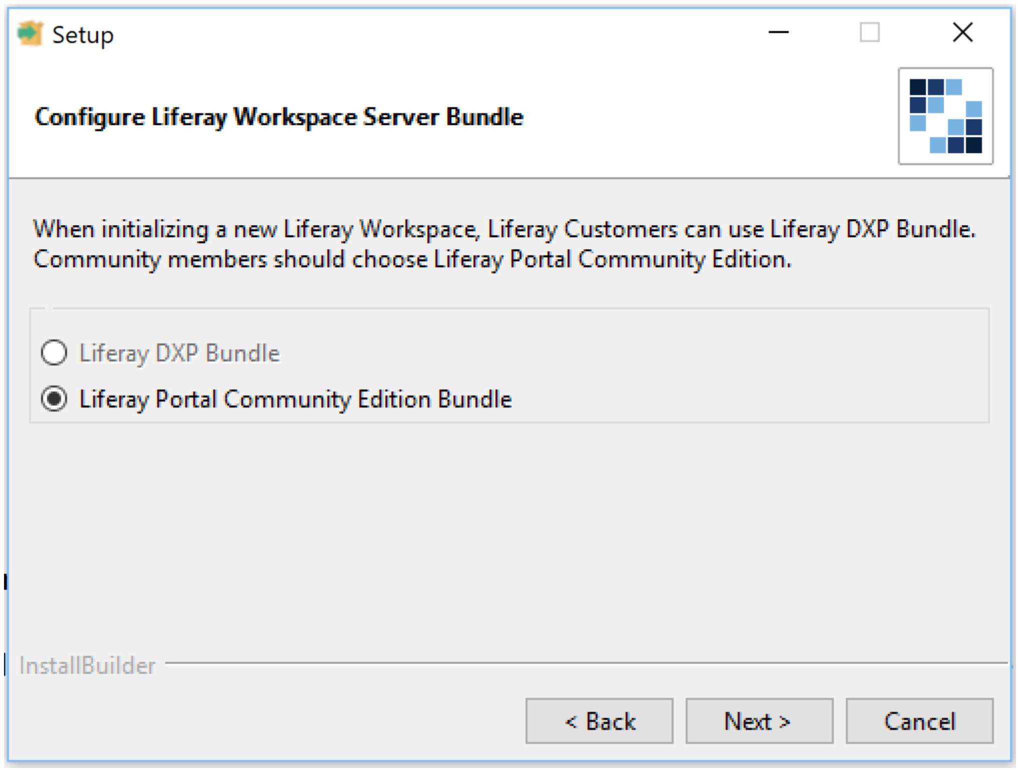
[✓ Don't initialize Liferay Workspace directory]

Figure 53.1: Determine where your Liferay Workspace should reside, if you want one.

Select the *Don't initialize Liferay Workspace directory* option if you only want to install Blade CLI. Then click *Next*.

4. If you decided to initialize a Liferay Workspace folder in the previous step, you'll have an additional option to select the Liferay product type you'll use with your workspace. Choose the product type and click *Next*.

5. Click *Next* to begin installing Blade CLI/Liferay Workspace on your computer.


Setup

Configure Liferay Workspace Server Bundle

When initializing a new Liferay Workspace, Liferay Customers can use Liferay DXP Bundle. Community members should choose Liferay Portal Community Edition.

○ Liferay DXP Bundle

● Liferay Portal Community Edition Bundle

Figure 53.2: Select the product version you'll use with your Liferay Workspace.

That's it! Blade CLI is installed on your machine! If you specified a location to initialize a Liferay Workspace folder, that is also available.

Blade CLI offers many `create` templates to help build 7.0 applications. It also offers various ways to deploy those apps and interact with your Liferay server. Be sure to explore more Blade CLI tutorials to learn how.

Installer Issues on macOS

If you're using macOS or Linux, you could experience an issue where the `blade` command is not available via command line. This is caused by the installer being unable to add JPM's `bin` folder to your user path. JPM is a Java package manager used in Blade CLI.

To add the required `bin` folder, execute the appropriate command based on your operating system.

macOS:

```
echo 'export PATH="$PATH:$HOME/Library/PackageManager/bin"' >> ~/.bash_profile
```

Linux:

```
echo 'export PATH="$PATH:$HOME/jpm/bin"' >> ~/.bash_profile
```

Once you restart the command line, the `blade` command should be available.

53.2 Installing Blade CLI with Proxy Requirements

If you have proxy server requirements and want to use Blade CLI, you must configure your http(s) proxy for it using JPM. Before beginning, make sure you've installed JPM and Blade CLI using a Liferay Workspace installer. Read the Installing Blade CLI tutorial for more details.

Once Blade CLI and JPM are installed, execute the following command to configure your proxy requirements for Blade CLI:

```
jpm command --jvmargs "-Dhttp(s).proxyHost=[your proxy host] -Dhttp(s).proxyPort=[your proxy port]" jpm
```

Excellent! You've configured Blade CLI with your proxy settings using JPM.

Note: When executing `blade update`, your Blade CLI's proxy settings are sometimes reset. Be sure to verify your proxy settings after every Blade CLI update.

53.3 Creating a Liferay Workspace with Blade CLI

In this tutorial, you'll learn how to generate a Liferay Workspace using Blade CLI. The Blade CLI tool you installed in the Installing Blade CLI section provides many different commands to help build and customize Liferay projects. The first thing you should do before building and customizing projects is create a Liferay Workspace. The workspace generated by Blade CLI is Gradle based; if you'd like to generate a Liferay Workspace built with Maven, see the Maven Workspace tutorial.

Your workspace is the home for all your custom Liferay projects. Navigate to the folder where you want your workspace and run the following command:

```
blade init -v 7.0 [WORKSPACE_NAME]
```

Note: Workspace automatically sets the default Liferay DXP version to develop against when it's first initialized. The default version is set to `7.2`. When adding the `-v 7.0` param to Blade's init command, the version is set for `7.0` Liferay DXP development. This is applied to create projects using appropriately versioned project templates.

You can update the default version after it has been set by opening your workspace's `.blade.properties` file and setting the `liferay.version.default` property.

Initializing a workspace requires no downloading or access to the internet.

If you still plan on using a Plugins SDK and wish to use it in conjunction with a workspace, navigate to your Plugins SDK root folder and run the following command:

```
blade init -u
```

This command builds a workspace and automatically adds and configures your current Plugins SDK environment for use inside the workspace. See the Using a Plugins SDK From Your Workspace section for more information on how to use a Plugins SDK from within a workspace.

Once your workspace is generated, look at its folder structure. Several folders and build/properties files were autogenerated:

- `configs`
- `gradle`

- `modules`
- `themes`
- `build.gradle`
- `gradle-local.properties`
- `gradle.properties`
- `gradlew`
- `settings.gradle`

The build/properties files included in your workspace's root directory sets your workspace's Gradle properties and facilitates the build processes of your modules. You can learn more about these generated files/folders in the Liferay Workspace tutorial. You'll learn about how to use these folders and properties files throughout the next few tutorials.

Next you'll learn about generating and using a Liferay DXP instance from within your workspace.

Running a Liferay Instance from Your Workspace

As discussed in the Liferay Workspace tutorial, Liferay Workspaces can generate and hold a Liferay Server. This lets you build/test your plugins against a running Liferay instance. Once you've properly generated and installed a Liferay server in your workspace, you can begin using it with the Blade CLI. To start your Liferay instance, run

```
blade server start -b
```

This command starts your Liferay server in a separate window. You also have the option to run your server in debug mode (`-d`).

Awesome! You have a built-in Liferay server in your workspace and can start the server using Blade CLI. Next you'll learn how to use a legacy Plugins SDK from your workspace.

Configuring a Plugins SDK in Your Workspace

Because Liferay DXP 7.0 uses a module-based framework, the current structure of a Liferay Workspace is centered around module development. There are still, however, many situations where you must create WAR-style plugins using the Plugins SDK. Because of this, your workspace can also work with the Plugins SDK. When configuring your SDK in a workspace, you can take advantage of all the new functionality workspaces provide and also use the SDK environment that you're used to.

Running the `blade init -u` command converted the Plugins SDK to a workspace that includes the Plugins SDK. If you created your workspace from scratch instead with `blade init`, you'll need to configure your Liferay workspace's Gradle properties.

If you revisit your workspace's `gradle.properties` file, you'll notice the Plugins SDK folder is set to `plugins-sdk`. This folder was not generated by default, so you must create it yourself. In your workspace's root folder, create the `plugins-sdk` folder. Then copy your legacy Plugins SDK files into the `plugins-sdk` folder. Lastly, generate its requirements by running `gradlew tasks`. Once this command successfully downloads all your Plugins SDK dependencies, it's ready to use in your workspace. For more information on manually configuring a Plugins SDK, see the Using a Plugins SDK from Your Workspace section.

53.4 Creating Projects with Blade CLI

When you use Blade CLI to create a project, your project's folder structure, build script (e.g., `build.gradle`), Java classes, and other resources (such as JSPs) are created based on the chosen template. In this tutorial, you'll learn how to use Blade CLI to create modules based on pre-existing templates and samples.

Using Blade CLI gives you the flexibility to choose how you want to create your application. You can do so in your own standalone environment, or within a Liferay Workspace. You can also create a project using either the Gradle or Maven build tool. Creating Liferay modules in a workspace using Blade CLI is very similar to creating them in a standalone environment.

When creating projects in a workspace, you should navigate to the appropriate folder corresponding to that type of project (e.g., the `/modules` folder for a module project). You can also provide further directory nesting into that folder, if preferred. For example, the Gradle workspace, by default, sets the directory where your modules should be stored by setting the following property in the workspace's `gradle.properties` file:

```
liferay.workspace.modules.dir=modules
```

Change this property if you'd like to store your modules in a different directory.

Note: Your projects should define a repository where external dependencies can be downloaded. Unlike Maven, Gradle does not define any repositories by default. For convenience, Gradle projects created with Blade CLI define Liferay's public Nexus repository as its default repository. This is defined, however, in different files depending on where the project was created.

If you used Blade CLI to create a Gradle project outside of a workspace, your repository is defined in the module's `build.gradle` file. Likewise, if you created your module inside a workspace, your repository is defined in the `settings.gradle` file located in the workspace's ROOT folder. This ensures that all modules residing in the workspace share the same repository URL.

First, you'll learn how to create a module using a template.

Project Templates

To create a new Liferay project, you can run the Blade `create` command, which offers many available templates. There are, however, many other options you can specify to help mold your project just the way you want it. To learn how to use the Blade `create` command and the many options it provides, enter `blade help create` into a terminal. A list of the `create` options are listed below:

- `-b, --build <string>`: The build type of the project. Available options are `gradle` (default) and `maven`.
- `-c, --classname <string>`: If a class is generated in the project, provide the name of the class to be generated. If not provided, the class name defaults to the project name.
- `C, --contributorType <string>`: Identifies your module as a theme contributor. This is also used to add the `Liferay-Theme-Contributor-Type` and `Web-ContextPath` bundle headers to the BND file.
- `-d, --dir <file>`: The directory to create the new project.
- `-h, --hostbundlebsn <string>`: If a new JSP hook fragment needs to be created, provide the name of the host bundle symbolic name.
- `-H, --hostbundleversion <string>`: If a new JSP hook fragment needs to be created, provide the name of the host bundle version.
- `-l, --listtemplates`: Prints a list of available project templates.
- `-p, --packagename <string>`: The package name to use when creating the project.
- `-s, --service <string>`: If a new Declarative Services (DS) component needs to be created, provide the name of the service to be implemented. Note that in this context, the term *service* refers to an OSGi service, not to a Liferay API.
- `-t, --template <string>`: The project template to use when creating the project. Run `blade create -l` for a listing of available Blade CLI templates.
- `-v, --liferay-version`: The Liferay DXP version to target when creating a project (e.g., `7.0`).

To create a module project, use the following syntax:

```
blade create [OPTIONS] <NAME>
```

For example, if you wanted to create an MVC portlet project with Gradle, you could execute the following:

```
blade create -t mvc-portlet -p com.liferay.docs.guestbook -c GuestbookPortlet my-guestbook-project
```

This command creates an MVC portlet project based on the template `mvc-portlet`. It uses the package name `com.liferay.docs.guestbook` and creates the portlet class `GuestbookPortlet`. The project name is `my-guestbook-project`. Since the directory was not specified, it is created in the folder you executed the command. When generating a project using Blade CLI, there is no downloading, which means internet access is not required.

Blade CLI can also create the same project with Maven by specifying the `-b maven` parameter. Using Blade CLI's Maven option isn't the only way leverage Liferay's Maven project templates; you can also generate them using Maven archetypes. See Liferay's Project Templates articles to see how.

When using Blade CLI, you'll have to manually edit your project's component class. Blade CLI gives you the ability to specify the class's name, but all other contents of the class can only be edited after the class is created. See the Creating Modules with Liferay @ide@ tutorial for further details and important dependency information on component classes.

Now that you know the basics on creating Liferay projects using `blade create`, you can visit the Project Templates reference section to view specific `create` templates and how they work.

Next, you'll explore Liferay's provided project samples and how to generate them using Blade CLI.

Project Samples

Liferay provides many sample projects that are useful for those interested in learning best practices on structuring their projects to accomplish specific tasks in Liferay DXP. These samples can be found in the liferay-blade-samples Github repository. You can also learn more about these samples by visiting the Liferay Sample Projects article.

You can generate these samples using Blade CLI for convenience, instead of cloning the repository and manually copy/pasting them to your environment. To do this, use the following syntax:

```
blade samples <NAME>
```

For example, if you wanted to generate the portlet-ds sample, you could execute

```
blade samples ds-portlet
```

For a full listing of all the available Blade samples, run

```
blade samples
```

Awesome! Now you know the basics on creating Liferay projects with Blade CLI.

53.5 Deploying Modules with Blade CLI

Deploying modules to a Liferay server using Blade CLI is easy. To use the Blade `deploy` command, you must first have built a module to deploy. See the Creating Projects with Blade CLI tutorials for more information about creating Liferay projects. Once you've built a module, navigate to it with your CLI and execute the following command to deploy it:

```
blade deploy
```

This can be used for WAR-style projects and modules (JARs). You can also deploy all projects in a folder by running the `deploy` command from the parent folder (e.g., `[WORKSPACE_ROOT]/modules`).

If you're using Liferay Workspace, the `deploy` command deploys your project based on the build tool's deployment configuration. For example, leveraging Blade CLI in a default Gradle Liferay Workspace uses the underlying Gradle deployment configuration. The build tool's deployment configuration is found by reading the Liferay Home folder set in your workspace's `gradle.properties` or `pom.xml` file. The `deploy` command works similarly if you're working outside of workspace; the Liferay Home folder, in contrast, is set by loading the Liferay extension object (Gradle) or the effective POM (Maven) and searching for the Liferay Home property stored there. If it's not stored, Blade prompts you to set it so it's available.

Note: If you prefer using pure Gradle or Maven to deploy your project, you can do this by applying the appropriate plugin and configuring your Liferay Home property. Here's how you can do this for Gradle and Maven:

Gradle:

First ensure the Liferay Gradle plugin is applied in your `build.gradle` file:

```
apply plugin: "com.liferay.plugin"
```

Then extend the Liferay extension object to set your Liferay Home and `deploy` folder:

```
liferay {
    liferayHome = "../../../../liferay-ce-portal-7.0.1-ga2"
    deployDir = file("${liferayHome}/deploy")
}
```

Maven:

Ensure the Bundle Support plugin is applied and configure Liferay Home in your `pom.xml`. See the Deploying a Module Built with Maven to Liferay Portal for details.

If you prefer not to use your underlying build tool's (Gradle or Maven) module deployment configuration, and instead, you want to deploy straight to Liferay DXP's OSGi container, run this command instead:

```
blade deploy -l
```

Blade CLI also offers a way to *watch* a deployed project, which compiles and redeploys a project when changes are detected. There are two ways to do this:

- `blade watch`
- `blade deploy -w`

The `blade watch` command is the fastest way to develop and test module changes, because the `watch` command does not rebuild your project every time a change is detected. When running `blade watch`, your project is not copied to Portal, but rather, is installed into the runtime as a reference. This means that the Portal does not make a cached copy of the project. This allows the Portal to see changes that are made to your project's files immediately. When you cancel the `watch` task, your module is uninstalled automatically.

Note: The `blade watch` command is available for Liferay Workspace versions 1.10.9+ (i.e., the `com.liferay.gradle.plugins.workspace` dependency). Maven projects cannot leverage the `watch` feature at this time.

The `blade deploy -w` command works similarly to `blade watch`, except it manually recompiles and deploys your project every time a change is detected. This causes slower update times, but does preserve your deployed project in Portal when it's shut down.

Cool! You've successfully deployed your module project using Blade CLI.

53.6 Managing Your Liferay Server with Blade CLI

In this tutorial, you'll learn how to manage a Liferay server using Blade CLI. For example, Blade CLI lets you install, start, stop, inspect, and modify a Liferay server.

Make sure you're in a Liferay Workspace and have a bundle installed and configured in the workspace before testing the Blade CLI commands on your own. To learn more about installing a Liferay server in a Liferay Workspace, see the Creating a Liferay Workspace with Liferay @ide@ section. The following Blade CLI commands are covered in this sub-section:

- `server`
- `sh`

The first thing that comes to mind when interacting with a server is simply turning it on/off. You can use the `server` sub-command to accomplish this. To turn on a Liferay server (Tomcat or Wildfly/JBoss), you can run

```
blade server start -b
```

Likewise, to turn off a server, run

```
blade server stop
```

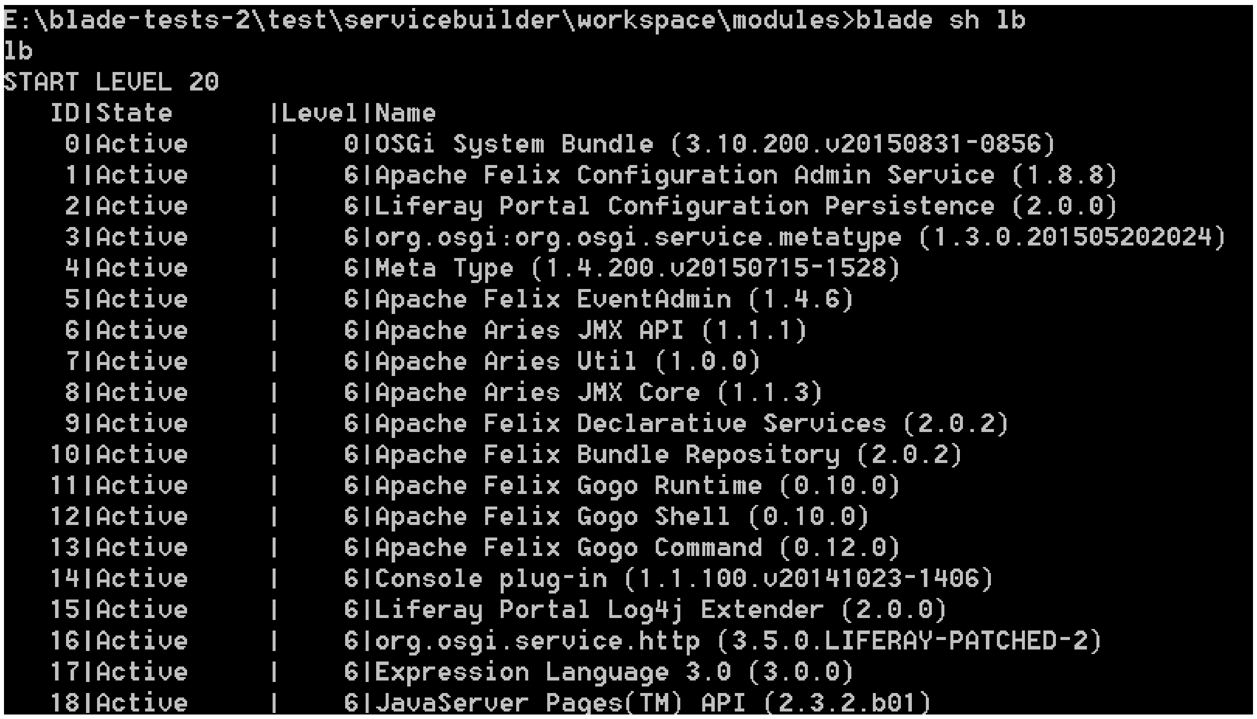
Once you've started your bundle, you can examine your server's OSGi container by using the `sh` command, which provides access to your server using the Felix Gogo shell. For example, to check if you successfully deployed your application from the previous section, you could run:

```
blade sh lb
```

Your output lists a long list of modules that are active/installed in your server's OSGi container.

You can run any Gogo command using `blade sh`. See the Using the Felix Gogo Shell section for more information on this tool.

Awesome! You learned how to conveniently interact with Liferay DXP using Blade CLI.



```
E:\blade-tests-2\test\servicebuilder\workspace\modules>blade sh lb
lb
START LEVEL 20
   ID|State      |Level|Name
    0|Active     |    0|OSGi System Bundle (3.10.200.v20150831-0856)
    1|Active     |    6|Apache Felix Configuration Admin Service (1.8.8)
    2|Active     |    6|Liferay Portal Configuration Persistence (2.0.0)
    3|Active     |    6|org.osgi:org.osgi.service.metatype (1.3.0.201505202024)
    4|Active     |    6|Meta Type (1.4.200.v20150715-1528)
    5|Active     |    6|Apache Felix EventAdmin (1.4.6)
    6|Active     |    6|Apache Aries JMX API (1.1.1)
    7|Active     |    6|Apache Aries Util (1.0.0)
    8|Active     |    6|Apache Aries JMX Core (1.1.3)
    9|Active     |    6|Apache Felix Declarative Services (2.0.2)
   10|Active     |    6|Apache Felix Bundle Repository (2.0.2)
   11|Active     |    6|Apache Felix Gogo Runtime (0.10.0)
   12|Active     |    6|Apache Felix Gogo Shell (0.10.0)
   13|Active     |    6|Apache Felix Gogo Command (0.12.0)
   14|Active     |    6|Console plug-in (1.1.100.v20141023-1406)
   15|Active     |    6|Liferay Portal Log4j Extender (2.0.0)
   16|Active     |    6|org.osgi.service.http (3.5.0.LIFERAY-PATCHED-2)
   17|Active     |    6|Expression Language 3.0 (3.0.0)
   18|Active     |    6|JavaServer Pages(TM) API (2.3.2.b01)
```

Figure 53.3: Blade CLI accesses the Gogo shell script to run the lb command.

53.7 Updating Blade CLI

If your Blade CLI version is outdated, you can run the following command to automatically download and install the latest version of Blade CLI:

```
blade update
```

For Windows users, the `blade update` command does not work because Windows cannot update a file that is currently in use. To bypass this issue, you can use JPM to update your version of Blade CLI:

```
jpm install -f https://releases.liferay.com/tools/blade-cli/latest/blade.jar
```

Blade CLI is updated frequently, so it's recommended to update your Blade CLI environment for new features. You can check the released versions of Blade CLI at https://releases.liferay.com/tools/blade-cli/. You can check your current installed version by running `blade version`.

Note: If you run `blade version` after updating, but don't see the expected version installed, you may have two separate Blade CLI installations on your machine. This is typically caused by users who installed an earlier version of Blade CLI, and then used the Liferay Workspace installer (at any time prior) to update the older Blade CLI instance. This is not recommended. Doing this installs Blade CLI in the global and user home folder of your machine. The latest Blade CLI update process installs to your user home folder, so you must delete the legacy Blade files in your global folder, if present. To do this, navigate to your `GLOBAL_FOLDER/JPM4J` folder and delete

- `/bin/blade`

- `/commands/blade`

The newest Blade CLI installation in your user home folder is now recognized and available.

Although Blade CLI is frequently released, if you want bleeding edge features not yet available, you can install the latest snapshot version:

```
blade update -s
```

This pulls the latest snapshot version of Blade CLI and installs it to your local machine. Running `blade version` after installing a snapshot displays output similar to this:

```
blade version 3.3.1.SNAPSHOT201811301746
```

Be careful; snapshot versions are unstable and should only be used for experimental purposes.

Awesome! You've successfully learned how to update Blade CLI.

53.8 Converting Plugins SDK Projects with Blade CLI

Blade CLI can automatically migrate a Plugins SDK project to a Liferay Workspace. During the process, the Ant-based Plugins SDK project is copied to the applicable workspace folder based on its project type (e.g., `wars`) and is converted to a Gradle-based Liferay Workspace project. This drastically speeds up the migration process when upgrading to a Liferay Workspace from a legacy Plugins SDK.

Note: There is no Maven command for the migration process yet, so you must complete it manually for Maven-based workspaces.

To copy your Plugins SDK project and convert it to Gradle, use the Blade `convert` command:

1. Navigate to the root folder of your workspace in a command line tool.

2. Execute the following command:

   ```
   blade convert -s [PLUGINS_SDK_PATH] [PLUGINS_SDK_PROJECT_NAME]
   ```

   You must provide the path of the Plugins SDK your project resides in and the project name you want to convert. If you prefer converting all the Plugins SDK projects at once, replace the project name variable with `-a` (i.e., specifying all plugins).

```
**Note:** If the `convert` task doesn't work as described above, you may
need to update your Blade CLI version. See the
[Updating Blade CLI](/docs/7-0/tutorials/-/knowledge_base/t/updating-blade-cli)
article for more information.
```

```
This Gradle conversion process also works for themes; they're converted to
automatically leverage NodeJS. If you're converting a Java-based theme, add
the `-t` option to your command too. This will incorporate the
[Theme Builder](/docs/reference/7-0/-/knowledge_base/r/theme-builder-gradle-plugin)
Gradle plugin for the theme instead. For more information on upgrading
6.2 themes, see the
[Upgrade a 6.2 Theme to 7.0](/docs/7-0/tutorials/-/knowledge_base/t/upgrading-themes)
article.
```

Note: When converting a Service Builder project, the `convert` task automatically extracts the project's service interfaces and implementations into OSGi modules (i.e., *-impl and -*api) and places them in the workspace's `modules` folder. Your portlet and controller logic remain a WAR and reside in the `wars` folder.

Your project is successfully converted to a Gradle-based workspace project! Great job!

Chapter 54

Liferay Workspace

A *Liferay Workspace* is a generated environment that is built to hold and manage your Liferay projects. This workspace is intended to aid in the management of Liferay projects by providing various Gradle build scripts and configured properties. This is the official way to create 7.0 modules using Gradle. For those developers that still want to develop WAR-style plugins using the Plugins SDK, this way is also supported using a Liferay Workspace. Do you prefer Maven over Gradle? See the Maven Workspace tutorial to learn about using Liferay Workspace with Maven.

Liferay Workspaces can be used in many different development environments, which makes it flexible and applicable to many different developers. You can download the Liferay Workspace installer and run it to install Blade CLI (default CLI for workspace) and initialize a new Liferay Workspace.

You can also use it with other developer IDEs. For example, a Liferay Workspace easily integrates with Liferay @ide@, providing a seamless development experience. To learn more about Liferay @ide@ and using workspace with it, see the Creating a Liferay Workspace with Liferay @ide@ tutorial.

Your workspace also offers Gradle properties that you can modify to help manage the generated folders. There are also some folders that aren't generated by default, but can be manually created and set. This provides you the power to customize your workspace's folder structure any way you'd like. To learn more info on a workspace's folder structure and how you can configure a workspace, see the Configuring a Liferay Workspace tutorial.

Liferay Workspaces offer a full development lifecycle for your modules to make your Liferay development easier than ever. The development lifecycle includes creating, building, deploying, testing, and releasing modules. To learn more about the development lifecycle of a Liferay Workspace, see the Development Lifecycle for a Liferay Workspace tutorial.

54.1 Installing Liferay Workspace

You can install Liferay Workspace using the Liferay Project SDK installer. This installs JPM and Blade CLI into your user home folder and optionally initializes a Liferay Workspace folder. This is the same installer used to install Blade CLI, which is covered in the Installing Blade CLI tutorial.

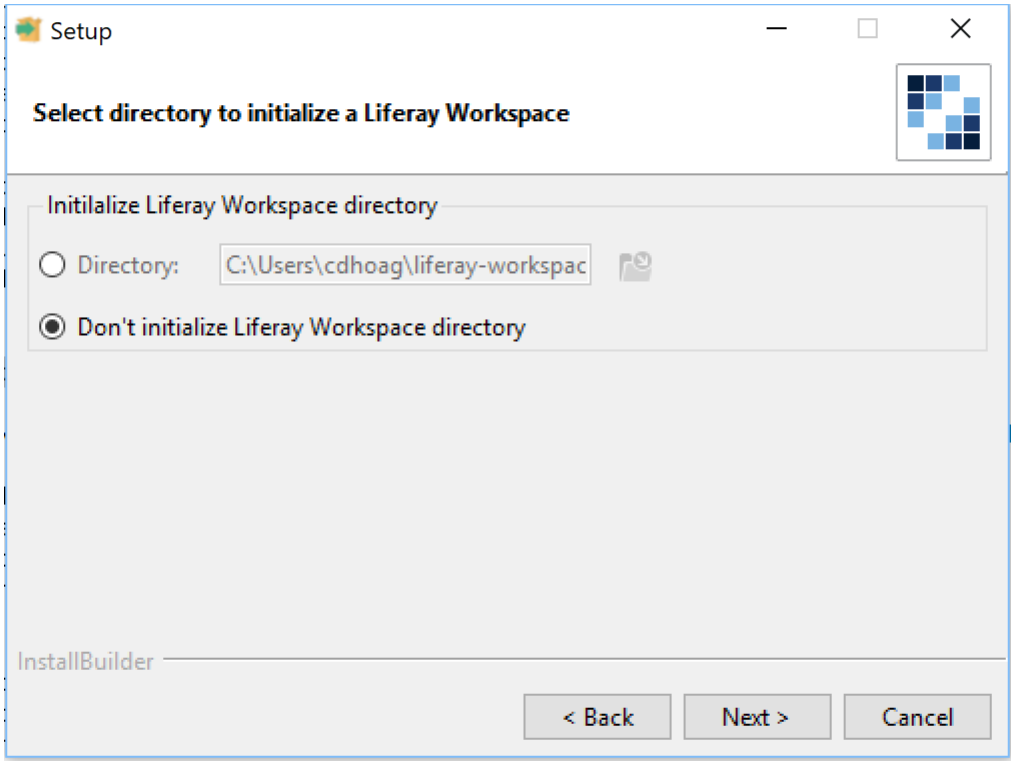
Follow the steps below to download and install Liferay Workspace:

1. Download the latest Liferay Project SDK installer that corresponds with your operating system (e.g., Windows, MacOS, or Linux). The Project SDK installer is listed under *Liferay IDE*, so the folder versions are based on IDE releases. You can select an installer that does not include @ide@, if you don't intend

to use it. The Project SDK installer is available for versions 3.2.0+. Do **not** select the large green download button; this downloads Liferay Portal instead.

2. Run the installer. Click *Next* to step through the installer's introduction.

3. Set the directory where your Liferay Workspace should be initialized.



Setup

Select directory to initialize a Liferay Workspace

Initilalize Liferay Workspace directory

Directory: C:\Users\cdhoag\liferay-workspac

[✓ Don't initialize Liferay Workspace directory]

Figure 54.1: Determine where your Liferay Workspace should reside.

Then click *Next*.

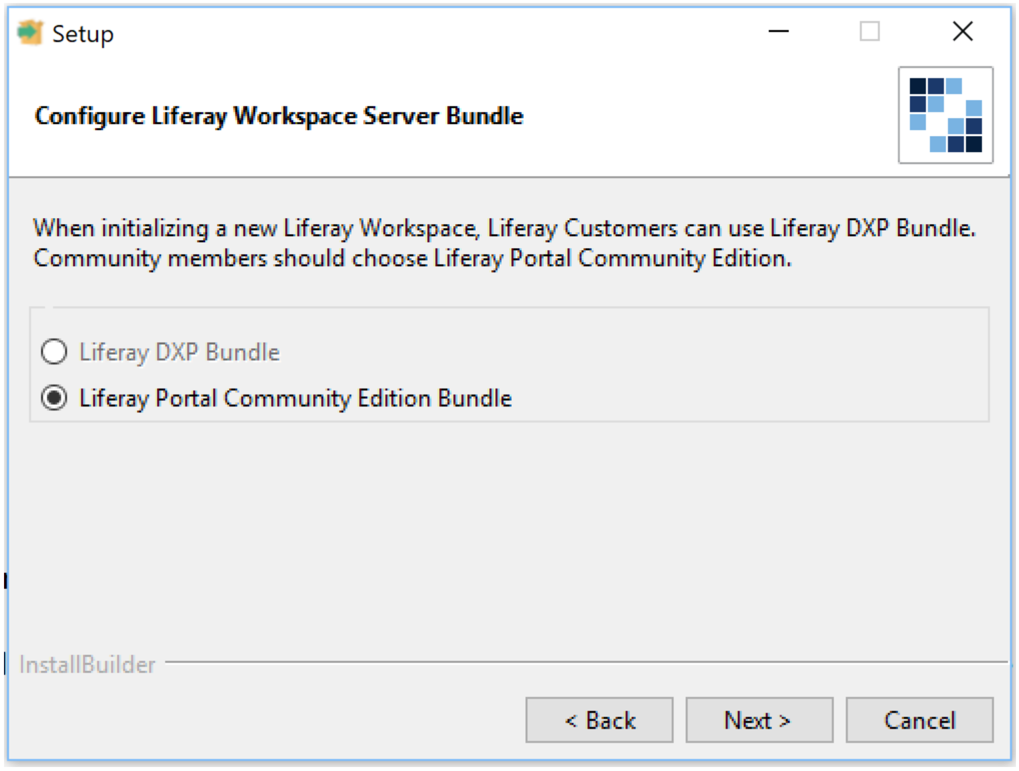
4. Choose the Liferay product type you intend to use with the workspace. Then click *Next*.

```
**Note:** You'll be prompted for your liferay.com username and password
before downloading the Liferay DXP bundle. Your credentials are not saved
locally; they're saved as a token in the `~/.liferay` folder. The token is
used by your workspace if you ever decide to redownload a DXP bundle.
Furthermore, the bundle that is downloaded in your workspace is also copied
to your `~/.liferay/bundles` folder, so if you decide to initialize another
Liferay DXP instance of the same version, the bundle is not re-downloaded. See
the
[Adding a Liferay Bundle to a Workspace](/docs/7-0/tutorials/-/knowledge_base/t/configuring-a-liferay-workspace#adding-a-liferay-
bundle-to-a-workspace)
for more information on this topic.
```

---

5. Click *Next* to begin installing Liferay Workspace on your machine.

That's it! Liferay Workspace is now installed on your machine!


Setup

Configure Liferay Workspace Server Bundle

When initializing a new Liferay Workspace, Liferay Customers can use Liferay DXP Bundle. Community members should choose Liferay Portal Community Edition.

( ) Liferay DXP Bundle
(●) Liferay Portal Community Edition Bundle

Figure 54.2: Select the product version you'll use with your Liferay Workspace.

54.2 Configuring a Liferay Workspace

A Liferay Workspace offers a development environment that can be configured to fit your development needs. You'll learn about the files/folders a workspace provides by default, and then you'll dive into configuring your workspace.

The top-level files/folder of a Liferay workspace are outlined below:

- `bundles` (generated): the default folder for Liferay DXP bundles.
- `configs`: holds the configuration files for different environments. These files serve as your global configuration files for all Liferay servers and projects residing in your workspace. To learn more about using the `configs` folder, see the Testing Modules section.
- `gradle`: holds the Gradle Wrapper used by your workspace.
- `modules`: holds your custom modules.
- `plugins-sdk` (generated): holds plugins to migrate from previous releases.
- `themes`: holds your custom themes which are built using the Theme Generator.
- `wars` (generated): holds traditional WAR-style web application projects.
- `build.gradle`: the common Gradle build file.
- `gradle-local.properties`: sets user-specific properties for your workspace. This lets multiple users use a single workspace, letting them configure specific properties for the workspace on their own machine.
- `gradle.properties`: specifies the workspace's project locations and Liferay DXP server configuration globally.
- `gradlew`: executes the Gradle command wrapper
- `settings.gradle`: applies plugins to the workspace and configures its dependencies.

The build/properties files included in your workspace's root folder sets your workspace's Gradle properties and facilitates the build processes of your modules.

Before you begin using your workspace, you should set your workspace Gradle properties in the `gradle.properties` file. There are several commented out properties in this file. These are the default properties set in your workspace. If you'd like to change a variable, uncomment the variable and set it to a custom value. For instance, if you want to store your modules in a folder other than `[ROOT]/modules`, uncomment the `liferay.workspace.modules.dir` variable and set it to a different value.

If you'd like to keep the global Gradle properties the same, but want to change them for yourself only (perhaps for local testing), you can override the `gradle.properties` file with your own `gradle-local.properties` file.

Note: Liferay Workspace provides many subprojects for you behind the scenes, which hides some complexities of Gradle. You can learn more about this in the Building Modules section.

Now that you know about a workspace's default folder structure and how to modify its Gradle properties, you'll learn how to add a Liferay bundle to your workspace.

Adding a Liferay Bundle to a Workspace

Liferay Workspaces can generate and hold a Liferay Server. This lets you build/test your plugins against a running Liferay instance. Before generating a Liferay instance, open the `gradle.properties` file located in your workspace's root folder. There are several configurable properties for your workspace's Liferay instance. You can set the version of the Liferay bundle you'd like to generate and install by setting the download URL for the `liferay.workspace.bundle.url` property (e.g., `https://releases-cdn.liferay.com/portal/7.0.6-ga7/liferay-ce-portal-tomcat-7.0-ga7-20180507111753223.zip`). You can also set the folder where your Liferay bundle is generated with the `liferay.workspace.home.dir` property. It's set to `bundles` by default.

You can download a Liferay DXP bundle for your workspace if you're a DXP subscriber. Do this by setting the `liferay.workspace.bundle.url` property to a ZIP hosted on *api.liferay.com*. For example,

`liferay.workspace.bundle.url=https://api.liferay.com/downloads/portal/7.0.10.8/liferay-dxp-digital-enterprise-tomcat-7.0-sp8-20180717152749345.zip`

It can be tricky to find the fully qualified ZIP name/number for the DXP bundle you want. You cannot access Liferay's API site directly to find it, so you must start to download DXP manually, take note of the file name, and append it to `https://api.liferay.com/downloads/portal/`.

You must also set the `liferay.workspace.bundle.token.download` property to `true` to allow your workspace to access Liferay's API site.

Once you've finalized your Gradle properties, navigate to your workspace's root folder and run

```
blade server init
```

This uses workspace's pre-bundled Blade CLI tool to download the version of Liferay DXP you specified in your Gradle properties and installs your Liferay instance in the `bundles` folder.

If you want to skip the downloading process, you can create the `bundles` folder manually in your workspace's ROOT folder and unzip your Liferay DXP bundle to that folder.

You can also produce a distributable Liferay bundle (Zip or Tar) from within a workspace. To do this, navigate to your workspace's root folder and run the following command:

```
./gradlew distBundle[Zip|Tar]
```

Your distribution file is available from the workspace's `/build` folder.

Note: You can define different environments for your Liferay bundle for easy testing. You can learn more about this in the Testing Modules section.

The Liferay Workspace is a great development environment for Liferay module development; however, what if you'd like to also stick with developing WAR-style applications? Liferay Workspace can handle that request too!

Using a Plugins SDK from Your Workspace

Because 7.0 uses a module-based framework, the current structure of a Liferay Workspace is centered around module development. There are still, however, many situations where you must create WAR-style plugins using the Plugins SDK. Because of this, your workspace can also work with the Plugins SDK. When configuring your SDK in a workspace, you can take advantage of all the new functionality workspaces provide and also use the SDK environment that you're used to. To learn more about upgrading legacy applications to 7.0 and what you should consider before converting them to modules, visit the tutorial Planning Plugin Upgrades and Optimizations.

The Blade CLI offers a command that adds and configures your current Plugins SDK environment automatically for use inside a newly generated workspace (e.g., `blade init -u`). You can learn more about this in the Creating a Liferay Workspace with Blade CLI tutorial. If you created your workspace from scratch and want to use a Plugins SDK, however, you can add one to your workspace by completing one of the two options:

1. Copy your existing Plugins SDK's files into the workspace.
2. Generate a new Plugins SDK to use in the workspace.

Follow the appropriate section based on the option you want to follow.

Copying an Existing Plugins SDK into Workspace

If you open your workspace's `gradle.properties` file, you'll notice the `liferay.workspace.plugins.sdk.dir` property sets the Plugins SDK folder to `plugins-sdk`. This is where the workspace expects any Plugins SDK files to reside. This folder was not generated by default, so you must create it yourself. In your workspace's root folder, create the `plugins-sdk` folder. Then copy your legacy Plugins SDK files into the `plugins-sdk` folder.

The copied Plugins SDK requires many build-related artifacts. To start the artifact download process, execute the following command in your workspace's root folder:

```
./gradlew upgradePluginsSDK
```

The Plugins SDK's artifacts are downloaded. The Plugins SDK is now ready for use!

Generating a New Plugins SDK in Workspace

You can easily generate a new Plugins SDK for your workspace by executing a single Gradle command in your workspace's root folder:

```
./gradlew upgradePluginsSDK
```

This generates a new 7.0 Plugins SDK into the folder set by the `liferay.workspace.plugins.sdk.dir` property, which is configured to `plugins-sdk` by default in the workspace's `gradle.properties` file. You can change the folder name by updating the property. The downloaded Plugins SDK version is the latest release at the time of execution. You can reference the latest Plugins SDK releases here.

Once the downloading is complete, your Plugins SDK is ready to use in your workspace!

54.3 Setting Proxy Requirements for Liferay Workspace

If you're working behind a corporate firewall that requires using a proxy server to access external repositories, you need to add some extra configuration to make Liferay Workspace work within your environment. You'll learn how to set proxy requirements for both Gradle and Maven environments.

Using Gradle

1. Open your `~/.gradle/gradle.properties` file. Create this file if it does not exist.

2. Add the following properties to the file:

```
systemProp.http.proxyHost=www.somehost.com
systemProp.http.proxyPort=1080
systemProp.https.proxyHost=www.somehost.com
systemProp.https.proxyPort=1080
```

   Make sure to replace the proxy host and port values with your own.

3. If the proxy server requires authentication, also add the following properties:

```
systemProp.http.proxyUser=userId
systemProp.http.proxyPassword=yourPassword
systemProp.https.proxyUser=userId
systemProp.https.proxyPassword=yourPassword
```

Excellent! Your proxy settings are set in your Liferay Workspace's Gradle environment.

Using Maven

1. Open your `~/.m2/settings.xml` file. Create this file if it does not exist.

2. Add the following XML snippet to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
    <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
        <proxies>
            <proxy>
                <id>httpProxy</id>
                <active>true</active>
                <protocol>http</protocol>
                <host>www.somehost.com</host>
                <port>1080</port>
            </proxy>
            <proxy>
                <id>httpsProxy</id>
                <active>true</active>
                <protocol>https</protocol>
                <host>www.somehost.com</host>
                <port>1080</port>
            </proxy>
        </proxies>
    </settings>
```

Make sure to replace the proxy host and port values with your own.

3. If the proxy server requires authentication, also add the `username` and `password` proxy properties. For example, the HTTP proxy authentication configuration would look like this:

```
<proxy>
  <id>httpProxy</id>
  <active>true</active>
  <protocol>http</protocol>
  <host>www.somehost.com</host>
  <port>1080</port>
  <username>userID</username>
  <password>somePassword</password>
</proxy>
```

Excellent! Your Maven proxy settings are now set.

54.4 Development Lifecycle for a Liferay Workspace

Liferay Workspaces provide an environment that supports all phases of a Liferay module's development lifecycle:

- Creating modules
- Building modules
- Deploying modules
- Testing modules
- Releasing modules

In this tutorial, you'll explore the development lifecycle phases Liferay Workspace provides for you. Then you'll be directed to other tutorials that go into further detail for leveraging the workspace's particular lifecycle phase for a specific tool (e.g., Blade CLI or Liferay @ide@). Let's get started!

Creating Modules

The first step of Liferay Workspace's development phase is the module creation process. Workspace provides a slew of templates that you can use to create many different types of Liferay modules.

You can configure where your workspace creates modules by editing the `liferay.workspace.modules.dir` property in the workspace's `gradle.properties` file. By default, modules are created in the `[ROOT]/modules` folder.

You can also control where themes are generated by specifying the `liferay.workspace.themes.dir` property in the `gradle.properties` file. Themes are typically migrated to the `themes` folder after being created using the Liferay Theme Generator.

To learn more about creating modules in a workspace using Blade CLI or Liferay @ide@, visit the Creating Modules with Blade CLI and Creating Modules with Liferay @ide@ tutorials, respectively.

Building Modules

Liferay Workspace abstracts many build requirements away so you can focus on developing modules instead of worrying about how to build them. Liferay Workspace is built using Gradle, so your modules leverage the Gradle build lifecycle.

Workspace includes a Gradle wrapper in its ROOT folder (e.g., `gradlew`), which you can leverage to execute Gradle commands. This means that you can run familiar Gradle build commands (e.g., `build`, `clean`, `compile`, etc.) from a Liferay Workspace without having Gradle installed on your machine.

Note: You can also use the workspace's Gradle wrapper by executing `blade gw` followed by the Gradle command. This is an easier way to run the workspace's Gradle wrapper without specifying its path. Since the workspace's Gradle wrapper resides in its root folder, it can sometimes be a hassle running it for a deeply nested module (e.g., `../../../../gradlew compileJava`). Running the Gradle wrapper from Blade CLI automatically detects the Gradle wrapper and can run it anywhere.

When using Liferay Workspace, the workspace plugin is automatically applied which adds a multitude of subprojects for you, hiding some complexities of Gradle. For example, a typical project's `settings.gradle` file could contain many included subprojects like this:

```
...
include images:base:oracle-jdk:oracle-jdk-6
include images:base:oracle-jdk:oracle-jdk-7
include images:base:oracle-jdk:oracle-jdk-8
include images:base:liferay-portal:liferay-portal-ce-tomcat-7.0-ga1
include images:source-bundles:glassfish
include images:source-bundles:jboss-eap
include images:source-bundles:tomcat
include images:source-bundles:websphere
include images:source-bundles:wildfly
include compose:jboss-eap-mysql
include compose:tomcat-mariadb
include compose:tomcat-mysql
include compose:tomcat-mysql-elastic
include compose:tomcat-postgres
include file-server
...
```

You don't have to worry about applying these subprojects because the workspace plugin does it for you. Likewise, if a folder in the `/themes` folder includes a `liferay-theme.json` file, the `gulp` plugin is applied to it; if a folder in the `/modules` folder includes a `bnd.bnd` file, the liferay-gradle plugin is applied to it. See the Gradle reference article for a list of Liferay Gradle plugins automatically provided for all Workspace apps. As you can see, Liferay Workspace provides many plugins and build configurations behind the scenes to make your development process convenient.

A good example of the Gradle build lifecycle abstraction is the module deployment process in a workspace. You can build/deploy your modules from workspace without ever running a Gradle command. You'll learn how to do this next.

Deploying Modules

Liferay Workspace provides easy-to-use deployment mechanisms that let you deploy your module to a Liferay server without any custom configuration. To learn more about deploying modules from a workspace using Blade CLI or Liferay @ide@, visit the Deploying Modules with Blade CLI and Deploying Modules with Liferay @ide@ tutorials, respectively.

Testing Modules

Liferay provides many configuration settings for 7.0. Configuring several different Liferay DXP installations to simulate/test certain behaviors can become cumbersome and time consuming. With Liferay Workspace, you can easily organize environment settings and generate an environment installation with those settings.

Liferay Workspace provides the `configs` folder, which lets you configure different environments in the same workspace. For example, you could configure separate Liferay DXP environment settings for development, testing, and production in a single Liferay Workspace. So how does it work?

The `configs` folder offers five subfolders:

- `common`: holds a common configuration that you want applied to all environments.
- `dev`: holds the development configuration.
- `local`: holds the configuration intended for testing locally.
- `prod`: holds the configuration for a production site.
- `uat`: holds the configuration for a UAT site.

You're not limited to just these environments. You can create any subfolder in the `configs` folder (e.g., `aws`, `docker`, etc.) to simulate any environment. Each environment folder can supply its own database, `portal-ext.properties`, Elasticsearch, etc. The files in each folder overlay your Liferay DXP installation, which you generate from within workspace.

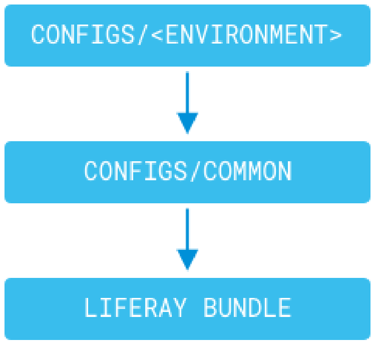


Figure 54.3: The `configs/common` and `configs/[environment]` overlay you Liferay DXP bundle when it's generated.

When workspace generates a Liferay DXP bundle, these things happen:

1. Configuration files found in the `configs/common` folder are applied to the Liferay DXP bundle.
2. The configured workspace environment (`dev`, `local`, `prod`, `uat`, etc.) is applied on top of any existing configurations from the `common` folder.

To generate a Liferay DXP bundle with a specific environment configuration to the workspace's `/bundles` folder, run

```
./gradlew initBundle -Pliferay.workspace.environment=[ENVIRONMENT]

<!-- `blade server init` is not able to pass the environment param in
currently. This new feature is requested in BLADE-343. -Cody -->
```

To generate a distributable Liferay DXP installation to the workspace's `/build` folder, run

```
./gradlew distBundle[Zip|Tar] -Pliferay.workspace.environment=[ENVIRONMENT]
```

The `ENVIRONMENT` variable should match the configuration folder (`dev`, `local`, `prod`, `uat`, etc.) you intend to apply.

Note: You may prefer to set your workspace environment in the `gradle.properties` file instead of passing it via Gradle command. If so, it's recommended to set the workspace envrionment variable inside the `[USER_HOME]/.gradle/gradle.properties` file.

```
liferay.workspace.environment=local
```

The variable is set to `local` by default.

To simulate using the `configs` folder, let's explore a typical scenario. Suppose you want a local Liferay DXP installation for testing and a UAT installation for simulating a production site. Assume you want the following configuration for the two environments:

Local Environment

- Use MySQL database pointing to localhost
- Skip setup wizard

UAT Environment

- Use MySQL database pointing to a live server
- Skip setup wizard

To configure these two environments in your workspace, follow the steps below:

1. Open the `configs/common` folder and add the `portal-setup-wizard.properties` file with the `setup.wizard.enabled=false` property.

2. Open the `configs/local` folder and configure the MySQL database settings for localhost in a `portal-ext.properties` file.

3. Open the `configs/uat` folder and configure the MySQL database settings for the live server in a `portal-ext.properties` file.

4. Now that your two environments are configured, generate one of them:

   ```
   ./gradlew distBundle[Zip|Tar] -Pliferay.workspace.environment=uat
   ```

   You've successfully configured two environments and generated one of them.

Awesome! You can now test various Liferay DXP bundle environments using Liferay Workspace.

Releasing Modules

Liferay Workspace does not provide a built-in release mechanism, but there are easy ways to use external release tools with workspace. The most popular choice is uploading your modules to a Maven Nexus repository. You could also use other release tools like Artifactory.

Uploading modules to a remote repository is useful if you need to share them with other non-workspace projects. Also, if you're ready for your modules to be in the spotlight, uploading them to a public remote repository gives other developers the chance to use them.

For more instructions on how to set up a Maven Nexus repository for your workspace's modules, see the Creating a Maven Repository and Deploying Liferay Maven Artifacts to a Repository tutorials.

54.5 Managing the Target Platform for Liferay Workspace

Liferay Workspace helps you target a specific release of Liferay DXP, so dependencies get resolved properly. This makes upgrades easy: specify your target platform, and Workspace points to the new version. All your dependencies are updated to the latest ones provided in the targeted release.

Note: There are times when configuring dependencies based on a version range is better than tracking exact versions. See the Semantic Versioning tutorial for more details.

Liferay @ide@ 3.2+ helps you streamline targeting a specific version even more. @ide@ can index the configured Liferay DXP source code to

- provide advanced Java search (Open Type and Reference Searching) (tutorial)
- debug Liferay DXP sources (tutorial)

To enable this functionality, set the following property in your workspace's `gradle.properties` file:

```
target.platform.index.sources=true
```

Note: Portal source indexing is disabled in Gradle workspace version 2.0.3+ (Target Platform plugin version 2.0.0+).

These options in @ide@ are only available when developing in a Liferay Workspace, or if you have the Target Platform Gradle plugin applied to your multi-module Gradle project with specific configurations. See the Targeting a Platform Outside of Workspace section for more info on applying the Target Platform Gradle plugin.

Next, you'll discover how all of this is possible.

Dependency Management with BOMs

You can target a version by importing a predefined bill of materials (BOM). This only requires that you specify a property in your workspace's `gradle.properties` file. You'll see how to do this later.

Each Liferay DXP version has a predefined BOM that you can specify for your workspace to reference. Each BOM defines the artifacts and their versions used in the specific release. BOMs list all dependencies in a management fashion, so it doesn't **add** dependencies to your project; it only **provides** your build tool (e.g., Gradle or Maven) the versions needed for the project's defined artifacts. This means you don't need to specify your dependency versions; the BOM automatically defines the appropriate artifact versions based on the BOM.

You can override a BOM's defined artifact version by specifying a different version in your project's `build.gradle`. Artifact versions defined in your project's build files override those specified in the predefined BOM. Note that overriding the BOM can be dangerous; make sure the new version is compatible in the targeted platform.

For more information on BOMs, see the Importing Dependencies section in Maven's official documentation.

Pretty cool, right? Next, you'll step through an example configuration.

Setting the Target Platform

Setting the version to develop for takes two steps:

1. Open the workspace's `gradle.properties` file and set the `liferay.workspace.target.platform.version` property to the version you want to target. For example,

   ```
   liferay.workspace.target.platform.version=7.0.6
   ```

   If you're using Liferay DXP, the versions are specified based on service packs. For example, you could set your target platform workspace Gradle property to

   ```
   liferay.workspace.target.platform.version=7.0.10.7
   ```

   Important: You can leverage the target platform features in Liferay Portal GA6+ and Liferay DXP 7.0 SP7+. Previous versions do not provide these features.

   The versions following the SP7 release of DXP follow fix pack versions (e.g., `7.0.10.fp69`, `7.0.10.fp70`, etc.).

2. Once the target platform is configured, check to make sure no dependencies in your Gradle build files specify a version. The versions are now imported from the configured target platform's BOM. For example, a simple MVC portlet's `build.gradle` may look something like this:

   ```
   dependencies {
       compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel"
       compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib"
       compileOnly group: "javax.portlet", name: "portlet-api"
       compileOnly group: "javax.servlet", name: "javax.servlet-api"
       compileOnly group: "jstl", name: "jstl"
       compileOnly group: "org.osgi", name: "osgi.cmpn"
   }
   ```

Note: The `liferay.workspace.target.platform.version` property also sets the distro JAR, which can be used in to validate your projects during the build process. See the Validating Modules Against the Target Platform tutorials for more info.

The target platform functionality is available in Liferay Workspace version 1.9.0+. If you have an older version, you must update it to leverage platform targeting. See the Updating Liferay Workspace tutorial to do this.

You now know how to configure a target platform in workspace and how dependencies without versions appear in your Gradle build files. You're all set!

Targeting a Platform Outside of Workspace

If you prefer to not use Liferay Workspace, but still want to target a platform, you must apply the Target Platform Gradle plugin to the root `build.gradle` file of your custom multi-module Gradle build.

To do this, your `build.gradle` file should look similar to this:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.target.platform", version: "1.1.6"
    }
    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.target.platform"

dependencies {
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom", version: "7.0.6"
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom.compile.only", version: "7.0.6"
}
```

Liferay DXP users must replace the artifact names and versions:

- `release.portal.bom` → `release.dxp.bom`
- `release.portal.bom.compile.only` → `release.dxp.bom.compile.only`
- `7.0.6` → `7.0.10.7`

This Gradle code

- applies Liferay's Target Platform Gradle plugin
- configures the repository that provides the necessary artifacts for your project build
- sets the Target Platform plugin's dependencies:
  - `release.portal.bom`: provides all the artifacts included in Liferay DXP.
  - `release.portal.bom.compile.only`: provides artifacts that are not included in Liferay DXP, but are necessary to reference during the build (e.g., `org.osgi.core`).

If you're interested in advanced search and/or debugging Liferay DXP's source using Liferay @ide@, you must also apply the following configuration:

```
targetPlatformIDE {
    includeGroups "com.liferay", "com.liferay.portal"
}
```

This indexes the target platform's source code and makes it available to @ide@.

Now you can define your target platform!

Chapter 55

Validating Modules Against the Target Platform

Important: Validating modules with the `resolve` task is deprecated. It only functions as it's documented here in versions prior to Liferay Workspace (Gradle only) version 2.0.3. It is being redesigned for workspace versions 2.0.3+ and is still in development at this time.

After you write a module in Liferay Workspace, you can validate it before deployment to make sure of several things:

- Will my app deploy successfully?
- Will there be some sort of missing requirement?
- If there's an issue, how do I diagnose it?

These are all common worries that can be frustrating.

Instead of deploying your app and checking for errors in the log, you can validate your app before deployment. This is done by calling Liferay Workspace's `resolve` task, which validates your modules against a targeted platform. Continue on to learn how this works.

55.1 Resolving Your Modules

Deploying your modules only to be met with console errors or mysterious problems can be frustrating. You can avoid this painful process by resolving your modules before deployment. This can be done by calling the `resolve` Gradle task provided by Liferay Workspace.

```
../gradlew resolve
```

This task gathers all the capabilities provided by

- the specified version of Liferay DXP (i.e., targeted platform)
- the current workspace's modules

Some capabilities/information gathered by the `resolve` task that are validated include

- declared required capabilities
- module versions
- package imports/use constraints
- service references

It also computes a list of run requirements for your project. Then it compares the current project's requirements against the gathered capabilities. If your project requires something not available in the gathered list of capabilities, the task fails.

The task can only validate OSGi modules. It does not work with WAR-style projects, themes, or npm portlets.

Note: The `resolve` task can be executed from a specific project folder or from the workspace's root folder. Running the task from the root folder validates all the modules in your workspace.

The `resolve` task can automatically gather the available capabilities from your workspace, but you must specify this for your targeted Liferay DXP version. To do this, open your workspace's `gradle.properties` file and set the `liferay.workspace.target.platform.version` property to the version you want to target. For example,

```
liferay.workspace.target.platform.version=7.0.6
```

If you're using Liferay DXP, the versions are specified based on service packs. For example, you could set your target platform workspace Gradle property to

```
liferay.workspace.target.platform.version=7.0.10.7
```

Important: You can leverage the target platform features in Liferay Portal GA6+ and Liferay DXP 7.0 SP7+. Previous versions do not provide these features.

The versions following the SP7 release of DXP follow service pack versions (e.g., `7.0.10.8` (SP8), `7.0.10.9` (SP9), etc.).

This provides a static *distro* JAR for the specified version of Liferay DXP, which contains all the metadata (i.e., capabilities, packages, versions, etc.) running in that version. The distro JAR is a complete snapshot of everything provided in the OSGi runtime; this serves as the target platform's list of capabilities that your modules are validated against.

You can now validate your module projects before deploying them! Sometimes, you must modify the `resolve` task's default behavior to successfully validate your app. See the Modifying the Target Platform's Capabilities tutorial for more information. For help resolving common output errors printed by the `resolve` task, see the Resolving Common Output Errors Reported by the resolve Task article.

55.2 Modifying the Target Platform's Capabilities

In a perfect world, everything the `resolve` task gathers and checks against would work during your development process. Unfortunately, there are exceptions that may force you to modify the default functionality of the `resolve` task. If you're unfamiliar with workspace's `resolve` task, see the Resolving Your Modules tutorial for more information.

There are two scenarios you may run into during development that require a modification for your project to pass the resolver check.

- You're depending on a third party library that is not available in the targeted Liferay DXP instance or the current workspace.

- You're depending on a customized distribution of Liferay DXP.

You'll explore these use cases next.

Depending on Third Party Libraries Not Included in Liferay DXP

The `resolve` task, by default, gathers all of Liferay DXP's capabilities and the capabilities of your workspace's modules. What if, however, your module depends on a third party project that is not included in either space (e.g., Google Guava)?. The `resolve` task fails by default if your project depends on this project type. You probably plan to have this project deployed and available at runtime, so it's not a concern, but the resolver doesn't know that; you must customize the resolver to bypass this.

There are three ways you can do this:

- Embed the third party library in your module
- Add the third party library's capabilities to the current static set of resolver capabilities
- Skip the resolving process for your module

For help resolving third party dependency errors, see the Resolving Third Party Library Package Dependencies tutorial.

Embed the Third Party Library in Your Module

If you only have one module that depends on the third party project, you can bypass the resolver failure by embedding the JAR in your module. This is not a best practice if more than one project in the OSGi container depends on that module. See the Embedding Libraries in a Module

section for more details.

Add the Third Party Library's Capabilities to the Current Static Set of Resolver Capabilities

You can add your third party dependencies to the target platform's default list of capabilities by listing them as provided modules. Do this by adding the following Gradle code into your workspace's root `build.gradle` file:

```
dependencies {
    providedModules group: "GROUP_ID", name: "NAME", version: "VERSION"
}
```

For example, if you wanted to add Google Guava as a provided module, it would look like this:

```
dependencies {
    providedModules group: "com.google.guava", name: "guava", version: "23.0"
}
```

This both provides the third party dependency to the resolver, and it downloads and includes it in your Liferay DXP bundle's `osgi/modules` folder when you initialize it (e.g., `blade server init`).

Skip the Resolving Process for Your Module

It may be easiest to skip validating a particular module during the resolve process. To do this, open your workspace's root `build.gradle` file and insert the following Gradle code at the bottom of the file:

```
targetPlatform {
    resolveOnlyIf { project ->
        project.name ≠ 'PROJECT_NAME'
    }
}
```

Be sure to replace the `PROJECT_NAME` filler with your module's name (e.g., `test-api`).

If you prefer to disable the Target Platform plugin altogether, you can add a slightly different directive to your `build.gradle` file:

```
targetPlatform {
    onlyIf { project ->
        project.name ≠ 'PROJECT_NAME'
    }
}
```

This both skips the `resolve` task execution and disables BOM dependency management.

Now the `resolve` task skips your module project.

Depending on a Customized Distribution of Liferay DXP

There are times when manually specifying your project's list of dependent JARs does not suffice. If your app requires a customized Liferay DXP instance to run, you must regenerate the target platform's default list of capabilities with an updated list. Two examples of a customized Liferay DXP instance are described below:

Example 1: Leveraging an External Feature

There are many external features/frameworks available that are not included in the downloadable bundle by default. After deploying a feature/framework, it's available for your module projects to leverage. When validating your app, however, the `resolve` task does not have access to external capabilities not included by default. For example, Audience Targeting is an example of this type of external framework. If you're creating a Liferay Audience Targeting rule that depends on the Audience Targeting framework, you can't easily provide a slew of JARs for your module. In this case, you should install the platform your code depends on and regenerate an updated list of capabilities that your Liferay DXP instance provides.

Example 2: Leveraging a Customized Core Feature

You can extend Liferay DXP's core features to provide a customized experience for your intended audience. Once deployed, you can assume these customizations are present and build other things on top of them. The new capabilities resulting from your customizations are not available, however, in the target platform's default list of capabilities. Therefore, when your application relies on non-default capabilities, it fails during the `resolve` task. To get around this, you must regenerate a new list of capabilities that your customized Liferay DXP instance provides.

To regenerate the target platform's capabilities (distro JAR) based on the current workspace's Liferay DXP instance, follow the steps below:

1. Start the Liferay DXP instance stored in your workspace. Make sure the platform you want to depend on is installed.

2. Download the BND Remote Agent JAR file and copy it into the `osgi/modules` folder.

3. From the root folder of your workspace, run the following command:

```
bnd remote distro -o custom_distro.jar release.portal.distro 7.0.6
```

Liferay DXP users must replace the `release.portal.distro` artifact name with `release.dxp.distro` and use the `7.0.10.7` version syntax.

This connects to the newly deployed BND agent running in Liferay DXP and generates a new distro JAR named `custom_distro.jar`. All other capabilities inherit their functionality based on your Liferay DXP instance, so verify the workspace bundle is the version you plan to release in production.

4. Navigate to your workspace's root `build.gradle` file and add the following dependency:

```
dependencies {
    targetPlatformDistro files('custom_distro.jar')
}
```

Now your workspace is pointing to a custom distro JAR file instead of the default one provided. Run the `resolve` task to validate your modules against the new set of capabilities.

55.3 Including the Resolver in Your Gradle Build

By default, Liferay Workspace provides the `resolve` task as an independent executable. It's provided by the Target Platform Gradle plugin and is not integrated in any other Gradle processes. This gives you control over your Gradle build without imposing strategies you may not want included in your default build process.

With that said, the `resolve` task can be useful to include in your build process if you want to check for errors in your module projects before deployment. Instead of resolving your projects separately from your standard build, you can build and resolve them all in one shot.

In Liferay Workspace, the recommended path for doing this is adding it to the default `check` Gradle task. The `check` task is provided by default in a workspace by the Java plugin. Adding the `resolve` task to the `check` lifecycle task also promotes the `resolve` task to run for CI and other test tools that typically run the `check` task for verification. Of course, Gradle's `build` task also depends on the `check` task, so you can run `gradlew build` and run the resolver too.

To call the `resolve` task during the `check` task automatically, open your workspace's root `build.gradle` file and add the following directive:

```
check.dependsOn resolve
```

You can also configure this for specific projects in a workspace if you don't want all modules to be included in the global `check`.

If the `resolve` task runs during every Gradle build, you may want to prevent the build from failing if there are errors reported by the resolver. To do this, open your workspace's root `build.gradle` file and add the following code:

```
targetPlatform {
    ignoreResolveFailures = true
}
```

This reports the failures without failing the build. Note, this can only be configured in the workspace's root `build.gradle` file.

Awesome! You can now run the `resolve` task in your current Gradle lifecycle.

55.4 Validating Modules Outside of Workspace

If you prefer to not use Liferay Workspace, but still want to validate modules against a target platform, you must apply the Target Platform Gradle plugin to the root `build.gradle` file of your multi-module Gradle build. Follow the Targeting a Platform Outside of Workspace section to do this.

Once you have the Target Platform plugin and its BOM dependencies configured, you must configure the `targetPlatformDistro` dependency. Open your project's root `build.gradle` file and add it to the list of dependencies. It should look like this:

```
dependencies {
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom", version: "7.0.6"
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom.compile.only", version: "7.0.6"
    targetPlatformDistro group: "com.liferay.portal", name "release.portal.distro", version: "7.0.6"
}

Liferay DXP users must replace the artifact names and versions:

- `release.portal.bom` &rarr; `release.dxp.bom`
- `release.portal.bom.compile.only` &rarr; `release.dxp.bom.compile.only`
- `release.portal.distro` &rarr; `release.dxp.distro`
- `7.0.6` &rarr; `7.0.10.7`
```

Now you can validate your modules against a target platform!

55.5 Updating Liferay Workspace

Liferay Workspace is continuously being updated with new features. If you created your Workspace a while ago, you may be missing out on some of the latest features that could improve your Liferay development experience. Updating your Liferay Workspace is easy; you'll learn how to do it next.

1. Find the latest Liferay Workspace version. To do this, open the Liferay Gradle Plugins Workspace Change Log and copy the version to which you want to upgrade. You can find the updates and new features associated with each version by browsing the change log too.

2. Open your Liferay Workspace's `settings.gradle` file. This file resides in your Workspace's root folder.

3. In the `dependencies` block, you'll find code similar to below:

   ```
   dependencies {
       classpath group: "com.liferay", name: "com.liferay.gradle.plugins.workspace", version: "[WORKSPACE_VERSION]"
   }
   ```

   Update the `com.liferay.gradle.plugins.workspace` dependency's `version` to the version number you copied from the change log in step 1.

4. Execute any Gradle command to initiate the update process for your Workspace (e.g., `blade gw tasks`).

Awesome! You learned where to check for Liferay Workspace's latest version, how to update your Workspace to that version, and how to initiate the update process.

Chapter 56

Maven

Maven is a viable option for managing Liferay projects if you don't want to use Liferay's default Gradle management system. Liferay provides several Maven plugins to let you generate and manage your project. Liferay also provides Maven artifacts that are easy to obtain and are required for Liferay Maven module development. In the Maven tutorials, you'll learn how to

- Install Liferay Maven artifacts.
- Generate Liferay projects using Maven archetypes.
- Create a Module JAR using Maven.
- Deploy a module built with Maven to Liferay DXP.
- Create a remote repository for Maven projects.
- Deploy a Maven project to a remote repository.
- Use Service Builder in a Maven project.
- Compile Sass files in a Maven project.
- Build a Liferay theme in a Maven project.
- Leverage the Maven Workspace.

Because Liferay DXP is tool agnostic, Maven is fully supported for Liferay DXP development. Read on to learn more!

56.1 Installing Liferay Maven Artifacts

To create Liferay modules using Maven, you'll need the archives required by Liferay (e.g., JAR and WAR files). This isn't a problem–Liferay provides them as Maven artifacts. You can retrieve them from a remote repository.

There are two repositories that contain Liferay artifacts: Central Repository and Liferay Repository. The Central Repository is the default repository used to download artifacts if you don't have a remote repository configured. The Central Repository *usually* offers the latest Liferay Maven artifacts, but using the the Liferay Repository *guarantees* the latest artifacts released by Liferay. Other than a slight delay between artifact releases between the two repositories, they're identical. You'll learn how to reference both of them next.

Using the Central Repository to install Liferay Maven artifacts only requires that you specify your module's dependencies in its `pom.xml` file. For example, the snippet below sets a dependency on Liferay's `com.liferay.portal.kernel` artifact:

```
<dependencies>
    <dependency>
        <groupId>com.liferay.portal</groupId>
        <artifactId>com.liferay.portal.kernel</artifactId>
        <version>2.0.0</version>
        <scope>provided</scope>
    </dependency>
    ...
</dependencies>
```

When packaging your module, the automatic Maven artifact installation process only downloads the artifacts necessary for that module from the Central Repository.

You can view the published Liferay Maven artifacts on the Central Repository by searching for *liferay maven* in the repo's Search bar. For convenience, you can reference the available artifacts at http://search.maven.org/#search|ga|1|liferay maven. Use the Latest Version column as a guide to see what's available for the intended version of Liferay DXP for which you're developing.

If you'd like to access Liferay's latest Maven artifacts, you can configure Maven to use Liferay's Nexus repository instead by inserting the following snippet in your project's parent `pom.xml`:

```
<repositories>
    <repository>
        <id>liferay-public-releases</id>
        <name>Liferay Public Releases</name>
        <url>https://repository.liferay.com/nexus/content/repositories/liferay-public-releases</url>
    </repository>
</repositories>

<pluginRepositories>
    <pluginRepository>
        <id>liferay-public-releases</id>
        <url>https://repository.liferay.com/nexus/content/repositories/liferay-public-releases/</url>
    </pluginRepository>
</pluginRepositories>
```

The above configuration retrieves artifacts from Liferay's release repository.

Note: Liferay also provides a snapshot repository that you can access by modifying the `<id>`, `<name>`, and `<url>` tags to point to that repo. This repository should only be used in special cases. You'll also need to enable accessing the snapshot artifacts:

```
 <snapshots>
     <enabled>true</enabled>
 </snapshots>
```

If you've configured the Liferay Nexus repository to access Liferay Maven artifacts and you've already been syncing from the Central Repository, you may need to clear out parts of your local repository to force Maven to re-download the newer artifacts. Also, do not leave the Liferay repository configured when publishing artifacts to Maven Central. You must comment out the Liferay Repository credentials when publishing your artifacts.

The Liferay Maven repository offers a good alternative for those who want the most up-to-date Maven artifacts produced by Liferay.

Congratulations! You've downloaded the Liferay artifacts and installed them to your chosen repository.

56.2 Generating New Projects Using Archetypes

Creating Maven projects from scratch can be a lot of work. What dependencies does my Liferay portlet project need? What does a Liferay Maven Service Builder project look like? How do I create a Liferay Maven-based context contributor? These questions can be answered with three words: Liferay Maven Archetypes.

Liferay provides a slew of Maven archetypes for easy Liferay module projects. In this tutorial, you'll learn how to use Liferay's Maven archetypes to generate your module project.

At the time of this writing, Liferay provides just under 40 Maven archetypes for you to use; expect this number to continue growing! These archetypes are generated from the Central Repository, unless you've configured for them to be generated from another remote repository (e.g., Liferay Repository. You can view the Liferay-provided Maven archetypes by running the following command:

```
mvn archetype:generate -Dfilter=liferay
```

The generated archetypes are not all intended for the latest Liferay DXP release. Some are intended for earlier versions of Liferay Portal (e.g., 6.2, 6.1, etc.). An easy way to tell if the archetype is compatible with 7.0 is by inspecting the archetype's package name. Archetypes with the `com.liferay.maven.archetypes` prefix are legacy archetypes. Those prefixed with `com.liferay.project.templates.[TYPE]` or `com.liferay.faces.archetype:[TYPE]` are compatible with 7.0.

Here's a brief list of some popular Maven archetypes provided by Liferay:

- Activator
- Context Contributor
- Liferay Faces portlets
  - Alloy
  - ICEfaces
  - JSF
  - PrimeFaces
  - RichFaces
- MVC Portlet
- Panel App
- Portlet Provider
- Service Builder
- Service Wrapper
- Vaadin Liferay portlet

For documentation on the archetypes (project templates) compatible with 7.0, see the Project Templates reference section. Visit Maven's Archetype Generation documentation for further details on how to modify the Maven archetype generation process.

Note: If you're creating a JSF portlet using Liferay Faces, you can find example archetype declarations for JSF component suites at http://www.liferayfaces.org.

Here's an example that creates a Liferay MVC portlet using its Liferay Maven archetype.

1. On the command line, navigate to where you want your Maven project. Run the Maven archetype generation command filtered for Liferay archetypes only:

```
mvn archetype:generate -Dfilter=liferay
```

2. Select the `com.liferay.project.templates.mvc.portlet` archetype by choosing its corresponding number (e.g., `8`).

   In most cases, you should choose the latest archetype version. The archetype versions provided are compatible with all 7.x versions of Liferay DXP.

3. Depending on the Maven archetype you selected, you're given a set of archetype options to fill out for your Maven project. For the MVC portlet archetype, you could use these properties:

   - `groupId`: `com.liferay`
   - `artifactId`: `com.liferay.project.templates.mvc.portlet`
   - `version`: `1.0.0`
   - `package`: `com.liferay.docs`
   - `className`: `SampleMVC`

   Once you've filled out the required property values, you're given a summary of the properties configuration you defined. Enter `Y` to confirm your project's configuration.

Your Maven project is generated and available from the folder for which you ran the archetype generation command. If you have an existing parent `pom.xml` file in that folder, your module project is automatically accounted for there:

```
<modules>
    ...
    <module>com.liferay.project.templates.mvc.portlet</module>
</modules>
```

The Liferay Maven archetypes generate deployable Liferay module projects, but they're bare bones and likely require further customizations.

If you want to generate a quick foundation for a Liferay module built with Maven, using Liferay Maven archetypes is your best option.

56.3 Creating a Module JAR Using Maven

If you have an existing Liferay module built with Maven that you created from scratch, or you're upgrading your Maven project from a previous version of Liferay, your project probably can't generate an executable OSGi JAR. Don't fret! You can do this by making a few minor configurations in your module's POMs.

Note: If you used Liferay's Maven archetypes to generate your module project, the project already has the Maven plugins required to generate an OSGi JAR.

Continue on to see how this is done.

1. In your project's `pom.xml` file, add the BND Maven Plugin declaration:

```
<plugin>
    <groupId>biz.aQute.bnd</groupId>
    <artifactId>bnd-maven-plugin</artifactId>
    <version>3.3.0</version>
    <executions>
        <execution>
            <goals>
                <goal>bnd-process</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

The BND Maven plugin prepares all your Maven module's resources (e.g., `MANIFEST.MF`) and inserts them into the generated `[Maven Project]/target/classes` folder. This plugin prepares your module to be packaged as an OSGi JAR deployable to Liferay DXP.

```
**Note:** Although WABs can be generated using the `bnd-maven-plugin`, this
is not supported by Liferay. WABs should be created as a standard WAR
project and deployed to the
[Liferay WAB Generator](/docs/7-0/tutorials/-/knowledge_base/t/using-the-wab-generator),
which generates a WAB for you.
```

---

2. In your project's `pom.xml` file, add the Maven JAR Plugin declaration:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.6</version>
            <configuration>
                <archive>
                    <manifestFile>${project.build.outputDirectory}/META-INF/MANIFEST.MF</manifestFile>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

The Maven JAR plugin builds your Maven project as a JAR file, including the resources generated by the BND Maven plugin. The above configuration also sets the default project `MANIFEST.MF` file path for your project, which is essential when packaging your module using the BND Maven plugin. By default, the Maven JAR Plugin ignores the `target/classes/META-INF/MANIFEST.MF` generated by the BND Maven plugin, so you must explicitly set it as the manifest file so it's included properly in the generated JAR file.

3. Make sure you've added a `bnd.bnd` file to your Liferay Maven project, residing in the same folder as your project's `pom.xml` file.

4. Build your Maven OSGi JAR by running

```
mvn package
```

Your Maven JAR is generated in your project's /target folder. You can deploy it manually into Liferay DXP's /deploy folder, or you can configure your project to deploy automatically to Liferay DXP by following the Deploying a Module Built with Maven to Liferay DXP tutorial.

Fantastic! You've configured your Liferay Maven project to package itself into a deployable OSGi module.

56.4 Deploying a Module Built with Maven to Liferay DXP

There are two ways to deploy a Maven-built Liferay module:

1. Copy your generated Maven module JAR to your Liferay DXP instance's /deploy folder.
2. Configure your Maven project to deploy to the Liferay DXP instance automatically by running a Maven command via the command prompt.

Although manually copying your module JAR for deployment is a viable option, this is an inefficient way to deploy your projects. With a small configuration in your Maven POMs, you can deploy a module to Liferay DXP with one command execution.

In previous versions of Liferay Portal, you were able to execute the liferay:deploy command to deploy your configured Maven project to a Liferay server. This is no longer possible since the liferay-maven-plugin is not applied to Maven projects built from Liferay archetypes.

A prerequisite for this tutorial is to have your project configured to generate an OSGi module JAR; if you haven't done this, visit the Creating a Module JAR Using Maven tutorial for more information.

1. Add the following plugin configuration to your Liferay Maven project's parent pom.xml file.

```
<build>
    <plugins>
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.portal.tools.bundle.support</artifactId>
            <version>2.0.1</version>
            <executions>
                <execution>
                    <id>default-deploy</id>
                    <goals>
                        <goal>deploy</goal>
                    </goals>
                    <phase>pre-integration-test</phase>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

   This POM configuration applies Liferay's Bundle Support plugin by defining its groupId, artifactId, and version. You can learn more about this plugin in the Maven Workspace tutorial. The logic also defines the executions tag, which configures the Bundle Support plugin to run during the pre-integration-test phase of your Maven project's build lifecycle. The deploy goal is defined for that lifecycle phase.

2. By default, the Bundle Support plugin deploys to the Liferay installation in the bundles folder, located in your plugin's parent folder. If you do not have your project set up this way, you must define your Liferay home folder in your POM. You can do this by adding the following logic within the plugin tags, but outside of the execution tags:

```
<configuration>
    <liferayHome>LIFERAY_HOME_PATH</liferayHome>
</configuration>
```

An example configuration would look like this:

```
<configuration>
    <liferayHome>C:/liferay/liferay-ce-portal-7.0-ga7</liferayHome>
</configuration>
```

```
**Note:** Maven applications built for previous Liferay Portal versions
required the `<liferay.maven.plugin.version>` tag to do various tasks (e.g.,
deploying to a Liferay server). This tag is not needed since the old
`liferay-maven-plugin` is no longer used.
```

---

3. Run this command to deploy your project:

   ```
   mvn verify
   ```

That's it! Your Liferay Maven project is built and deployed automatically to your Liferay DXP instance.

56.5 Creating a Maven Repository

You'll frequently want to share Liferay artifacts and modules with teammates or manage your repositories using a GUI. Sonatype Nexus is a valuable tool for managing your repositories. It's a Maven repository management server that facilitates creating and managing release servers, snapshot servers, and proxy servers. There are several other Maven repository management servers you can use (for example, Artifactory), but this tutorial focuses on Nexus.

To create a Maven repository using Nexus, download Nexus and follow the instructions at http://books.sonatype.com/nexus-book/reference/install.html to install and start it.

To create your own repository using Nexus, follow these steps:

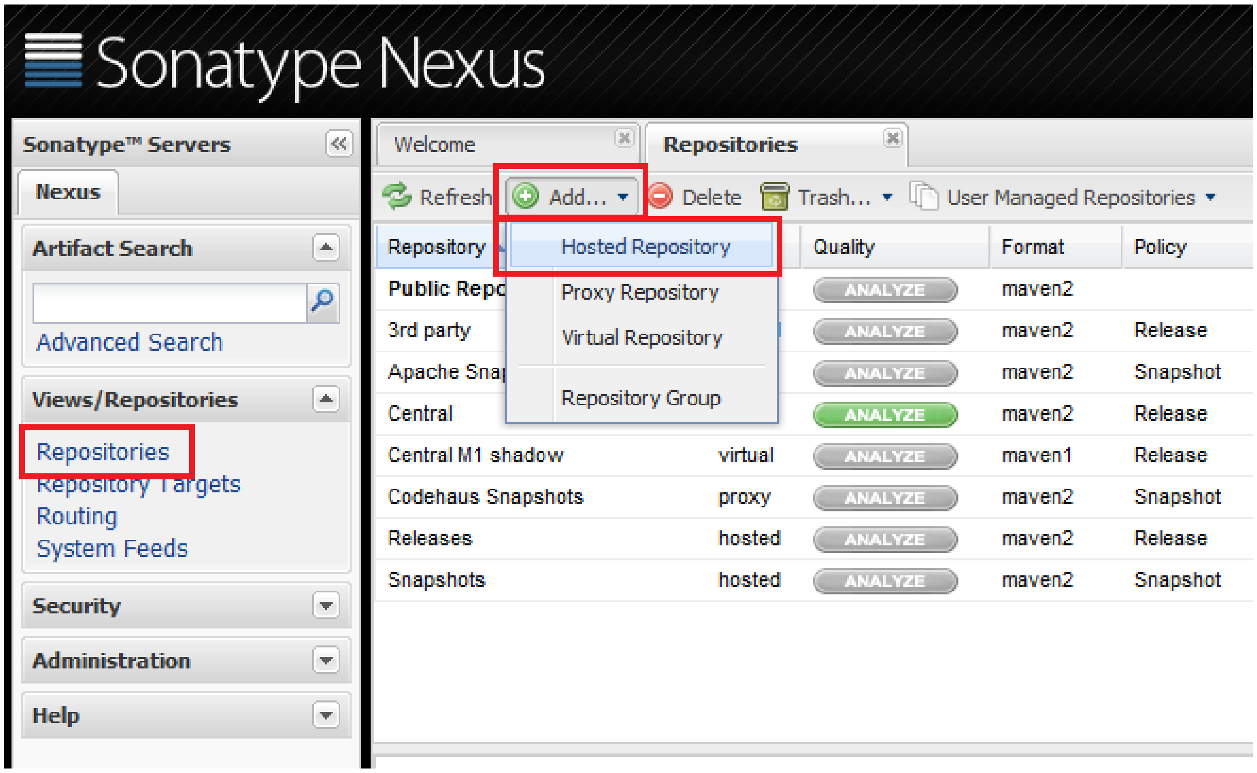
1. Open your web browser; navigate to your Nexus repository server (e.g., http://localhost:8081/nexus) and log in. The default user name is `admin` with password `admin123`.

2. Click on *Repositories* and navigate to *Add...* → *Hosted Repository*.

   To learn more about each type of Nexus repository, read Sonatype's Managing Repositories guide.

3. Enter repository properties appropriate for the type of artifacts it will hold. If you're installing release version artifacts into the repository, specify *Release* as the repository policy. Below are example repository property values:

   - **Repository ID:** *liferay-releases*
   - **Repository Name:** *Liferay Release Repository*
   - **Provider:** *Maven2*
   - **Repository Policy:** *Release*



Repository		Quality	Format	Policy
Public Repo		ANALYZE	maven2	
3rd party		ANALYZE	maven2	Release
Apache Snap		ANALYZE	maven2	Snapshot
Central		ANALYZE	maven2	Release
Central M1 shadow	virtual	ANALYZE	maven1	Release
Codehaus Snapshots	proxy	ANALYZE	maven2	Snapshot
Releases	hosted	ANALYZE	maven2	Release
Snapshots	hosted	ANALYZE	maven2	Snapshot

Figure 56.1: Adding a repository to hold your Liferay artifacts is easy with Nexus.

4. Click *Save*.

You just created a Liferay Maven repository accessible from your Nexus repository server! Congratulations!

It's also useful to create a Maven repository to hold snapshots of each Liferay module you create. Creating a snapshot repository is almost identical to creating a release repository. The only difference is that you specify *Snapshot* as its repository policy. For example, examine an example snapshot repository's property values:

- **Repository ID:** *liferay-snapshots*
- **Repository Name:** *Liferay Snapshot Repository*
- **Provider:** *Maven2*
- **Repository Policy:** *Snapshot*

Voila! You've created a repository for your Liferay releases (i.e., `liferay-releases`) and Liferay snapshots (i.e., `liferay-snapshots`). To learn how to deploy your Liferay Maven artifacts to a Nexus repository, see the Deploying Liferay Maven Artifacts to a Repository tutorial.

Next, you'll configure your new repository servers in your Maven settings to install artifacts to them.

Configuring Local Maven Settings

Before using your repository servers, you must specify them in your Maven environment settings. Your repository settings let Maven find the repository and retrieve and install artifacts. You can configure your local Maven settings in the `[USER_HOME]/.m2/settings.xml` file.

You only need to configure a repository server if you're sharing artifacts (e.g., Liferay artifacts and/or your modules) with others. If you're automatically installing Liferay artifacts from the Central/Liferay Repository and aren't interested in sharing artifacts, you don't need a repository server specified in your Maven settings. You can find out more about installing artifacts from the Central Repository or Liferay's own Nexus repository in the Installing Liferay Maven Artifacts tutorial.

To configure your Maven environment to access your `liferay-releases` and `liferay-snapshots` repository servers, do the following:

1. Navigate to your `[USER_HOME]/.m2/settings.xml` file. Create it if it doesn't yet exist.

2. Provide settings for your repository servers. Here are contents from a `settings.xml` file that has `liferay-releases` and `liferay-snapshots` repository servers configured:

```
<?xml version="1.0"?>
<settings>
    <servers>
        <server>
            <id>liferay-releases</id>
            <username>admin</username>
            <password>admin123</password>
        </server>
        <server>
            <id>liferay-snapshots</id>
            <username>admin</username>
            <password>admin123</password>
        </server>
    </servers>
</settings>
```

The user name `admin` and password `admin123` are the credentials of the default Nexus administrator account. If you changed these credentials for your Nexus server, make sure to update `settings.xml` with these changes.

Now that your repositories are configured, they're ready to receive all the Liferay Maven artifacts you'll download and the Liferay module artifacts you'll create!

56.6 Deploying Liferay Maven Artifacts to a Repository

Deploying artifacts to a remote repository is important if you intend to share your Maven projects with others. First, you must have a remote repository that can hold deployed Maven artifacts. If you do not currently have a remote repository, see the Creating a Maven Repository tutorial to learn how you can set up a Nexus repository. Also make sure your `[USER_HOME]/.m2/settings.xml` file specifies your remote repository's ID, user name, and password.

To deploy to a remote repository, your Liferay module should be packaged using Maven. Maven provides a packaging command that creates an artifact (JAR) that can be easily deployed to your remote repository. You'll learn how to do this with a Liferay portlet module.

Once you've created a deployable artifact, you'll configure your module project to communicate with your remote repository and use Maven's deploy command to send it on its way. Once your module project resides on the remote repository, other developers can configure your remote repository in their projects and set dependencies in their project POMs to reference it.

To follow this tutorial, you'll need a Liferay module built with Maven. For demonstration purposes, this tutorial uses the `portlet.ds` sample module project. To follow along with this module, download the portlet.ds ZIP.

1. Create a folder anywhere on your machine to serve as the parent folder for your Liferay modules. Unzip the `portlet.ds` module project into that folder.

2. Create a `pom.xml` file inside this folder. Copy the following logic into the parent POM:

```
<?xml version="1.0" encoding="UTF-8"?>
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
>

    <modelVersion>4.0.0</modelVersion>
    <groupId>liferay.sample</groupId>
    <artifactId>liferay.sample.maven</artifactId>
    <version>1.0.0</version>
    <name>Liferay Maven Module Projects</name>
    <packaging>pom</packaging>

    <distributionManagement>
        <repository>
            <id>liferay-releases</id>
            <url>http://localhost:8081/nexus/content/repositories/liferay-releases</url>
        </repository>
    </distributionManagement>

    <modules>
        <module>portlet.ds</module>
    </modules>
</project>
```

   The tags `<modelVersion>` through `<packaging>` are POM tags that are used frequently in parent POMs. Visit Maven's POM Reference documentation for more information.

   The `<distributionManagement>` tag specifies the deployment repository for all module projects residing in the parent folder. You should include the repository's ID and URL. The above `distributionManagement` declaration is configured for the Liferay Nexus repository created in the Creating a Maven Repository tutorial. That tutorial also created the `[USER_HOME]/.m2/settings.xml`, which specified the remote repository's ID, user name, and password. Both the parent POM and `settings.xml` file's repository declarations are required to deploy your modules to that remote repository.

   Finally, you must list the modules residing in the parent folder that you want deployed using the `<modules>` tag. The `portlet.ds` module is specified within that tag.

3. Open the `portlet.ds` module's `pom.xml` file. If you did not download the `portlet.ds` module project Zip, you can reference its POM below.

```
<project
    xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"
>

    <modelVersion>4.0.0</modelVersion>
    <artifactId>portlet.ds</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <parent>
        <groupId>liferay.sample</groupId>
        <artifactId>liferay.sample.maven</artifactId>
        <version>1.0.0</version>
        <relativePath>../pom.xml</relativePath>
    </parent>

    <dependencies>
        <dependency>
            <groupId>javax.portlet</groupId>
            <artifactId>portlet-api</artifactId>
            <version>2.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.osgi</groupId>
            <artifactId>org.osgi.service.component.annotations</artifactId>
            <version>1.3.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

The `portlet.ds` module's POM specifies its own attributes first, followed by the parent POM's attributes. Declaring the `<parent>` tag like above links the `portlet.ds` module to its parent POM, which is necessary to deploy to the remote repository. Then the module's dependencies are listed. These dependencies are downloaded from the Central Repository and installed to your local `.m2` repository when you package the `portlet.ds` module.

4. Now that you've configured your parent POM and module POM, package your Maven project. Navigate to your module project (e.g., `project.ds`) in your command prompt and run the Maven package command:

   ```
   mvn package
   ```

   This downloads and installs all your module's dependencies and packages the project into a JAR file. Navigate to your module project's generated build folder (e.g., `/target`). You'll notice there is a newly generated JAR file. This is the artifact you'll deploy to your Nexus repository.

5. Run Maven's deploy command to deploy your module project's artifact to your configured remote repository.

   ```
   mvn deploy
   ```

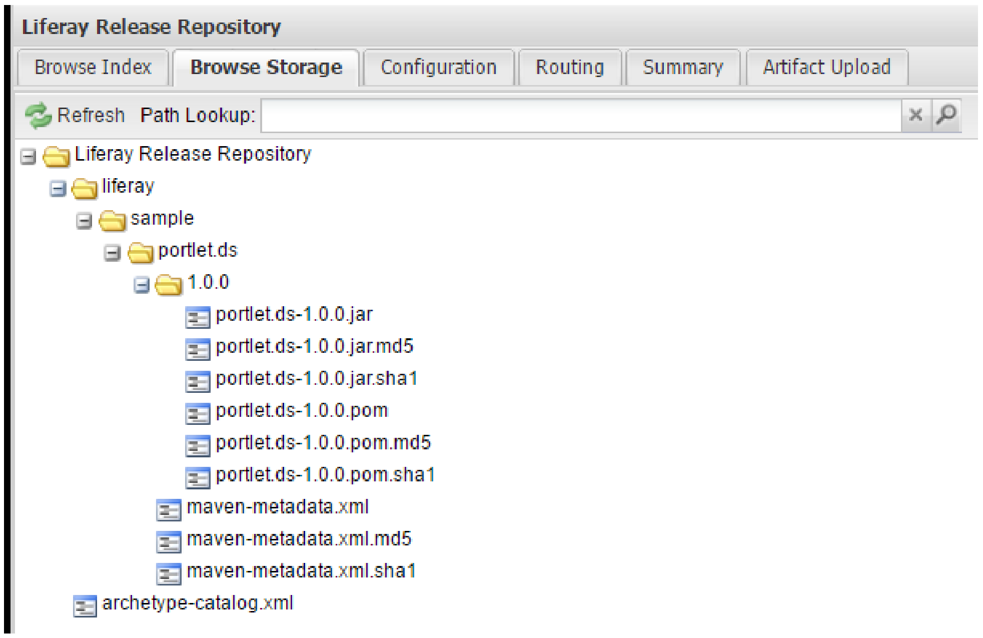
   Your console shows output from the artifact being deployed into your repository server.

To verify that your artifact is deployed, navigate to the Repositories page of your Nexus server and select your repository. A window appears below showing the Liferay artifact now deployed to your repository.

Awesome! You can now share your Liferay module projects with anyone by deploying them as artifacts to a remote repository!

56.7 Using Service Builder in a Maven Project

Liferay's Service Builder is a model-driven service generation tool that is frequently used by many Liferay module projects. If you have a Liferay Maven project, you may be wondering if Service Builder works with your Maven modules; the answer is a resounding yes!


Liferay Release Repository

Browse Index | Browse Storage | Configuration | Routing | Summary | Artifact Upload

Refresh Path Lookup:

```
Liferay Release Repository
    liferay
        sample
            portlet.ds
                1.0.0
                    portlet.ds-1.0.0.jar
                    portlet.ds-1.0.0.jar.md5
                    portlet.ds-1.0.0.jar.sha1
                    portlet.ds-1.0.0.pom
                    portlet.ds-1.0.0.pom.md5
                    portlet.ds-1.0.0.pom.sha1
                maven-metadata.xml
                maven-metadata.xml.md5
                maven-metadata.xml.sha1
    archetype-catalog.xml
```

Figure 56.2: Your repository server now provides access to your Liferay Maven artifacts.

The easiest way to add Service Builder to your Maven project is to create a new Maven project using Liferay's provided Service Builder archetype. You can learn how to generate a Service Builder Maven project by visiting the Service Builder Template tutorial. In some cases, this may not be possible due to a number of reasons:

- You're updating a legacy Maven project to follow OSGi modular architecture.
- You have a pre-existing modular Maven project and don't want to start over.

Time to get started!

1. Apply the Service Builder plugin in your Maven project's `pom.xml` file:

```
<build>
    <plugins>
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.portal.tools.service.builder</artifactId>
            <version>1.0.174</version>
            <configuration>
                <apiDirName>../blade.servicebuilder.api/src/main/java</apiDirName>
                <autoImportDefaultReferences>true</autoImportDefaultReferences>
                <autoNamespaceTables>true</autoNamespaceTables>
                <buildNumberIncrement>true</buildNumberIncrement>
                <hbmFileName>src/main/resources/META-INF/module-hbm.xml</hbmFileName>
                <implDirName>src/main/java</implDirName>
                <inputFileName>service.xml</inputFileName>
                <mergeModelHintsConfigs>src/main/resources/META-INF/portlet-model-hints.xml</mergeModelHintsConfigs>
                <modelHintsFileName>src/main/resources/META-INF/portlet-model-hints.xml</modelHintsFileName>
                <osgiModule>true</osgiModule>
                <propsUtil>com.liferay.blade.samples.servicebuilder.service.util.PropsUtil</propsUtil>
                <resourcesDirName>src/main/resources</resourcesDirName>
                <springFileName>src/main/resources/META-INF/spring/module-spring.xml</springFileName>
                <springNamespaces>beans,osgi</springNamespaces>
                <sqlDirName>src/main/resources/META-INF/sql</sqlDirName>
                <sqlFileName>tables.sql</sqlFileName>
                <testDirName>src/main/test</testDirName>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Service Builder is applied by specifying its `groupId`, `artifactId`, and `version`. The `configuration` tag used above is an example of what a Service Builder project's configuration could look like. All the configuration attributes above should be modified to fit your project.

The above code configures Service Builder for a `blade.servicebuilder.svc` module. When running Service Builder with this configuration, the project's API classes are generated in the `blade.servicebuilder.api` module's `src/main/java` folder. You can also specify your hibernate module mappings, implementation directory name, model hints file, Spring configurations, SQL configurations, etc. You can reference all the configurable Service Builder properties in the Service Builder with Maven reference article. Also, visit the Defining an Object-Relational Map with Service Builder tutorial for more information on defining a `service.xml` file to configure Service Builder.

2. Apply the WSDD Builder plugin declaration directly after the Service Builder plugin declaration:

```
<plugin>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.portal.tools.wsdd.builder</artifactId>
    <version>1.0.8</version>
    <configuration>
        <inputFileName>service.xml</inputFileName>
        <outputDirName>src/main/java</outputDirName>
        <serverConfigFileName>src/main/resources/server-config.wsdd</serverConfigFileName>
    </configuration>
</plugin>
```

The WSDD Builder is necessary to generate your project's remote services. Visit the Creating Remote Services tutorial for more information on WSDD (Web Service Deployment Descriptor). Similar to the Service Builder configuration, the `service.xml` file is set to define your project's remote services. Also, the `outputDirName` defines where the `*_deploy.wsdd` and `*_undeploy.wsdd` files are generated. Make sure to define your `server-config.wsdd` file, as well.

Terrific! You've successfully configured your Maven project to use Service Builder by applying the `com.liferay.portal.tools.service.builder` and `com.liferay.portal.tools.wsdd.builder` plugins in your project's POM. To run Service Builder, see the Running Service Builder and Understanding the Generated Code tutorial for instructions.

56.8 Compiling Sass Files in a Maven Project

If your Liferay Maven project uses Sass files to style its UI, you must configure the project to convert its Sass files into CSS files so they are recognizable for Maven's build lifecycle. It would be a real pain to convert your Sass files into CSS files manually before building your Maven project!

Liferay provides the `com.liferay.css.builder` plugin. The CSS Builder converts Sass files into CSS files so the Maven build can parse your style sheets.

Here's how to apply Liferay's CSS builder to your Maven project.

1. Open your project's `pom.xml` file and apply Liferay's CSS Builder:

```
<plugin>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.css.builder</artifactId>
    <version>2.0.1</version>
    <executions>
        <execution>
            <id>default-build</id>
            <phase>compile</phase>
            <goals>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <docrootDirName>${com.liferay.portal.tools.theme.builder.outputDir}</docrootDirName>
        <outputDirName>/</outputDirName>
        <portalCommonPath>target/deps/com.liferay.frontend.css.common.jar</portalCommonPath>
    </configuration>
</plugin>
```

The above configuration applies the CSS Builder by specifying its `groupId`, `artifactId`, and `version`. It then defines the CSS Builder's execution and configuration.

- The `executions` tag configures the CSS Builder to run during the `compile` phase of your Maven project's build lifecycle. The `build` goal is defined for that lifecycle phase.
- The `configuration` tag defines two important properties:
    - `docrootDirName`: The base resources folder containing the Sass files to compile.
    - `outputDirName`: The name of the sub-directories where the SCSS files are compiled to.
    - `portalCommonPath`: The file path for the Liferay Frontend Common CSS JAR file.

2. If you're using Bourbon in your Sass files, you'll need to add an additional plugin dependency to your project's POM. If you're not using Bourbon, skip this step. Add the following plugin dependency:

```
<plugin>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>copy</goal>
            </goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>com.liferay</groupId>
                        <artifactId>com.liferay.frontend.css.common</artifactId>
                        <version>2.0.4</version>
                    </artifactItem>
                </artifactItems>
                <outputDirectory>${project.build.directory}/deps</outputDirectory>
                <stripVersion>true</stripVersion>
            </configuration>
        </execution>
    </executions>
</plugin>
```

The maven-dependency-plugin copies the `com.liferay.frontend.css.common` dependency from Maven Central to your project's build folder so the CSS Builder can leverage it.

3. Use this command to compile your Maven project's Sass files:

   ```
   mvn compile
   ```

Note: Liferay's CSS Builder is supported for Oracle's JDK and uses a native compiler for increased speed. If you're using an IBM JDK, you may experience issues when building your SASS files (e.g., when building a theme). It's recommended to switch to using the Oracle JDK, but if you prefer using the IBM JDK, you must use the fallback Ruby compiler. To do this, add the following tag to your CSS Builder configuration in your POM:

```
<sassCompilerClassName>ruby</sasscompilerClassName>
```

Be aware that the Ruby-based compiler doesn't perform as well as the native compiler, so expect longer compile times.

Awesome! You can now compile Sass files in your Liferay Maven project.

56.9 Building Themes in a Maven Project

Liferay's Theme Builder is a tool used to build Liferay DXP theme files in your project. You can incorporate the Theme Builder into your Maven project to generate WAR-style themes deployable to Liferay DXP. To learn more about theming in Liferay DXP, see the Themes and Layout Templates tutorial section.

The easiest way to create a Liferay theme with Maven is to create a new Maven project using Liferay's provided Theme archetype. You can learn how to generate a Maven Theme project by visiting the Generating New Projects Using Archetypes tutorial. In some cases, however, this may not be convenient. For instance, if you have a legacy theme project and don't want to start over, generating a new project is not ideal.

For cases like this, you should manually configure your Maven project to build a theme. You'll learn how to do this next.

1. Configure Liferay's Theme Builder plugin in your project's `pom.xml` file:

   ```
   <build>
       <plugins>
           <plugin>
               <groupId>com.liferay</groupId>
               <artifactId>com.liferay.portal.tools.theme.builder</artifactId>
               <version>${com.liferay.portal.tools.theme.builder.version}</version>
               <executions>
                   <execution>
                       <phase>generate-resources</phase>
                       <goals>
                           <goal>build</goal>
                       </goals>
                       <configuration>
                           <diffsDir>${maven.war.src}</diffsDir>
                           <name>${project.artifactId}</name>
                           <outputDir>${project.build.directory}/${project.build.finalName}</outputDir>
                           <parentDir>${project.build.directory}/deps/com.liferay.frontend.theme.styled.jar</parentDir>
                           <parentName>_styled</parentName>
                           <templateExtension>ftl</templateExtension>
                           <unstyledDir>${project.build.directory}/deps/com.liferay.frontend.theme.unstyled.jar</unstyledDir>
                       </configuration>
                   </execution>
               </executions>
   ```

```
        </plugin>
    </plugins>
</build>
```

The above configuration applies the Theme Builder by specifying its `groupId`, `artifactId`, and `version`. It then defines the Theme Builder's execution and configuration.

- The `executions` tag configures the Theme Builder to run during the `generate-resources` phase of your Maven project's build lifecycle. The `build` goal is defined for that lifecycle phase.
- The `configuration` tag defines several important properties:
    - `diffsDir`: The folder holding the files to copy over the parent theme.
    - `name`: The new theme's name.
    - `outputDir`: The folder to build the theme.
    - `parentDir`: The parent theme's folder.
    - `parentName`: The parent theme's name.
    - `templateExtension`: The extension of the template files (e.g., `ftl` or `vm`).
    - `unstyledDir`: The unstyled theme's folder.

2. Apply the CSS Builder plugin, which is required to use the Theme Builder:

```
<plugin>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.css.builder</artifactId>
    <version>${com.liferay.css.builder.version}</version>
    <executions>
        <execution>
            <id>default-build</id>
            <phase>compile</phase>
            <goals>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <docrootDirName>target/${project.build.finalName}</docrootDirName>
        <outputDirName>/</outputDirName>
        <portalCommonPath>target/deps/com.liferay.frontend.css.common.jar</portalCommonPath>
    </configuration>
</plugin>
```

You can learn more about the CSS Builder's Maven configuration by visiting the Compiling Sass Files in a Maven Project tutorial.

3. You can configure your project to exclude Sass files from being packaged in your theme. This is optional, but is a nice convenience to keep any unnecessary source code out of your WAR. Since the Theme Builder creates a WAR-style theme, you should apply the maven-war-plugin so it instructs the WAR file packaging process to exclude Sass files:

```
<plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.0.0</version>
    <configuration>
        <packagingExcludes>**/*.scss</packagingExcludes>
    </configuration>
</plugin>
```

4. Insert the <packaging> tag in your project's POM so your project is correctly packaged as a WAR file. This tag can be placed with your project's `groupId`, `artifactId`, and `version` specifications like this:

```
<groupId>com.liferay</groupId>
<artifactId>com.liferay.project.templates.theme</artifactId>
<version>1.0.0</version>
<packaging>war</packaging>
```

5. Building themes requires certain dependencies. You can configure these dependenices in your project's `pom.xml` as directories or JAR files. If you choose to use JARs, you must apply the maven-dependency-plugin and have it copy JAR dependencies into your project from Maven Central:

```
<plugin>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>copy</goal>
            </goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>com.liferay</groupId>
                        <artifactId>com.liferay.frontend.css.common</artifactId>
                        <version>${com.liferay.frontend.css.common.version}</version>
                    </artifactItem>
                    <artifactItem>
                        <groupId>com.liferay</groupId>
                        <artifactId>com.liferay.frontend.theme.styled</artifactId>
                        <version>${com.liferay.frontend.theme.styled.version}</version>
                    </artifactItem>
                    <artifactItem>
                        <groupId>com.liferay</groupId>
                        <artifactId>com.liferay.frontend.theme.unstyled</artifactId>
                        <version>${com.liferay.frontend.theme.unstyled.version}</version>
                    </artifactItem>
                </artifactItems>
                <outputDirectory>${project.build.directory}/deps</outputDirectory>
                <stripVersion>true</stripVersion>
            </configuration>
        </execution>
    </executions>
</plugin>
```

This configuration copies the `com.liferay.frontend.css.common`, `com.liferay.frontend.theme.styled`, and `com.liferay.frontend.theme.unstyled` dependencies into your Maven project. Notice that you've set the `stripVersion` tag to `true` and you're setting the artifact versions within each `artifactItem` tag. You'll set these versions and a few other properties for your Maven project next.

6. Configure the properties for your project in its `pom.xml` file:

```
<properties>
    <com.liferay.css.builder.version>2.0.1</com.liferay.css.builder.version>
    <com.liferay.frontend.css.common.version>2.0.4</com.liferay.frontend.css.common.version>
    <com.liferay.frontend.theme.styled.version>2.0.28</com.liferay.frontend.theme.styled.version>
    <com.liferay.frontend.theme.unstyled.version>2.2.5</com.liferay.frontend.theme.unstyled.version>
    <com.liferay.portal.tools.theme.builder.version>1.1.4</com.liferay.portal.tools.theme.builder.version>
</properties>
```

The properties above set the versions for the CSS and Theme Builder plugins and their dependencies.

You've successfully configured your Maven project to build a Liferay theme! For info on running the Theme Builder in your Maven project, see the Theme Builder tutorial.

56.10 Maven Workspace

A Liferay Maven Workspace is a generated environment that is built to hold and manage Liferay projects built with Maven. This workspace aids in Liferay project management by applying various Maven plugins and configured properties. The Liferay Maven Workspace offers a full development lifecycle for your Maven projects to make developing them for Liferay DXP easier than ever. In this tutorial, you'll learn how to leverage the development lifecycle of a Liferay Maven Workspace.

First, you'll learn how to install a Maven Workspace.

Installation

The Maven Workspace is installed by generating the workspace project from an archetype. You can do this by executing the following command with your command line tool:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.workspace \
    -DgroupId=[GROUP_ID] \
    -DartifactId=[WORKSPACE_NAME] \
    -Dversion=[VERSION]
```

A Maven Workspace is generated in the current folder. No other tools or CLIs are required for Maven Workspace.

Anatomy

The default Maven Workspace contains the following folders/files:

- `[MAVEN_WORKSPACE]`
    - `configs`
        - `common`
        - `dev`
        - `local`
        - `prod`
        - `uat`
    - `modules`
        - `pom.xml`
    - `themes`
        - `pom.xml`
    - `wars`
        - `pom.xml`

    - pom.xml

For more information on the configs folder, see the Testing Modules section. The modules, themes, and wars folders hold projects of that type. The parent pom.xml configures your workspace as a Maven project and applies the Bundle Support plugin, which is required for your Maven Workspace to handle 7.0 projects. You can also configure workspace properties in your POM, which you'll learn about later.

Next, you'll learn how to initialize and package Liferay DXP bundles using workspace.

Adding a Liferay Bundle to a Maven Workspace

Liferay Maven Workspaces can generate and hold a Liferay Server. This lets you build/test your plugins against a running Liferay instance. Before generating a Liferay instance, open the pom.xml file located in your workspace's root folder and set the version of the Liferay bundle to generate and install by setting the download URL for the liferay.workspace.bundle.url property. For example,

```
<properties>
    <liferay.workspace.bundle.url>
        https://releases-cdn.liferay.com/portal/7.0.6-ga7/liferay-ce-portal-tomcat-7.0-ga7-20180507111753223.zip
    </liferay.workspace.bundle.url>
    ...
</properties>
```

You can also set location of your Liferay bundle with the liferay.workspace.home.dir property. It's set to bundles by default.

Important: Make sure the com.liferay.portal.tools.bundle.support plugin in your POM is configured to use version 3.2.0+. The liferay.workspace.bundle.url property does not work for workspaces using an older version of the Bundle Support plugin. See the Updating a Maven Workspace section for instructions on how to update the plugin.

Once you've finalized your workspace properties, navigate to your workspace's root folder and run

```
blade server init
```

This uses workspace's pre-bundled and installs your Liferay DXP instance in the bundles folder. Blade CLI tool to download the version of Liferay DXP you specified in your POM file and installs your Liferay DXP instance in the bundles folder. If you prefer to not use Blade CLI or do not have it installed, the pure Maven equivalent for this command is mvn bundle-support:init.

If you want to skip the downloading process, you can create the bundles folder manually in your workspace's ROOT folder and extract your Liferay Portal bundle to that folder.

You can also produce a distributable Liferay DXP bundle (Zip) from within a workspace. To do this, navigate to your workspace's root folder and run the following command:

```
mvn bundle-support:dist
```

Your distribution file is available from the workspace's /target folder.

Configuring Maven Workspace Properties

There are many configurable workspace properties you can set in the root pom.xml file:

- liferay.workspace.bundle.dest: the destination folder for downloaded Liferay DXP bundle ZIP files.
- liferay.workspace.bundle.url: the URL used to download the Liferay DXP bundle. For more information, see Adding a Liferay Bundle to a Maven Workspace.

- `liferay.workspace.default.repository.enabled`: whether Liferay CDN is set as the default repository in the root project.
- `liferay.workspace.deploy.war.dir`: the deployment folder for WAR projects.
- `liferay.workspace.deploy.modules.dir`: the deployment folder for module projects.
- `liferay.workspace.environment`: the name of a `configs` subfolder holding the Liferay DXP server configuration to use. See Testing Modules for more information.
- `liferay.workspace.home.dir`: the Liferay DXP bundle root folder.
- `liferay.workspace.modules.default.repository.enabled`: whether the Liferay CDN is set as the default repository for module projects.
- `liferay.workspace.modules.dir`: the module projects' root folder.
- `liferay.workspace.plugins.sdk.dir`: the converted Plugins SDK's root folder. For more information, see Using a Plugins SDK from Your Workspace
- `liferay.workspace.themes.dir`: the theme projects' root folder.
- `liferay.workspace.wars.dir`: the WAR projects' root folder.

Properties can be set by adding tags with the property name. See the property configurations below for an example on how these can be set in your POM:

```
<properties>
    <liferay.workspace.home.dir>${liferay.workspace.basedir}/bundles</liferay.workspace.home.dir>
    <liferay.workspace.bundle.dest>${user.home}/.liferay/bundles/liferay-ce-portal-tomcat-7.0-ga7-20180507111753223.zip</liferay.workspace.bundle.dest>
    <liferay.workspace.bundle.url>https://releases-cdn.liferay.com/portal/7.0.6-ga7/liferay-ce-portal-tomcat-7.0-ga7-20180507111753223.zip</liferay.workspac
    <liferay.workspace.deploy.war.dir>${liferay.workspace.home.dir}/osgi/war</liferay.workspace.deploy.war.dir>
    <liferay.workspace.deploy.modules.dir>${liferay.workspace.home.dir}/osgi/modules</liferay.workspace.deploy.modules.dir>
    <liferay.workspace.environment>local</liferay.workspace.environment>
</properties>
```

Next, you'll learn how to add and deploy modules/projects in your Maven Workspace.

Module Management

Maven Workspace makes managing your Maven project easier than ever. To create a project, navigate to the appropriate workspace folder for that type of project (e.g., `modules`, `wars`, etc.). Then generate the project archetype. You can view a full listing of the available archetypes in the Project Templates reference section. Once the project is generated, it can leverage all of Maven Workspace's functionality.

Maven Workspace also lets you deploy your projects to Liferay DXP using Maven. See the Deploying a Module Built with Maven to Liferay DXP tutorial for more information.

Want to leverage Maven Workspace's testing infrastructure so you can simulate your Maven projects in a specific environment? See the Testing Modules section for more information.

Once you have your Maven projects solidified and ready for the limelight, it'd be great to release your projects to the public. Maven Workspace doesn't provide this functionality, but there are easy ways to use external release tools with workspace. See the Releasing Modules section for more information.

Next, you'll learn how to update a Maven Workspace.

Updating a Maven Workspace

Liferay Workspace is updated periodically with new features, so you'll want to update your workspace instance accordingly. To update your Maven Workspace, you must update the Bundle Support plugin configured in your workspace's root `pom.xml` file:

```
<plugin>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.portal.tools.bundle.support</artifactId>
    <version>3.2.5</version>
    ...
</plugin>
```

Update the version to the latest available release. You can reference the available releases for the Bundle Support plugin here.

Chapter 57

IntelliJ IDEA

The Liferay IntelliJ plugin provides support for Liferay DXP development in IntelliJ IDEA. Liferay's IntelliJ plugin provides the following built-in features:

- Creating a Liferay Workspace (Gradle and Maven based)
- Creating Liferay projects (Gradle and Maven based)
- Liferay DXP Tomcat server support for project deployment and debugging
- Support for adding line markers for each entity in the service editor
- Syntax checking, highlighting, and code completion (e.g., Bnd and XML files)

In these tutorials, you'll learn how to install the Liferay IntelliJ plugin and leverage its features to improve Liferay development with IntelliJ IDEA.

57.1 Installing the Liferay IntelliJ Plugin

To install the Liferay IntelliJ plugin in IntelliJ IDEA, follow the steps below:

1. Navigate to the JetBrains' Liferay IntelliJ plugin page and download it to your local machine.

2. In IntelliJ, navigate to *File* → *Settings* → *Plugins* and select *Install plugin from disk...*.

3. Select the Liferay IntelliJ plugin ZIP and select *OK* → *OK* → *Restart*.

Once IntelliJ restarts, the Liferay IntelliJ plugin is installed and ready for use. Great job! You're now ready to develop for Liferay DXP in IntelliJ!

57.2 Creating a Liferay Workspace with IntelliJ IDEA

In this tutorial, you'll learn how to generate a Liferay Workspace using IntelliJ IDEA, which runs on Blade CLI behind the scenes. IntelliJ gives you a graphical interface instead of the command prompt, which can streamline your workflow. You'll also learn how to import an existing Liferay Workspace into IntelliJ. To learn more about Liferay Workspaces, visit its dedicated tutorial section.

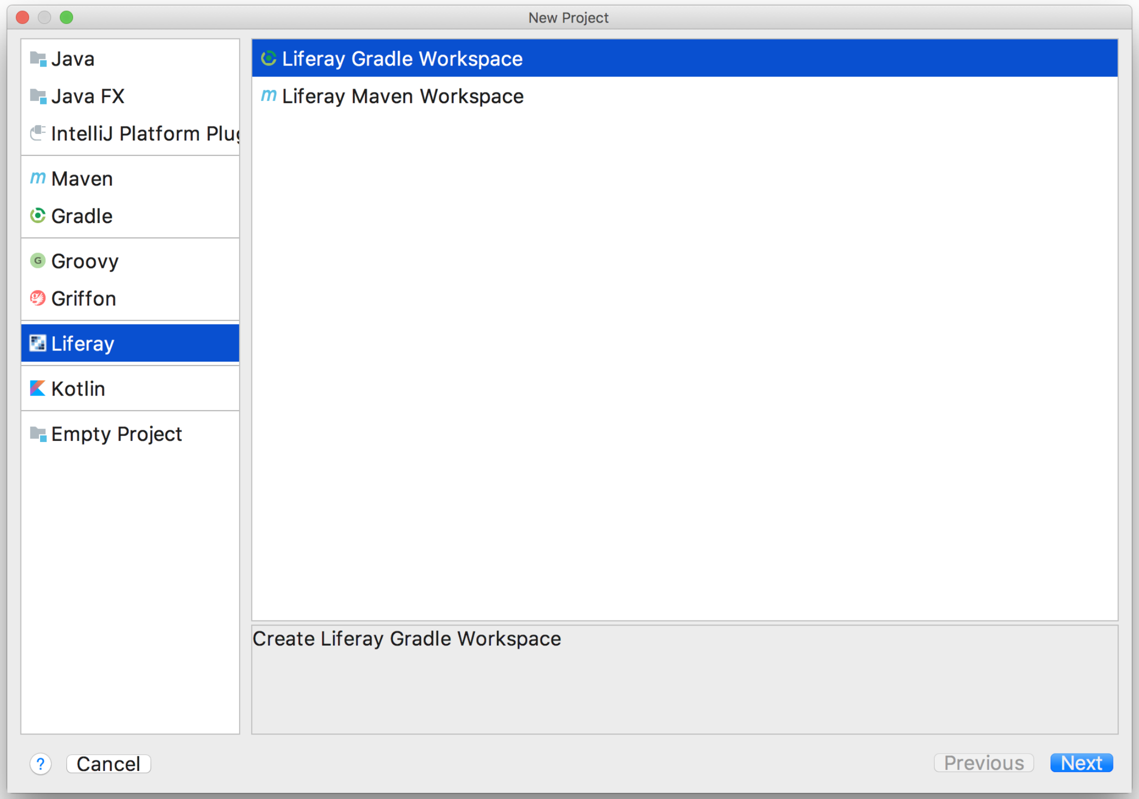
Creating a Liferay Workspace

Follow the steps below to create a Liferay Workspace:

1. Open the New Project wizard by selecting *File* → *New* → *Project*. If you're starting IntelliJ for the first time, you can do this by selecting *Create New Project* in the opening window.

2. Select *Liferay* from the left menu.

3. Choose the build type for your workspace (i.e., Gradle or Maven). Then click *Next*.



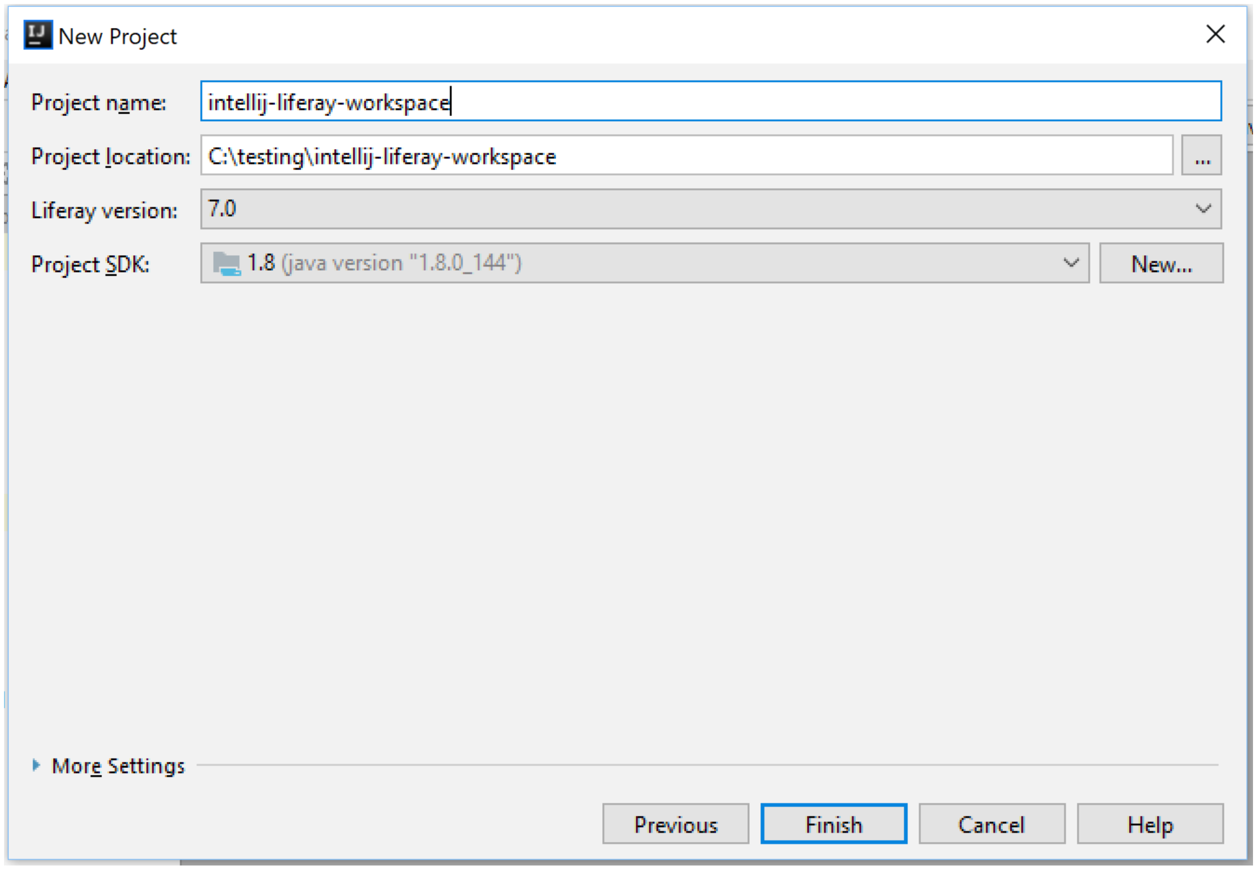
Create Liferay Gradle Workspace

Figure 57.1: Choose *Liferay Gradle Workspace* or *Liferay Maven Workspace*, depending on the build you prefer.

4. Specify your workspace's name, location, intended Liferay DXP version, and SDK (i.e., Java JDK). Then click *Finish*.

5. A window opens for additional build configurations for the build type you selected (i.e., Gradle or Maven). Verify the settings and click *OK*.

Awesome! You've successfully created a Liferay Workspace in IntelliJ IDEA!


New Project

Project name: intellij-liferay-workspace

Project location: C:\testing\intellij-liferay-workspace

Liferay version: 7.0

Project SDK: 1.8 (java version "1.8.0_144")

Figure 57.2: Specify your workspace's configurations.

Importing a Liferay Workspace

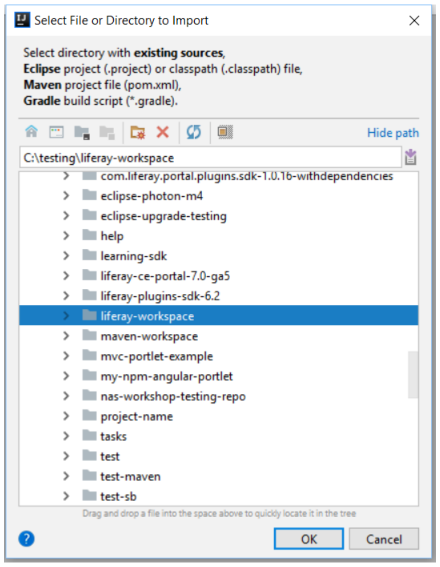
To import an existing workspace into IntelliJ, follow the steps below:

1. Select *File* → *New* → *Project from Existing Sources...*.
2. Select the workspace you want to import. Then click *OK*.
3. Click the *Import project from external model* radio button and select the build tool your workspace uses (e.g., Gradle or Maven).
4. Configure the project import (if necessary) and then click *Finish*. See the Import a Project section of IntelliJ's official documentation for more information.
5. Step through the remaining import prompts and then open your imported workspace as you desire (i.e., in the current window or a new window).

Excellent! Your existing Liferay Workspace is now imported in IntelliJ IDEA!

57.3 Creating Projects with IntelliJ IDEA

IntelliJ IDEA provides a New Liferay Modules wizard to create a variety of different module projects. You can also use the same wizard to create theme projects, WAR-style projects, and more. Before beginning,


Select File or Directory to Import

Select directory with **existing sources**,
Eclipse project (.project) or classpath (.classpath) file,
Maven project file (pom.xml),
Gradle build script (*.gradle).

C:\testing\liferay-workspace

- com.liferay.portal.plugins.sdk-1.0.16-withdependencies
- eclipse-photon-m4
- eclipse-upgrade-testing
- help
- learning-sdk
- liferay-ce-portal-7.0-ga5
- liferay-plugins-sdk-6.2
- liferay-workspace
- maven-workspace
- mvc-portlet-example
- my-npm-angular-portlet
- nas-workshop-testing-repo
- project-name
- tasks
- test
- test-maven
- test-sb

Drag and drop a file into the space above to quickly locate it in the tree

Figure 57.3: Specify your workspace's configurations.

ensure you've created/imported a Liferay Workspace in your IntelliJ environment. Follow the steps below to create a Liferay DXP module:

1. Navigate to *File* → *New* → *Liferay Module*.

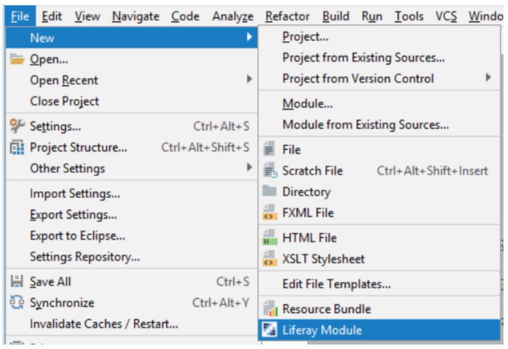


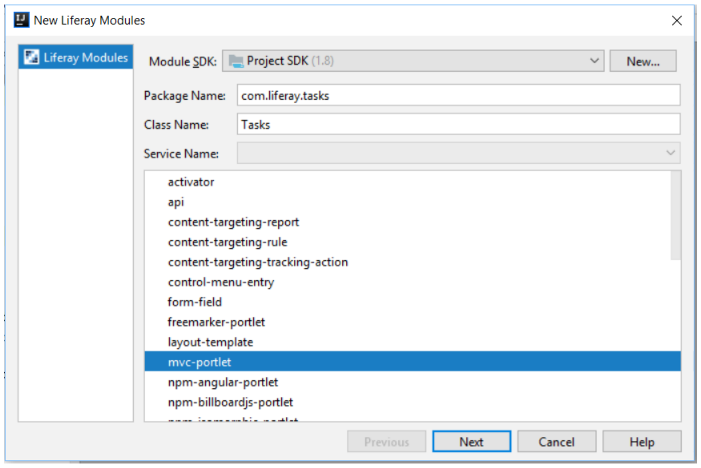
Figure 57.4: Selecting *Liferay Module* opens the New Liferay Modules wizard.

2. Select the project you want to create. Although the wizard characterizes itself for *modules*, there are many available projects that are not OSGi-based modules (e.g., `theme`, `war-mvc-portlet`, etc.). See the Project Templates reference section for more information on the available templates.

3. Configure your project's SDK (i.e., JDK), package name, class name, and service name, if necessary. Then click *Next*.

4. Give your project a name. Then click *Finish*.

Awesome! Your project is available under its project type folder in your workspace.


New Liferay Modules

Liferay Modules

Module SDK: Project SDK (1.8)

Package Name: com.liferay.tasks

Class Name: Tasks

Service Name:

activator
api
content-targeting-report
content-targeting-rule
content-targeting-tracking-action
control-menu-entry
form-field
freemarker-portlet
layout-template
mvc-portlet
npm-angular-portlet
npm-billboardjs-portlet

Figure 57.5: Choose the project template to create your module.

57.4 Installing a Server in IntelliJ IDEA

Installing a Liferay server in IntelliJ is easy. In just a few steps, you'll have your server up and running.

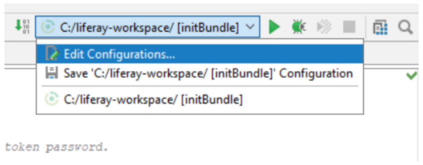
Note: Tomcat and Wildfly are the only supported Liferay app server bundles available to install in IntelliJ.

Follow these steps to install your server:

1. Right-click your Liferay workspace and select *Liferay → InitBundle*.

   This downloads the Liferay DXP bundle specified in your workspace's `gradle.properties` file. You can change the default bundle by updating the `liferay.workspace.bundle.url` property. For example, this is required to update the default bundle version and/or type (e.g., Wildfly). The downloaded bundle is stored in the workspace's `bundles` folder.

2. Navigate to the top right Configurations dropdown menu and select *Edit Configurations*. From here, you can configure your server's run and debug configurations.


C:/liferay-workspace/ [initBundle]

Edit Configurations...

Save 'C:/liferay-workspace/ [initBundle]' Configuration

C:/liferay-workspace/ [initBundle]

token password.

Figure 57.6: You have several options to choose from the server dropdown menu.

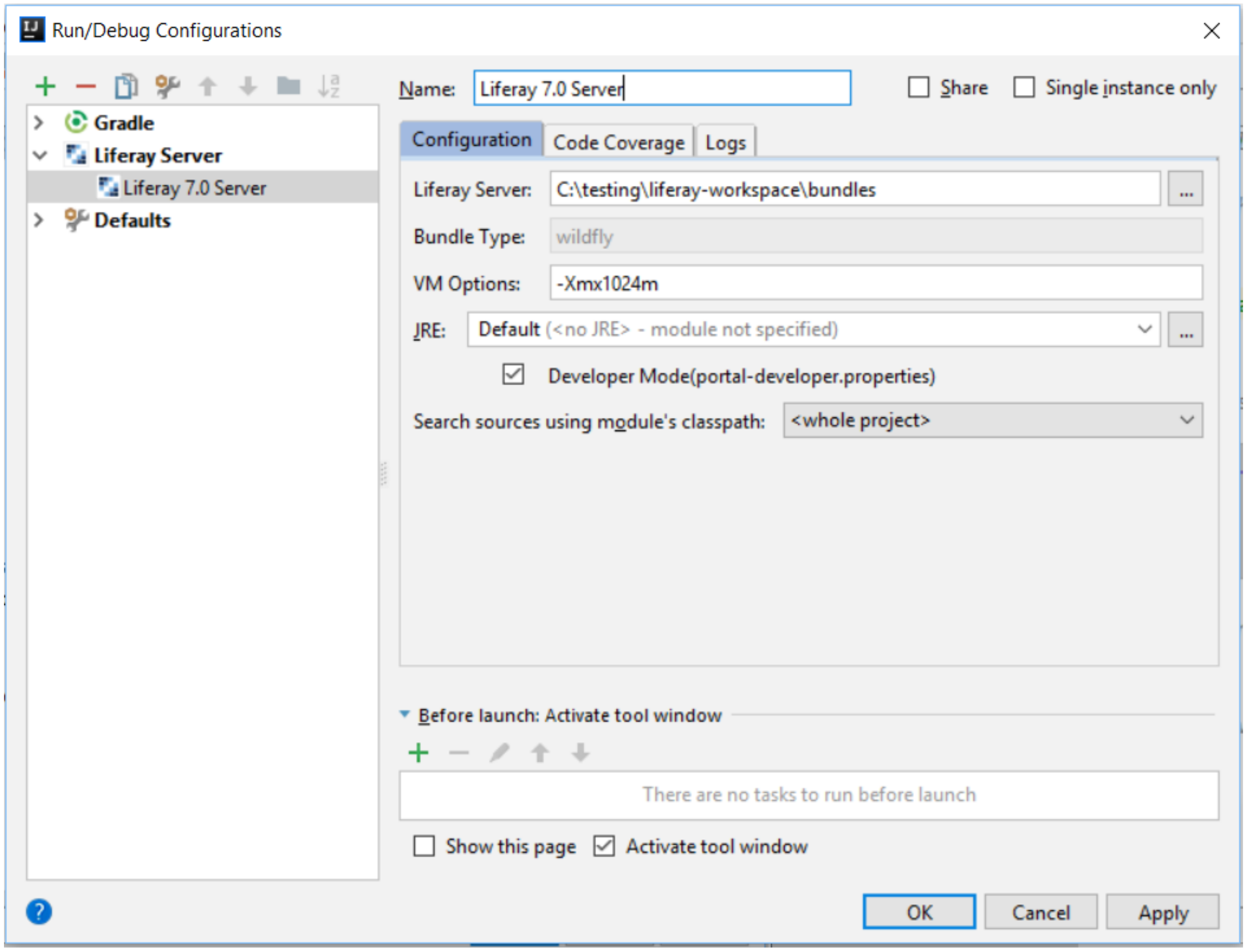
3. Click the *Add New Configuration* button (+) and select *Liferay Server*.

4. Give your server a better name and modify any other configurations, if necessary. Then select *OK* .

Your server is now available in IntelliJ! Make sure to select it in the Configurations dropdown before executing the configuration buttons (below).

For reference, here's how the IntelliJ configuration buttons work with your Liferay DXP instance:

- *Start* (▶): Starts the server.



Run/Debug Configurations

Name: Liferay 7.0 Server ☐ Share ☐ Single instance only

Configuration | Code Coverage | Logs

Liferay Server: C:\testing\liferay-workspace\bundles

Bundle Type: wildfly

VM Options: -Xmx1024m

JRE: Default (<no JRE> - module not specified)

☑ Developer Mode(portal-developer.properties)

Search sources using module's classpath: <whole project>

Before launch: Activate tool window

There are no tasks to run before launch

☐ Show this page ☑ Activate tool window

Figure 57.7: Set your Liferay server's configurations in the Run/Debug Configurations menu.

- *Stop* (■): Stops the server.
- *Debug* (●): Starts the server in debug mode. For more information on debugging in IntelliJ, see the IntelliJ Debugging article.

Now you're ready to use your server in IntelliJ!

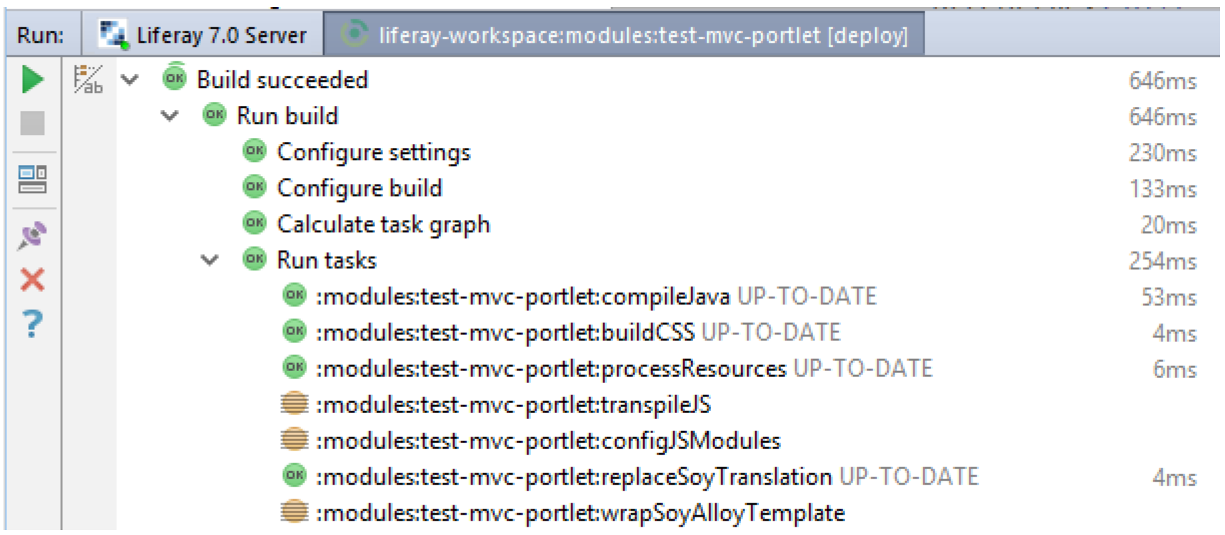
57.5 Deploying Projects with IntelliJ IDEA

Once you've created a project and installed your Liferay server in IntelliJ, you'll want to deploy your project. Follow the steps below to do this:

1. Right-click your project from within the Liferay Workspace folder structure and select *Liferay* → *Deploy*.

   This automatically loads a build progress window viewable at the bottom of your IntelliJ instance.

2. Verify that your project builds successfully from the build progress window. Then navigate back to your server's window and confirm it starts in your configured Liferay DXP instance. You should receive a message like this:



```
Build succeeded                                                       646ms
  Run build                                                           646ms
    Configure settings                                                230ms
    Configure build                                                   133ms
    Calculate task graph                                               20ms
    Run tasks                                                         254ms
      :modules:test-mvc-portlet:compileJava UP-TO-DATE                 53ms
      :modules:test-mvc-portlet:buildCSS UP-TO-DATE                     4ms
      :modules:test-mvc-portlet:processResources UP-TO-DATE             6ms
      :modules:test-mvc-portlet:transpileJS
      :modules:test-mvc-portlet:configJSModules
      :modules:test-mvc-portlet:replaceSoyTranslation UP-TO-DATE        4ms
      :modules:test-mvc-portlet:wrapSoyAlloyTemplate
```

Figure 57.8: Verify that your project build successfully.

```
INFO  [fileinstall-C:/liferay-workspace/bundles/osgi/modules][BundleStartStopLogger:35] STARTED com.liferay.docs_1.0.0 [652]
```

That's it! You've successfully deployed your project to Liferay DXP!

Chapter 58

Liferay Sample Projects

Liferay provides working examples of sample projects that target different integration points in Liferay DXP. These working examples can be copy/pasted into your own independent project so you can take advantage of various Liferay extension points. Each sample is a standalone project and includes its own build files. Liferay's sample projects can be found in the liferay-blade-samples repository on GitHub. You can find documentation for Liferay's sample projects in the Sample Projects reference section.

If you'd like to browse the repository locally or copy sample projects into your own project, fork and clone the `liferay-blade-samples` repository.

At first glance, you'll notice that the repository is broken up into three primary folders:

- `gradle`
- `liferay-workspace`
- `maven`

The provided sample projects are organized by their development toolchains to cater to a variety of developers. Each folder offers the same set of sample Liferay projects. Their only difference is that the build files are specific to their toolchain. For example, the `gradle` folder contains projects using standard OSS Gradle plugins that can be added to any Gradle composite build. The same concept also applies to the `liferay-workspace` and `maven` projects.

The `gradle` folder also uses the Liferay Gradle plugin (e.g., `com.liferay.plugin`) which encompasses additional functionality for various types of Liferay projects. The Liferay Gradle plugin is recommended for Gradle users developing for Liferay.

Some samples also come configured with logging to help you fully understand what the sample is accomplishing behind the scenes. For example, OSGi module logging is implemented for several samples (e.g., action-command-portlet, document-action, service-builder/jdbc, etc.), which lets OSGi modules supply their own logging configuration defaults without external configuration. See the Adjusting Module Logging tutorial for more information.

For a list of sample template projects available, visit the Liferay extension points sub-section in the Liferay Blade Samples repository. This list is not comprehensive. A subset of missing extension point samples can be found in the Liferay extension points without template projects sub-section. Visit the repo's Contribution Guidelines section for details on contributing to this repository.

58.1 Liferay Upgrade Planner

The Liferay Upgrade Planner provides an automated way to adapt your installation's data and legacy plugins to your desired Liferay DXP upgrade version. We recommend leveraging this tool for any of the following upgrades:

- Liferay Portal 6.2 → Liferay DXP 7.0, 7.1, or 7.2
- Liferay DXP 7.0 → Liferay DXP 7.1 or 7.2
- Liferay DXP 7.1 → Liferay DXP 7.2

The Upgrade Planner is provided in Liferay Dev Studio (versions 3.6+). Here's what the Upgrade Planner does:

- Updates your development environment.
- Identifies code affected by the API changes.
- Describes each API change related to the code.
- Suggests how to adapt the code.
- Provides options, in some cases, to adapt code automatically.
- Transfers database and server data to your new environment.

Even if you prefer tools other than Dev Studio (which is based on Eclipse), you should upgrade your data and legacy plugins using the Upgrade Planner first–you can use your favorite tools afterward.

To start the Upgrade Planner in Dev Studio, do this:

1. Navigate to *Project* → *New Liferay Upgrade Plan…*.

2. In the New Liferay Upgrade Plan wizard, assign your plan a name and choose an upgrade plan outline. The data and code upgrade processes are separate, so you must step through each process independently.

3. Choose your current Liferay version and the new version you're upgrading to.

4. If you chose to complete a code upgrade, you must also select the folder where your legacy plugins reside (e.g., Plugins SDK for Liferay 6.2 projects).

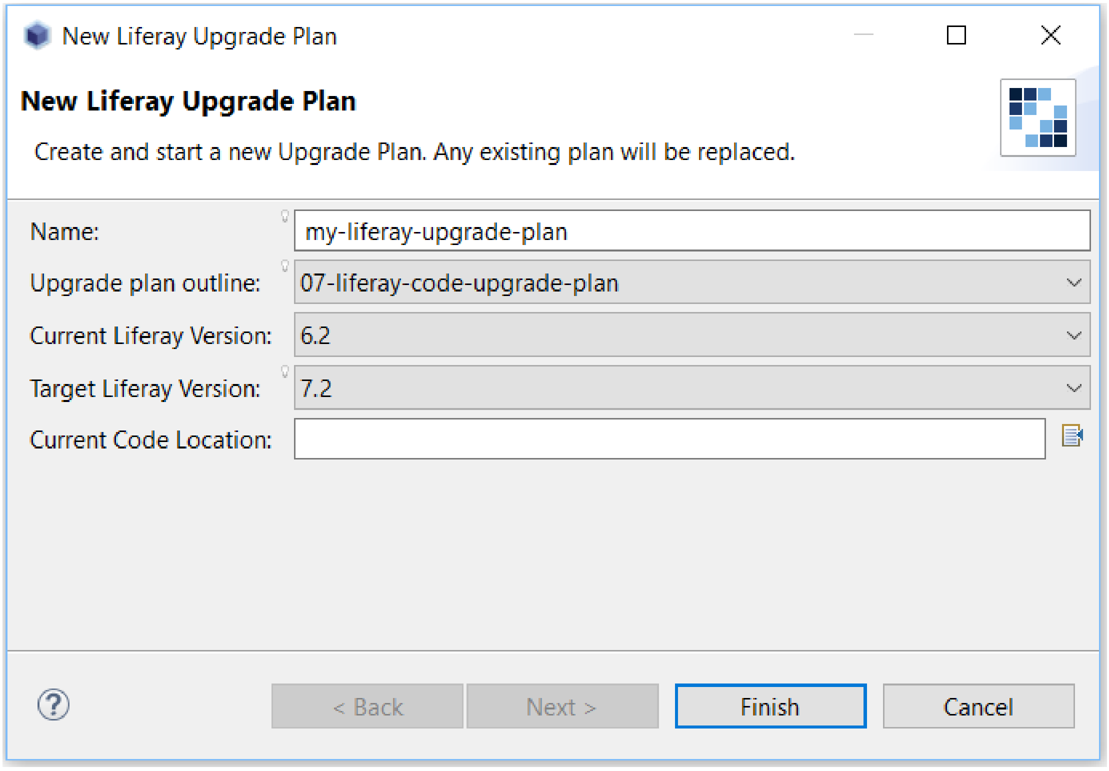
5. Click *Finish*.

Switch to the new Liferay Upgrade Planner perspective (prompted automatically). You're now offered several windows in the UI:

- *Project Explorer:* displays your legacy plugin environment and new development environment. It also displays your upgrade problems that are detected during the *Fix Upgrade Problems* step.
- *Liferay Upgrade Plan:* outlines the upgrade plan's steps and step summaries.
- *Liferay Upgrade Plan Info:* shows official documentation that describes the upgrade step.

To progress through your upgrade plan, click the steps outlined in the Liferay Upgrade Plan window. Each step can have several options:

- *Click to preview:* previews what an automated step will perform.
- *Click to perform:* executes an automated process provided with the step. This is only offered for steps where the Upgrade Planner can assist.



New Liferay Upgrade Plan

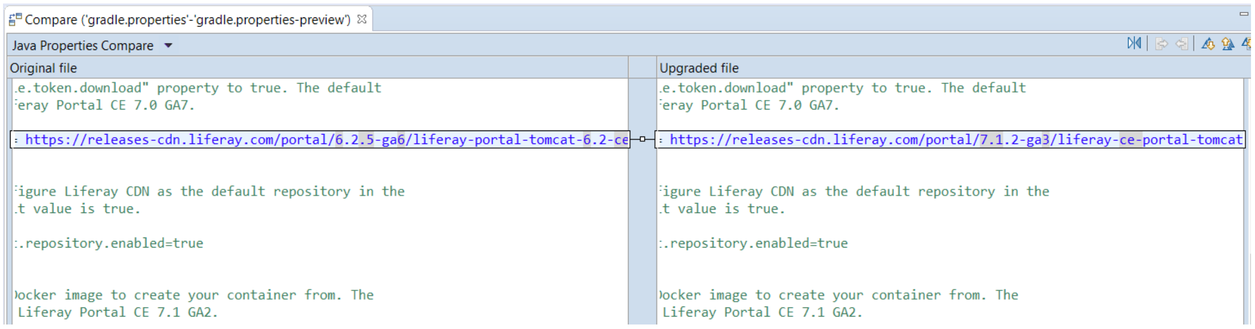
New Liferay Upgrade Plan

Create and start a new Upgrade Plan. Any existing plan will be replaced.

Name:	my-liferay-upgrade-plan
Upgrade plan outline:	07-liferay-code-upgrade-plan
Current Liferay Version:	6.2
Target Liferay Version:	7.2
Current Code Location:	

Figure 58.1: Configure your upgrade plan before beginning the upgrade process.

- *Click when complete:* marks the step as complete. This is only offered when the Upgrade Planner cannot provide automated assistance and, instead, only offers documentation to assist in completing the step manually.
- *Restart:* marks a completed step as unfinished. The step is performed again if automation is involved.
- *Skip:* skips the step and jumps to the next step in the outline.



Compare ('gradle.properties'-'gradle.properties-preview')

Original file

```
.e.token.download" property to true. The default
'eray Portal CE 7.0 GA7.

: https://releases-cdn.liferay.com/portal/6.2.5-ga6/liferay-portal-tomcat-6.2-ce


'igure Liferay CDN as the default repository in the
.t value is true.

:.repository.enabled=true


)ocker image to create your container from. The
 Liferay Portal CE 7.1 GA2.
```

Upgraded file

```
.e.token.download" property to true. The default
'eray Portal CE 7.0 GA7.

: https://releases-cdn.liferay.com/portal/7.1.2-ga3/liferay-ce-portal-tomcat


'igure Liferay CDN as the default repository in the
.t value is true.

:.repository.enabled=true


)ocker image to create your container from. The
 Liferay Portal CE 7.1 GA2.
```

Figure 58.2: You can preview the Upgrade Planner's automated updates before you perform them.

Great! You now have a good understanding of the Liferay Upgrade Planner's UI and how to get started.

Note: The Upgrade Planner upgrades data and code to Liferay DXP versions that include 7.0 and the latest DXP version. It links to the latest Liferay DXP upgrade documentation. 7.0 upgrade documentation is available here:

- Data Upgrade
- Code Upgrade

58.2 Using the Upgrade Planner with Proxy Requirements

If you have proxy server requirements and want to configure your http(s) proxy
to work with the Liferay Upgrade Planner, follow the instructions below.

1. In Dev Studio's `DeveloperStudio.ini/eclipse.ini` file, add the following parameters:

   ```
   -Djdk.http.auth.proxying.disabledSchemes=
   -Djdk.http.auth.tunneling.disabledSchemes=
   ```

2. Launch Dev Studio.

3. Go to *Window* → *Preferences* → *General* → *Network Connections*.

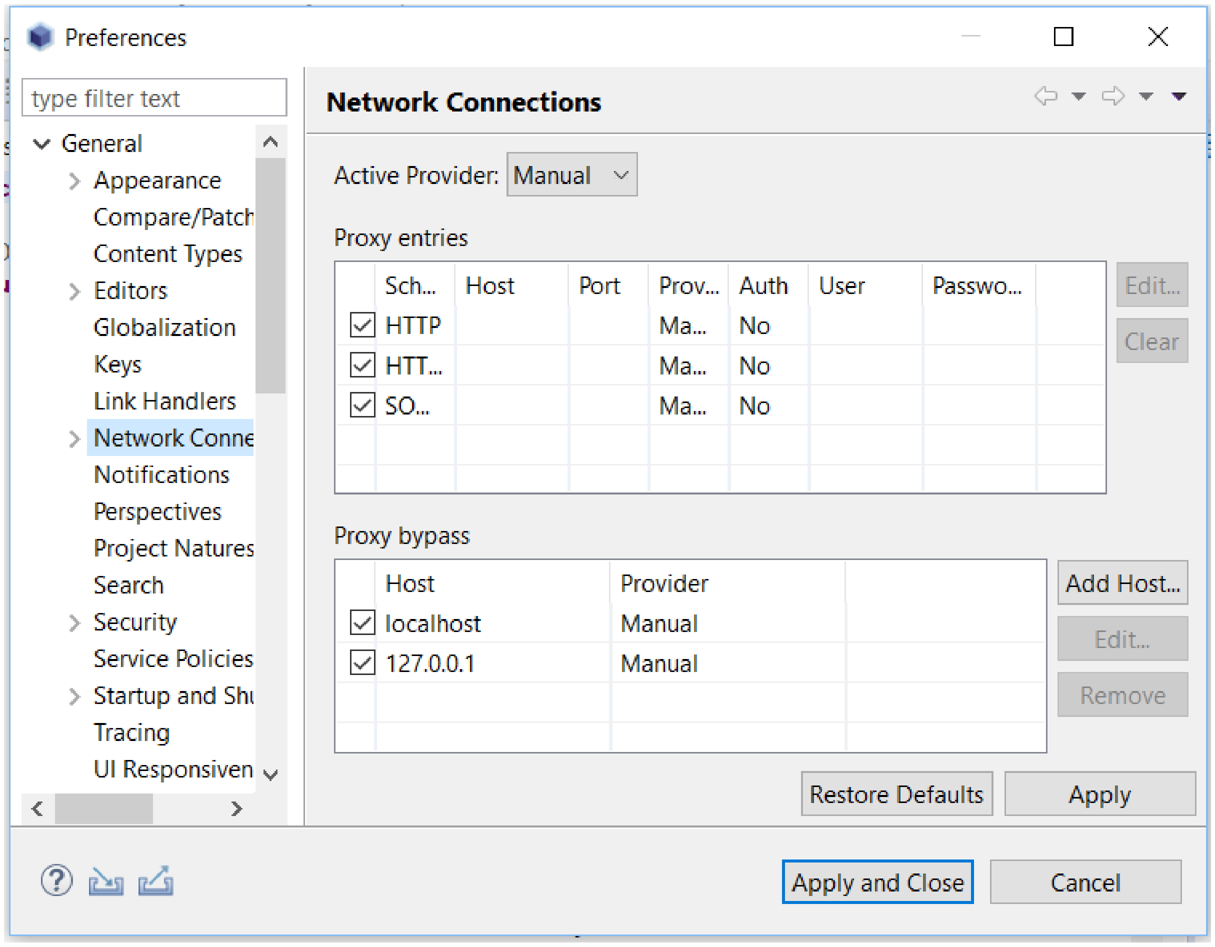
4. Set the *Active Provider* drop-down selector to *Manual*.

5. Under *Proxy entries*, configure both proxy HTTP and HTTPS by clicking the field and selecting the *Edit* button.

6. For each schema (HTTP and HTTPS), enter your proxy server's host, port, and authentication settings (if necessary). Do not leave whitespace at the end of your proxy host or port settings.

7. Once you've configured your proxy entry, click *Apply and Close*.

Awesome! You've successfully configured the Upgrade Planner's proxy settings!



Active Provider: Manual

Proxy entries

	Sch...	Host	Port	Prov...	Auth	User	Passwo...
✓	HTTP			Ma...	No		
✓	HTT...			Ma...	No		
✓	SO...			Ma...	No		

Proxy bypass

	Host	Provider
✓	localhost	Manual
✓	127.0.0.1	Manual

Figure 58.3: You can configure your proxy settings in Dev Studio's Network Connections menu.

Chapter 59

Portlets

Web apps in Liferay DXP are called *portlets*. Like many web apps, portlets process requests and generate responses. In the response, the portlet returns content (e.g. HTML, XHTML) for display in browsers. You might now be thinking, "Ok, besides the funky name, how are portlets different from other types of web apps?" This is a fantastic question! One key difference is that portlets run in a portion of the web page. When you're writing a portlet application, you only need to worry about that application: the rest of the page–the navigation, the top banner, and any other global component of the interface–is handled by other components. Another difference is that portlets run only in a portal server, like the one in Liferay DXP. Portlets can therefore use the portal's existing support for user management, authentication, permissions, page management, and more. This frees you to focus on developing the portlet's core functionality. In many ways, writing your application as a portlet is easier than writing a standalone application.

Portlets can be placed on pages by users or portal administrators, who can place several different portlets on a single page. For example, a page in a community site could have a calendar portlet for community events, an announcements portlet for important announcements, and a bookmarks portlet for links of interest to the community. And because the portal controls page layout, you can reposition and resize one or more portlets on a page without altering any portlet code. Doing all this in other types of web apps would require manual re-coding. Alternatively, a single portlet can take up an entire page if it's the only app you need on that page. For example, a message boards or wiki portlet is best suited on its own page. In short, portlets alleviate many of the traditional pain points associated with developing web apps.

What's more, portals and portlets are standards-based. In 2003, Java Portlet Specification 1.0 (JSR-168) first defined portal and portlet behavior. In 2008, Java Portlet Specification 2.0 (JSR-286) refined and built on JSR-168, while maintaining backwards compatibility, to define features like inter-portlet communication (IPC) and more. The recently released Java Portlet Specification 3.0 (JSR-362) continues portal and portlet evolution. Liferay leads in this space by having a member in the Expert Group.

So what do these specifications define? We won't bore you with the gory details; if that's what you want you can read the specifications. We will tell you, however, how portlets differ from other types of servlet-based web apps. Portlets handle requests in multiple phases. This makes portlets much more flexible than servlets. Each portlet phase executes different operations:

- **Render:** Generates the portlet's contents based on the portlet's current state. When this phase runs on one portlet, it also runs on all other portlets on the page. The Render phase runs when any portlets on the page complete the Action or Event phases.

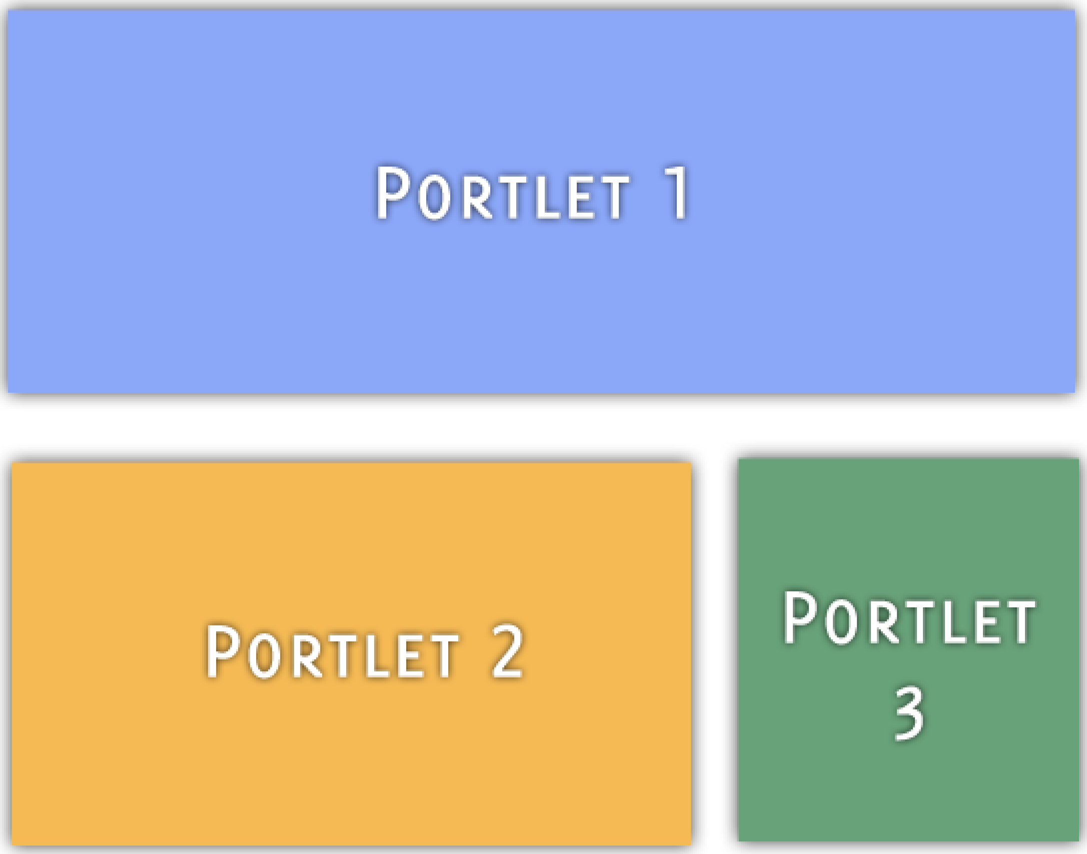


Figure 59.1: You can place multiple portlets on a single page.

- **Action:** In response to a user action, performs some operation that changes the portlet's state. The Action phase can also trigger events that are processed by the Event phase. Following the Action phase and optional Event phase, the Render phase then regenerates the portlet's contents.
- **Event:** Processes events triggered in the Action phase. Events are used for IPC. Once the portlet processes all events, the portal calls the Render phase on all portlets on the page.
- **Resource-serving:** Serves a resource independent from the rest of the lifecycle. This lets a portlet serve dynamic content without running the Render phase on all portlets on a page. The Resource-serving phase handles AJAX requests.

Compared to servlets, portlets also have some other key differences. Since portlets only render a portion of a page, tags like `<html>`, `<head>`, and `<body>` aren't allowed. And because you don't know the portlet's page ahead of time, you can't create portlet URLs directly. Instead, the portlet API gives you methods to create portlet URLs programmatically. Also, because portlets don't have direct access to the `javax.servlet.ServletRequest`, they can't read query parameters directly from a URL. Portlets instead access a `javax.portlet.PortletRequest` object. The portlet specification only provides a mechanism for a portlet to read its own URL parameters or those declared as public render parameters. Liferay DXP does, however, provide utility methods that can access the `ServletRequest` and query parameters. Portlets also have a *portlet filter* available for each phase in the portlet lifecycle. Portlet filters are similar to servlet filters in that they allow request and response modification on the fly.

Portlets also differ from servlets by having distinct modes and window states. Modes distinguish the portlet's current function:

- **View mode:** The portlet's standard mode. Use this mode to access the portlet's main functionality.
- **Edit mode:** The portlet's configuration mode. Use this mode to configure a custom view or behavior. For example, the Edit mode of a weather portlet could let you choose a location to retrieve weather data from.
- **Help mode:** A mode that displays the portlet's help information.

Most modern applications use View Mode only.

Portlet window states control the amount of space a portlet takes up on a page. Window states mimic window behavior in a traditional desktop environment:

- **Normal:** The portlet can be on a page that contains other portlets. This is the default window state.
- **Maximized:** The portlet takes up an entire page.
- **Minimized:** Only the portlet's title bar shows.

When you develop portlets for Liferay DXP, you can leverage all the features defined by the portlet specification. Depending on how you develop and package your portlet, however, it may not be able to run on other portal containers. You may now be saying, "Hold on a minute! I thought Liferay DXP was standards-compliant? What gives?" Liferay DXP is standards-compliant, but it contains some sweeteners in the form of APIs designed to make developers' lives easier. For example, Liferay DXP contains an MVC framework that makes it simpler to implement MVC in your portlet. This framework, however, is only available in Liferay's portal. Without modification, a portlet that uses this framework won't run if deployed to a non-Liferay portal container. Note, though, that we don't force you to use Liferay DXP's MVC framework or any of its other unique APIs. For example, you can develop your portlet with strictly standards-compliant frameworks and APIs, package it in a WAR file, and then deploy it on any standards-compliant portal container.

Liferay DXP also contains an OSGi runtime. This means that you don't have to develop and deploy your portlet as a traditional WAR file; you can do so as OSGi modules instead. We recommend the latter, so you can take advantage of the modularity features inherent in OSGi. For a detailed description of these features, see the tutorial OSGi and Modularity. Note, however, that Liferay DXP portlets you develop as OSGi modules won't run on other portlet containers that lack an OSGi runtime. Even so, the advantages of modularity are so great that we still recommend you develop your portlets as OSGi modules.

With that said, you can use a variety of technologies to develop portlets that run on Liferay DXP. Have you ever heard the saying, "There's more than one way to skin a cat?" It's gross, but it's probably true. Liferay DXP doesn't force you to use a single tool or set of tools to develop portlets. This section shows you how to develop portlets using the following frameworks and techniques:

- Liferay's `MVCPortlet`
- Soy Portlet
- Spring MVC
- JavaServer Faces (JSF) Portlets with Liferay Faces
- Making URLs Friendlier
- Preparing Your JavaScript Files for ES2015
- Applying Lexicon Styles to Your App

- Automatic Single Page Applications
- Creating Layouts Inside Custom Portlets

Chapter 60

Liferay MVC Portlet

Web applications are often developed following the Model View Controller (MVC) pattern. But Liferay has developed a groundbreaking new pattern called the *Modal Veal Contractor* (MVC) pattern. Okay, that's not true: the framework is actually another implementation of Model View Controller. If you're an experienced developer, this will not be the first time you've heard about Model View Controller. In this article you'll need to stay focused, because there will be several attempts to show you why Liferay's implementation of Model View Controller is different, when instead you're hearing about another MVC framework. With that in mind, let's get back to the *Medial Vein Constriction* pattern we were discussing.

If there are so many implementations of MVC frameworks in Java, why did Liferay create yet another one? Stay with us and you'll see that Liferay MVC provides these benefits:

- It's lightweight, as opposed to many other Java MVC frameworks.
- There are no special configuration files that need to be kept in sync with your code.
- It's a simple extension of `GenericPortlet`.
- You avoid writing a bunch of boilerplate code, since Liferay's MVC framework simply looks for some pre-defined parameters when the `init()` method is called.
- The controller can be broken down into MVC command classes, each of which handles the controller code for a particular portlet phase (render, action, and resource serving phases).
- Liferay's portlets use it. That means there are plenty of robust implementations to reference when you need to design or troubleshoot your Liferay applications.

The Liferay MVC portlet framework is light, it hides part of the complexity of portlets, and it makes the most common operations easier. The default `MVCPortlet` project uses separate JSPs for each portlet mode: For example, `edit.jsp` is for *edit* mode and `help.jsp` is for *help* mode.

Before diving in to the Liferay MVC swimming pool with all the other cool kids (applications), review how each layer of the *Moody Vase Conscription* pattern helps you separate the concerns of your application.

60.1 MVC Layers and Modularity

In MVC, there are three layers, and you can probably guess what they are.

Model: The model layer holds the application data and logic for manipulating it.

View: The view layer contains logic for displaying data.

Controller: The middle man in the MVC pattern, the Controller contains logic for passing the data back and forth between the view and the model layers.

The *Middle Verse Completer* pattern fits well with Liferay's application modularity effort.

Liferay's applications are divided into multiple discrete modules. With Service Builder, the model layer is generated into a `service` and an `api` module. That accounts for the model in the MVC pattern. The view and the controller share a module, the `web` module.

Generating the skeleton for a multi-module Service Builder driven MVC application using Liferay Blade CLI saves you lots of time and gets you started on the more important (and interesting, if we're being honest) development work.

60.2 Liferay MVC Command Classes

In a larger application, your `-Portlet` class can become monstrous and unwieldy if it holds all of the controller logic. Liferay provides MVC command classes to break up your controller functionality.

- **`MVCActionCommand`:** Use `-ActionCommand` classes to hold each of your portlet actions, which are invoked by action URLs.
- **`MVCRenderCommand`:** Use `-RenderCommand` classes to hold a `render` method that dispatches to the appropriate JSP, by responding to render URLs.
- **`MVCResourceCommand`:** Use `-ResourceCommand` classes to execute resource serving in your MVC portlet, by responding to resource URLs.

There must be some confusing configuration files to keep everything wired together and working properly, right? Wrong: it's all easily managed in the OSGi component in the `-Portlet` class.

60.3 Liferay MVC Portlet Component

Whether or not you plan to split up the controller into MVC command classes, you use a portlet component class with a certain set of properties. Here's a simple portlet component as an example:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.css-class-wrapper=portlet-hello-world",
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.icon=/icons/hello_world.png",
        "com.liferay.portlet.preferences-owned-by-group=true",
        "com.liferay.portlet.private-request-attributes=false",
        "com.liferay.portlet.private-session-attributes=false",
        "com.liferay.portlet.remoteable=true",
        "com.liferay.portlet.render-weight=50",
        "com.liferay.portlet.use-default-template=true",
        "javax.portlet.display-name=Hello World",
        "javax.portlet.expiration-cache=0",
        "javax.portlet.init-param.always-display-default-configuration-icons=true",
        "javax.portlet.name=" + HelloWorldPortletKeys.HELLO_WORLD,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=guest,power-user,user",
        "javax.portlet.supports.mime-type=text/html"
    },
    service = Portlet.class
)
public class HelloWorldPortlet extends MVCPortlet {
}
```

When using MVC commands, the `javax.portlet.name` property is important. This property is one of two that must be included in each MVC command component; it links a particular portlet URL/command combination to the correct portlet.

Important: Make your portlet name unique, considering how Liferay DXP uses the name to create the portlet's ID.

There can be some confusion over exactly what kind of `Portlet.class` implementation you're publishing with this component. Liferay's service registry expects this to be `javax.portlet.Portlet`, so make sure that's the class you import, and not, for example, `com.liferay.portal.kernel.model.Portlet`.

Note: To find a list of all the Liferay-specific attributes you can specify as properties in your portlet components, check out the liferay-portlet-app_7_0_0.dtd.

Consider the `<css-class-wrapper>` element from the above link as an example. To specify that property in your component, use this syntax in your property list:

```
"com.liferay.portlet.css-class-wrapper=portlet-hello-world",
```

The properties namespaced with `javax.portlet....` are elements of the portlet.xml descriptor.

60.4 A Simpler MVC Portlet

With all for this focus on MVC commands, you might be concerned that you'll be forced into a more complex pattern than is necessary, especially if you're developing only a small Liferay MVC application. Not so; just put all of your logic into the `-Portlet` class if you don't want to split up your MVC commands.

In simpler applications, if you don't have an MVC command to rely on, your portlet render URLs specify the path to the JSP in an `mvcPath` parameter.

```
<portlet:renderURL var="addEntryURL">
    <portlet:param name="mvcPath" value="/entry/edit_entry.jsp" />
    <portlet:param name="redirect" value="<%= redirect %>" />
</portlet:renderURL>
```

As you've seen, Liferay's *Medical Vortex Concentrator* (MVC) portlet framework gives you a well-structured controller layer that takes very little time to implement. With all your free time, you could

- Learn a new language
- Take pottery classes
- Lift weights
- Work on your application's business logic

It's entirely up to you.

60.5 Creating an MVC Portlet

You're convinced that Liferay's MVC Framework is right for you, and you want to learn how to configure it.

You'll need:

- A module that publishes a portlet component with the necessary properties.

- Controller code to handle the request and response.
- JSPs to implement your view layer.

Along the way you'll want to know how to call services from your controller and how to pass information from the view layer to the controller.

Keep in mind that you can take two paths with your Liferay MVC portlet implementation. If you have a small application that won't be heavy on controller logic (maybe just a couple of action methods), you can put all your controller code in the `-Portlet` class. If you have more complex needs (lots of actions, complex render logic to implement, or maybe even some resource serving code), consider breaking the controller into MVC Action Command classes, MVC Render Command classes, and MVC Resource Command classes.

In this tutorial you'll learn to implement a Liferay MVC portlet with all the controller code in the `-Portlet` class.

Configuring a WEB Module

As a naming convention, the module with your controller code and view layer is referred to as the WEB module. A very basic WEB module might look like this:

```
docs.liferaymvc.web/
    src/main/java/
        com/liferay/docs/liferaymvc/web/portlet/LiferayMVCPortlet.java
    src/main/resources/
        content/
            Language.properties
        META/-INF/resources/
            init.jsp
            view.jsp
    build.gradle
    bnd.bnd
```

Of course you're not tied to the use of Gradle or BndTools to build your project. However, you do need a JAR with the proper OSGi headers defined, which is easily done if you provide a `bnd.bnd` file. To see Liferay MVC portlets built with Maven and Gradle, you can check out the tutorial on Liferay Sample Projects.

Specifying OSGi Metadata

At a minimum, you should specify the bundle symbolic name and the bundle version for the OSGi runtime. Providing a human readable bundle name is also recommended.

```
Bundle-Name: Example Liferay MVC Web
Bundle-SymbolicName: com.liferay.docs.liferaymvc.web
Bundle-Version: 1.0.0
```

If you don't specify a `Bundle-SymbolicName`, one will be generated from the project's directory path, which is suitable for many cases. If you specify the bundle symbolic name yourself, it's a nice convention to use the root package name as the bundle symbolic name.

Creating a Portlet Component

Using the OSGi Declarative Services component model makes it easy to publish service implementations to the OSGi runtime. In this case an implementation of the `javax.portlet.Portlet` service must be published. Declare this using an `@Component` annotation in the portlet class:

```
@Component(
    immediate = true,
    service = Portlet.class
)
public class LiferayMVCPortlet extends MVCPortlet {
}
```

Since Liferay's `MVCPortlet` class is itself an extension of `javax.portlet.Portlet`, you've provided the right implementation. That's good in itself, but the component needs to be fleshed out with some properties:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.display-name=Liferay MVC Portlet",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class LiferayMVCPortlet extends MVCPortlet {
}
```

Some of those properties might look familiar to you if you've developed Liferay MVC portlets before 7.0. That's because they correspond with the XML attributes you used to specify in `liferay-portlet.xml`, `liferay-display.xml`, and `portlet.xml`. To find a list of all the Liferay-specific attributes you can specify as properties in your portlet components, check out the liferay-portlet-app_7_0_0.dtd. This is still maintained as a DTD to keep compatibility with the JSR-168 and JSR-286 portlet specs.

Consider the `<instanceable>` element from the above link as an example. To specify that property in your component, use this syntax in your property list:

```
"com.liferay.portlet.instanceable=true",
```

The properties namespaced with `javax.portlet....` are elements of the portlet.xml descriptor.

Also note that it is possible to create nested categories using the `com.liferay.portlet.display-category` property. The format for creating these categories is to write out the category path starting with the root and separating each category in descending order by the use of `//`. Here's an example:

```
com.liferay.portlet.display-category=root//category.category1//category.category2
```

Liferay's DTD files can be found here.

You can publish this portlet component, but it doesn't do anything yet. You'll implement the Controller code next.

Writing Controller Code

In MVC, your controller receives requests from the front end, and it receives data from the back end. It's responsible for sending that data to the right front end view so it can be displayed to the user, and it's responsible for taking data the user entered in the front end and passing it to the right back end service. For this reason, it needs a way to process requests from the front end and respond to them appropriately, and it needs a way to determine the appropriate front end view to pass data back to the user.

For data coming from the user to the back end, Liferay's MVC portlet framework offers you two ways to do this. One of these is designed for smaller applications, and the other is designed for larger applications.

First, you'll learn about processing requests in smaller applications. After that, you'll learn about how data is rendered from the back end to the user. For processing requests in larger applications, see the tutorials MVC Action Command, MVC Render Command, and MVC Resource Command. But read these after you finish this one, so you can understand how the whole framework works.

Action Methods

If you have a small application, you can implement all your controller code in your portlet class (the same one you annotated with `@Component`), which acts as your controller by itself. For processing requests, you use action methods. Here's what an action method might look like:

```
public void addGuestbook(ActionRequest request, ActionResponse response)
        throws PortalException, SystemException {

    ServiceContext serviceContext = ServiceContextFactory.getInstance(
        Guestbook.class.getName(), request);

    String name = ParamUtil.getString(request, "name");

    try {
        _guestbookService.addGuestbook(serviceContext.getUserId(),
                name, serviceContext);

        SessionMessages.add(request, "guestbookAdded");

    } catch (Exception e) {
        SessionErrors.add(request, e.getClass().getName());

        response.setRenderParameter("mvcPath",
            "/html/guestbook/edit_guestbook.jsp");
    }

}
```

In this action method, the `javax.portlet.ActionRequest` object is used to retrieve two pieces of information that are needed to call the `addGuestbook` service, which is the point of the method. If successful, the `SessionMessages` object is used to store a success message. If an exception is thrown, it's caught, and the appropriate `SessionErrors` object is used to store the exception message. Note the call to the `setRenderParameter` method on the `ActionResponse`. This is used to render the `edit_guestbook.jsp` if a guestbook could not be added, by setting the `mvcPath` parameter. This parameter is a convention in Liferay's `MVCPortlet` framework that denotes the next view that should be displayed to the user.

Render Logic

So what might a render method look like? First, note that implementing render logic might not be necessary at all. Note the `init-param` properties you set in your Component:

```
"javax.portlet.init-param.template-path=/",
"javax.portlet.init-param.view-template=/view.jsp",
```

With these, you're directing the default rendering to your `view.jsp`. The `template-path` property tells the MVC framework where your JSP files live. In the above example, / means that the JSP files are located in your project's root resources folder. That's why it's important to follow Liferay's standard folder structure, outlined above. Here's the path of a hypothetical Web module's resource folder:

```
docs.liferaymvc.web/src/main/resources/META-INF/resources
```

In this case, the `view.jsp` file is found at

```
docs.liferaymvc.web/src/main/resources/META-INF/resources/view.jsp
```

and that's the default view of the application. When the `init` method is called, the initialization parameters you specify are read and used to direct rendering to the default JSP. Throughout the controller, you can render a different view (JSP file) by setting the render parameter `mvcPath`, like this:

```
actionResponse.setRenderParameter("mvcPath", "/error.jsp");
```

In some cases, the uses of initialization parameters and render parameters obviates the need for additional render logic. However, it's often necessary to provide additional render logic. To do this, override the render method. Here's an example:

```
@Override
public void render(RenderRequest renderRequest,
        RenderResponse renderResponse) throws PortletException, IOException {

    try {
        ServiceContext serviceContext = ServiceContextFactory.getInstance(
                Guestbook.class.getName(), renderRequest);

        long groupId = serviceContext.getScopeGroupId();

        long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");

        List<Guestbook> guestbooks = _guestbookService
                .getGuestbooks(groupId);

        if (guestbooks.size() == 0) {
            Guestbook guestbook = _guestbookService.addGuestbook(
                    serviceContext.getUserId(), "Main", serviceContext);

            guestbookId = guestbook.getGuestbookId();

        }

        if (!(guestbookId > 0)) {
            guestbookId = guestbooks.get(0).getGuestbookId();
        }

        renderRequest.setAttribute("guestbookId", guestbookId);

    } catch (Exception e) {

        throw new PortletException(e);
    }

    super.render(renderRequest, renderResponse);

}
```

With render logic, you're providing the view layer with information to display the data properly to the user. In this case, there's some information needed at the outset, and then there's some logic in the `if` statements that determine if there are any Guestbooks that can be displayed. If not, a Guestbook should be created by default. If there are Guestbooks in the database, the ID that's first in the list retrieved via the `getGuestbooks` method should be displayed. This is accomplished by passing the appropriate ID to the `RenderRequest` using the `setAttirbute` method. Since this logic should be executed before the default `render` method, the method concludes by calling `super.render`.

Note: Are you wondering how to call Service Builder services in 7.0? Refer to the tutorial on Finding and Invoking Liferay Services for a more detailed explanation. In short, obtain a reference to the service by annotating a setter method with the `@Reference` Declarative Services annotation and set the service object as a private variable.

```
private GuestbookService _guestbookService;

@Reference(unbind = "-")
protected void setGuestbookService(GuestbookService guestbookService) {
    _guestbookService = guestbookService;
}
```

Once done, you can call the service's methods.

```
_guestbookService.addGuestbook(serviceContext.getUserId(), "Main",
        serviceContext);
```

Setting and Retrieving Request Parameters and Attributes

In the portlet class's render method action methods, and even in your JSPs, you can use a handy utility class called `ParamUtil` to retrieve parameters from an `ActionRequest` or a `RenderRequest`.

```
long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");
```

In the above example, the parameter was set into an action request in a JSP:

```
<portlet:param name="guestbookId"
        value="<%= String.valueOf(entry.getGuestbookId()) %>" />
```

You can also set attributes into the request using the method

```
request.setAttribute();
```

in your portlet class. To read the attribute in a JSP, use the method

```
request.getAttribute();
```

To set parameters into the response in your controller code, you can use the `setRenderParameter` method.

```
actionResponse.setRenderParameter("mvcPath", "/error.jsp");
```

Passing information back and forth from your view and controller is important, but there's more to the view layer than that.

Configuring the View Layer

You now know how to extend Liferay's `MVCPortlet` to write controller code and register a Component in the OSGi runtime. You also need a view layer, of course, and for that, you'll use JSPs. Lexicon can be used to guide your app's styling so it matches Liferay's. To learn about Lexicon and about some of Liferay's taglibs, refer to the tutorial Applying Lexicon Styles to Your App. This section will briefly cover how to get your view layer working, from organizing your imports in one JSP file, to configuring URLs that direct processing to your code in the portlet class.

It's a good practice to put all your Java imports, tag library declarations, and variable initializations into an `init.jsp` file. If you use Liferay @ide@ to create your Web module, these taglib declarations and initializations are automatically added to your `init.jsp`:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>

<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
<%@ taglib uri="http://liferay.com/tld/portlet" prefix="liferay-portlet" %>
<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>

<liferay-theme:defineObjects />

<portlet:defineObjects />
```

Make sure you include the `init.jsp` in your other JSP files:

```
<%@include file="/html/init.jsp"%>
```

You can, if necessary, write Java code in your JSPs using scriptlets. Perhaps you set an attribute into the request in your controller:

```
renderRequest.setAttribute("guestbookId", guestbookId);
```

You can reference it in your JSP by calling the `renderRequest.getAttribute` method:

```
<%
    long guestbookId = Long.valueOf((Long) renderRequest
            .getAttribute("guestbookId"));
%>
```

To construct a URL that calls the `render` method of your controller, you can use the `portlet:renderURL` tag:

```
<portlet:renderURL var="searchURL">
        <portlet:param name="mvcPath" value="/admin/view.jsp" />
</portlet:renderURL>
```

You create a variable to hold the generated URL with the var attribute. Then you can set any parameters you need using the `portlet:param` tag. The `mvcPath` parameter is used to direct to another JSP. The example above points to a JSP in

```
docs.liferaymvc.web/src/main/resources/META-INF/resources/admin/view.jsp
```

You can then use the var value to invoke the URL in your JSP code, perhaps in a button or navigation bar item.

Action URLs can be similarly created with the `portlet` taglib:

```
<portlet:actionURL name="doSomething" var="doSomethingURL">
    <portlet:param name="redirect" value="<%= redirect %>" />
</portlet:actionURL>
```

The name of the action URL should match the action method name in your portlet class; that's all Liferay's MVC framework needs in order to know that the action method of the same name should run when this action URL is invoked. Use the var attribute like you did the var attribute in the render URL; to call the action URL in your JSP code, whether it's in an icon, a button, or somewhere else.

As you can see, with Liferay MVC it's pretty easy to make your controller talk to your view layer.

Beyond the Basics

This tutorial should get you up and running with a Liferay MVC Web module, but there's more to know about creating an app in Liferay. Here are a few useful jumping off points:

- Making URLs Friendlier
- Applying Lexicon Styles to your App
- Localizing your Application
- Liferay's Workflow Framework
- Model Listeners
- Application Security
- Asset Framework
- Service Builder

60.6 MVC Action Command

Liferay's MVC framework lets you split your portlet's action methods into separate classes. This can be very helpful in portlets that have many actions. Each action URL in your portlet's JSPs then calls the appropriate action class when necessary.

First, use the `<portlet:actionURL>` tag to create the action URL in your JSP. For example, the edit blog entry action in Liferay's Blogs app is defined in the `edit_entry.jsp` file as follows:

```
<portlet:actionURL name="/blogs/edit_entry" var="editEntryURL" />
```

When the action URL is triggered, the matching action class processes the action. Implement the action by creating a class that implements the `MVCActionCommand` interface. To avoid writing oodles of boilerplate code, your `*MVCActionCommand` class should extend the `BaseMVCActionCommand` class instead of implementing `MVCActionCommand` directly. The `BaseMVCActionCommand` class already implements `MVCActionCommand` and provides many useful method implementations. Naming your `*MVCActionCommand` class after the action it performs is a good convention. For example, if your action edits some kind of entry, you could name its class `EditEntryMVCActionCommand`.

Your `*MVCActionCommand` class must also have a `@Component` annotation like the following. Set the property `javax.portlet.name` to your portlet's internal ID, and the property `mvc.command.name` to the value of the `name` property in your JSP's matching `actionURL`. To register the component in the OSGi container as using the `MVCActionCommand` class, you must set the `service` property to `MVCActionCommand.class`:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=your_portlet_name_YourPortlet",
        "mvc.command.name=/your/jsp/action/url"
    },
    service = MVCActionCommand.class
)
public class YourMVCActionCommand extends BaseMVCActionCommand {
    // implement your action
}
```

For example, this is the @Component annotation for the Blogs app's EditEntryMVCActionCommand class:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS,
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_AGGREGATOR,
        "mvc.command.name=/blogs/edit_entry"
    },
    service = MVCActionCommand.class
)
public class EditEntryMVCActionCommand extends BaseMVCActionCommand {
    // the app's edit blog entry action implementation
}
```

Note that you can use multiple javax.portlet.name values to indicate the component works with multiple portlets.

In your *MVCActionCommand class, process the action by overriding the BaseMVCActionCommand class's doProcessAction method. This method takes javax.portlet.ActionRequest and javax.portlet.ActionResponse parameters that you can use to process your action. Your *MVCActionCommand class should also contain any other code required to implement your action. For a real-world example of a *MVCActionCommand class, see the Blogs app's EditEntryMVCActionCommand class.

Related Topics

MVC Render Command

MVC Resource Command

MVC Command Overrides

60.7 MVC Render Command

If you're here, that means you know that MVCRenderCommands are used to respond to portlet render URLs, and you want to know how to create and use MVC render commands. If you just want to learn about Liferay's MVC Portlet framework in general, that information is in a separate article.

To use MVC render commands, you need these things:

- An implementation of the MVCRenderCommand interface.
- A portlet render URL in your view layer.
- a Component that publishes the MVCRenderCommand service, with two properties.

Implementing MVCRenderCommand

What is it you want to do when a portlet render URL is invoked? Using the mvcRenderCommandName parameter, direct the request to an MVCRenderCommand implementation. Now override the render method.

Some MVCRenderCommands will simply render a particular JSP. Here's what BlogsViewMVCRenderCommand looks like:

```
@Override
public String render(
    RenderRequest renderRequest, RenderResponse renderResponse) {

    return "/blogs/view.jsp";
}
```

Sometimes you'll want to add logic to render a certain JSP based on one or more conditions:

```
@Override
public String render(
    RenderRequest renderRequest, RenderResponse renderResponse)
    throws PortletException {

    try {
        ActionUtil.getEntry(renderRequest);
    }
    catch (Exception e) {
        if (e instanceof NoSuchEntryException ||
            e instanceof PrincipalException) {

            SessionErrors.add(renderRequest, e.getClass());

            return "/hello/error.jsp";
        }
        else {
            throw new PortletException(e);
        }
    }

    return "/hello/edit_entry.jsp";
}
```

If there's an error caught following the call to `ActionUtil.getEntry` in the code above, the `error.jsp` is rendered. If the call is returned without an exception being caught, `edit_entry.jsp` is rendered.

How does a request get directed to your MVC render command? Using a portlet render URL.

Creating a Portlet Render URL

You can generate a render URL for your portlet using the `<portlet:renderURL>` taglib. To invoke your MVC render command from the render URL, you need to specify the parameter `mvcRenderCommandName` with the same value as your Component property `mvc.command.name`.

For example, you can create a URL that directs the user to a page with a form for editing an entry like this (in a JSP):

```
<portlet:renderURL var="editEntryURL">
    <portlet:param name="mvcRenderCommandName" value="/hello/edit_entry" />
    <portlet:param name="entryId" value="<%= String.valueOf(entry.getEntryId()) %>" />
</portlet:renderURL>
```

Now the request will contain a parameter named `mvcRenderCommandName`. To find the proper MVC render command, the OSGi runtime needs to have a `mvc.command.name` property with a matching value.

Registering the MVC Render Command

In order to respond to a particular render URL, you need an `MVCRenderCommand` Component that with two properties:

```
 "javax.portlet.name=" + HelloWorldPortletKeys.HELLO_WORLD,
 "mvc.command.name=/hello/edit_entry"
```

Using the above properties as an example, any portlet render URL for the portlet that includes a parameter called mvcRenderCommand with the value /hello/edit_entry will be handled by this MVCRenderCommand.

The Component must also publish a MVCRenderCommand.class service to the OSGi runtime. Here's a basic Component that publishes an MVC render command.

```
@Component(
    immediate = true,
    property = {
       "javax.portlet.name=" + HelloWorldPortletKeys.HELLO_WORLD,
       "mvc.command.name=/hello/edit_entry"
    },
    service = MVCRenderCommand.class
)
public class EditEntryMVCRenderCommand implements MVCRenderCommand {
```

One command can be used by one portlet, as the example above shows. If you want, one command can be used for multiple portlets by adding more javax.portlet.name entries in the property list. Likewise, multiple commands can invoke the MVC command class by adding more mvc.command.name entries. If you're really feeling wild, you can specify multiple portlets and multiple command URLs in the same command component, like this:

```
@Component(
    immediate = true,
    property = {
       "javax.portlet.name=" + HelloWorldPortletKeys.HELLO_MY_WORLD,
       "javax.portlet.name=" + HelloWorldPortletKeys.HELLO_WORLD,
       "mvc.command.name=/hello/edit_super_entry",
       "mvc.command.name=/hello/edit_entry"
    },
    service = MVCRenderCommand.class
)
```

As you can see, MVC render commands are flexible and very easy to implement.

Related Topics

MVC Resource Command

MVC Action Command

MVC Command Overrides

60.8 MVC Resource Command

When using Liferay's MVC framework, you can create resource URLs in your JSPs to retrieve images, XML, or any other kind of resource from a Liferay instance. The resource URL then invokes the corresponding MVC resource command class (*MVCResourceCommand) that processes the resource request and response.

First, use the <portlet:resourceURL> tag to create the resource URL in a JSP. For example, the Login Portlet's /login-web/src/main/resources/META-INF/resources/navigation/create_account.jsp file defines the following resource URL for retrieving a CAPTCHA image during account creation:

```
<portlet:resourceURL id="/login/captcha" var="captchaURL" />
```

When the resource URL is triggered, the matching *MVCResourceCommand class processes the resource request and response. You can create this class by implementing the MVCResourceCommand interface, or extending the BaseMVCResourceCommand class. The latter may save you time, since it already implements MVCResourceCommand.

Also, it's a good idea to name your *MVCResourceCommand class after the resource it handles, and suffix it with MVCResourceCommand. For example, the resource command class matching the preceding CAPTCHA resource URL in the Login Portlet is CaptchaMVCResourceCommand. In an application with several MVC command classes, this will help differentiate them.

Your *MVCResourceCommand class must also have a @Component annotation like the following. Set the property javax.portlet.name to your portlet's internal ID, and the property mvc.command.name to the value of the id property in your JSP's matching resourceURL. To register the component in the OSGi container as using the MVCResourceCommand class, you must set the service property to MVCResourceCommand.class:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=your_portlet_name_YourPortlet",
        "mvc.command.name=/your/jsp/resource/url"
    },
    service = MVCResourceCommand.class
)
public class YourMVCResourceCommand implements MVCResourceCommand {
    // your resource handling code
}
```

As a real-world example, consider the Login Portlet's CaptchaMVCResourceCommand class (find this class in the Liferay source code at modules/apps/foundation/login/login-web/src/main/java/com/liferay/login/web/internal/portle

```
@Component(
    property = {
        "javax.portlet.name=" + LoginPortletKeys.FAST_LOGIN,
        "javax.portlet.name=" + LoginPortletKeys.LOGIN,
        "mvc.command.name=/login/captcha"
    },
    service = MVCResourceCommand.class
)
public class CaptchaMVCResourceCommand implements MVCResourceCommand {

    @Override
    public boolean serveResource(
        ResourceRequest resourceRequest, ResourceResponse resourceResponse) {

        try {
            CaptchaUtil.serveImage(resourceRequest, resourceResponse);

            return false;
        }
        catch (Exception e) {
            _log.error(e, e);

            return true;
        }
    }

    private static final Log _log = LogFactoryUtil.getLog(
        CaptchaMVCResourceCommand.class);
}
```

In the @Component annotation, note that javax.portlet.name has two different settings. This lets multiple portlets use the same component. In this example, the portlet IDs are defined as constants in the LoginPortletKeys class. Also note that the mvc.command.name property setting /login/captcha matches the resourceURL's id setting shown earlier in this tutorial, and that the service property is set to MVCResourceCommand.class.

The CaptchaMVCResourceCommand class implements the MVCResourceCommand interface with only a single method: serveResource. This method processes the resource request and response via the javax.portlet.ResourceRequest and javax.portlet.ResourceResponse parameters, respectively. Note that the try block uses the helper class CaptchaUtil to serve the CAPTCHA image. Though you don't have to create such a helper class, doing so often simplifies your code.

Great! Now you know how to use MVCResourceCommand to process resources in your Liferay MVC portlets.

Related Topics

MVC Render Command

MVC Action Command

MVC Command Overrides

Chapter 61

Liferay Soy Portlet

A Soy portlet is an extension of Liferay's MVC portlet framework. This gives you access to all the MVC Portlet functionality you are familiar with, plus the added bonus of using Soy templates for writing your front-end. Soy templates use an easy templating language that also lets you use MetalJS components. With all these benefits and more, Soy portlets can be a good front-end tool to have in your utility belt.

You can learn about Liferay MVC portlets in the Creating an MVC Portlet tutorial.

This section covers how to implement a Soy portlet.

61.1 Creating a Soy Portlet

To create a Soy portlet, you'll need these key components:

- A module that publishes a portlet component with the necessary properties
- Controller code to handle the request and response
- Soy templates to implement your view layer

Configuring the Web Module

First, familiarize yourself with a Soy portlet's anatomy. You may recognize it, since a Soy portlet extends an MVC portlet:

- `my-soy-portlet`
    - `bnd.bnd`
    - `build.gradle`
    - `package.json`
    - `src/main/`
        - `java/path/to/portlet/`
            - `MySoyPortlet.java`
            - `action/`
            - `*MVCRenderCommand.java`

* resources/META-INF/resources/

  - content/
  - Language.properties

  - View.es.js (MetalJS component)
  - View.soy (Soy template)

Now that you know the basic structure of a Soy portlet module, you can configure it. You can use the soy portlet Blade template to build your initial project if you wish. Otherwise, you can follow the instructions in this section to manually configure the module.

Specifying OSGi Metadata

Add the OSGi metadata to your module's `bnd.bnd` file. A sample BND configuration is shown below:

```
Bundle-Name: Liferay Hello Soy Web
Bundle-SymbolicName: com.liferay.hello.soy.web
Bundle-Version: 1.0.3
Require-Capability:\
    soy;\
        filter:="(type=metal)"
Include-Resource: package.json
```

In addition to the standard metadata, notice the `Require-Capability` property. This specifies that this bundle requires modules that provide the capability `soy` with a `type` of `metal` to work. Also note the `Include-Resource` property. **You must include your `package.json` file to load the Soy Portlet's JavaScript files.**

Specifying JavaScript Dependencies

Specify the JavaScript module dependencies in your `package.json`. At a minimum, you should have the following dependencies and configuration parameters:

```
{
    "dependencies": {
        "metal-component": "^2.4.5",
        "metal-soy": "^2.4.5"
    },
    "devDependencies": {
        "liferay-module-config-generator": "^1.2.1",
        "metal-cli": "^4.0.1"
    },
    "name": "my-portlet-name",
    "version": "1.0.0"
}
```

This provides everything you need to create a Metal component based on Soy. Note that the `version` in your `package.json` should match the `Bundle-Version` in your `bnd.bnd` file.

Next you can specify your module's build dependencies.

Specifying Build Dependencies

Add the dependencies shown below to your `build.gradle` file:

```
dependencies {
    provided group: "com.liferay", name: "com.liferay.portal.portlet.bridge.soy", version: "3.1.0"
    provided group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    provided group: "com.liferay.portal", name: "com.liferay.util.java", version: "2.0.0"
    provided group: "javax.portlet", name: "portlet-api", version: "2.0"
    provided group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    provided group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

Note: These are current at the time of this writing, but may change. Please check the Nexus Repository for the proper versions for your Liferay DXP instance.

Now that your module build is configured, you can learn how to create the Soy portlet component.

Creating a Soy Portlet Component

Create a Soy Portlet component that extends the `SoyPortlet` class. This requires an implementation of the `javax.portlet.portlet` service to run. Declare this using an `@Component` annotation in the portlet class:

```
@Component(
    immediate = true,
    service = Portlet.class
)
public class MySoyPortlet extends SoyPortlet {
    @Override
    public void render(RenderRequest renderRequest, RenderResponse renderResponse) {
        //do things here
    }
}
```

Liferay DXP's `SoyPortlet` class extends Liferay DXP's `MVCPortlet` class, which is an extension itself of `javax.portlet.Portlet`, so you've provided the right implementation.

The component requires some properties as well. A sample configuration is shown below:

```
@Component(
        immediate = true,
        property = {
          "com.liferay.portlet.add-default-resource=true",
          "com.liferay.portlet.application-type=full-page-application",
          "com.liferay.portlet.application-type=widget",
          "com.liferay.portlet.display-category=category.sample",
          "com.liferay.portlet.layout-cacheable=true",
          "com.liferay.portlet.preferences-owned-by-group=true",
          "com.liferay.portlet.private-request-attributes=false",
          "com.liferay.portlet.private-session-attributes=false",
          "com.liferay.portlet.render-weight=50",
          "com.liferay.portlet.scopeable=true",
          "com.liferay.portlet.use-default-template=true",
          "javax.portlet.display-name=Hello Soy Portlet",
          "javax.portlet.expiration-cache=0",
          "javax.portlet.init-param.copy-request-parameters=true",
          "javax.portlet.init-param.template-path=/",
          "javax.portlet.init-param.view-template=View",
          "javax.portlet.name=hello_soy_portlet",
          "javax.portlet.resource-bundle=content.Language",
            "javax.portlet.security-role-ref=guest,power-user,user",
            "javax.portlet.supports.mime-type=text/html"
        },
        service = Portlet.class
)
```

Some of these properties may seem familiar to you, as they are the same ones used to develop an MVC portlet. You can find a list of all the Liferay-specific attributes that are available for use as properties in your portlet components in the `liferay-portlet-app_7_0_0.dtd`.

The `javax.portlet...` properties are elements of the portlet.xml descriptor

Liferay's DTD files can be found here

Now that you've set your Soy portlet component's foundation, you can write the controller code.

Writing Controller Code

Soy portlets extend MVC portlets, so they use the same Model-View-Controller framework to operate. Your controller receives requests from the front-end, and it receives data from the back-end. It's responsible for sending that data to the right front-end view so it can be displayed to the user, and it's responsible for taking data the user entered in the front-end and passing it to the right back-end service. For this reason, it needs a way to process requests from the front-end and respond to them appropriately, and it needs a way to determine the appropriate front-end view to pass data back to the user.

Render Logic

The render logic is where all the magic happens. After all, what's the use of a portlet if you can't see it? Note the `init-param` properties you set in your Component class:

```
"javax.portlet.init-param.template-path=/",
"javax.portlet.init-param.view-template=View",
```

This directs the default rendering to View (`View.soy`). The `template-path` property tells the framework the location of your Soy templates. The / above means that the Soy files are located in your project's root `resources` folder. That's why it's important to follow Liferay DXP's standard folder structure, outlined above. Here's the path of a hypothetical web module's resource folder:

```
docs.liferaysoy.web/src/main/resources/META-INF/resources
```

In this case, the `View.soy` file is found at:

```
docs.liferaysoy.web/src/main/resources/META-INF/resources/View.soy
```

That's the default view of the application. When the `init` method is called, the initialization parameters you specify are read and used to direct rendering to the default template. Throughout this framework, you can render a different view (Soy template) by setting the `mvcRenderCommandName` parameter of the `javax.portlet.PortletURL` to the Soy template. The example below uses a portlet URL called `navigationURL` to render the view `View`:

```
navigationURL.setParameter("mvcRenderCommandName", "View");
```

Each view, excluding the default template view, **must have** an implementation of the `MVCRenderCommand` class. The `*MVCRenderCommand` implementation must declare itself as a component with the `MVCRenderCommand` service, and it must specify the portlet's name and MVC command name using the `javax.portlet.name` and `mvc.command.name` properties respectively. Below is an example `MVCRenderCommand` implementation for a `Navigation` Soy template:

```
@Component(
        immediate = true,
        property = {
                "javax.portlet.name=hello_soy_portlet",
                "mvc.command.name=Navigation"
        },
        service = MVCRenderCommand.class
)
public class HelloSoyNavigationExampleMVCRenderCommand
        implements MVCRenderCommand {

        @Override
        public String render(
                RenderRequest renderRequest, RenderResponse renderResponse) {

                Template template = (Template)renderRequest.getAttribute(
                        WebKeys.TEMPLATE);

                PortletURL navigationURL = renderResponse.createRenderURL();

                navigationURL.setParameter("mvcRenderCommandName", "View");

                template.put("navigationURL", navigationURL.toString());

                return "Navigation";
        }

}
```

The render logic provides the view layer with information to display the data properly to the user. In this case the MVC command name is set to `Navigation` (the Soy template with namespace `Navigation`). The MVC render command name for the `PortletURL` `navigationURL` is set to `View` (the Soy template with namespace `Navigation`), using the `mvcRenderCommandName` attribute. The `navigationURL` parameter is passed to the `Navigation` Soy template as the variable `navigationURL`, using the `template.put()` method. Finally, the `*MVCRenderCommand` class returns the MVC render command name as a `String`.

Note that Soy portlet parameters are scoped to the portlet class they're written in. For instance, you can have a `navigationURL` parameter in two different classes, each with a different value. Below is an example `HelloSoyPortlet` class that also defines a `navigationURL` parameter:

```
public class HelloSoyPortlet extends SoyPortlet {

    @Override
    public void render(
            RenderRequest renderRequest, RenderResponse renderResponse)
        throws IOException, PortletException {

        PortletURL navigationURL = renderResponse.createRenderURL();

        navigationURL.setParameter("mvcRenderCommandName", "Navigation");

        template.put("navigationURL", navigationURL.toString());

        template.put("releaseInfo", ReleaseInfo.getReleaseInfo());

        super.render(renderRequest, renderResponse);
    }

}
```

The `navigationURL` points to the `Navigation` Soy template this time. The `navigationURL` and `releaseInfo` parameters are passed to the `View` Soy template. Since this logic should be executed before the default `render` method, the method concludes by calling `super.render`.

Now that you understand the render logic, you can learn how the view layer works.

Configuring the View Layer

Your portlet also requires a view layer, and for that you'll use Soy templates, which is the whole point of developing a Soy portlet, isn't it? This section briefly covers how to get your view layer working, from including other Soy templates, to creating a MetalJS component for rendering your views.

Soy templates are defined in a file with the extension `.soy`. The filename is arbitrary. The Soy template's name is specified at the top of the template using the `namespace` declaration. For example, this template has the namespace `View`:

```
{namespace View}
```

It can be accessed in another Soy template by calling the `render` method on the namespace as shown below:

```
{call View.render data="all"}{/call}
```

Below is an example `View` Soy template that includes `Header` and `Footer` Soy templates:

```
{namespace View}

/**
 * Prints the portlet main view.
 */
{template .render}
  <div id="{$id}">
    {call Header.render data="all"}{/call}

    <p>{msg desc=""}here-is-a-message{/msg}</p>

    {call Footer.render data="all"}{/call}
  </div>
{/template}
```

Each view has a corresponding `*es.js` file (usually with the same name) that imports the Soy templates the view requires and registers the view as a MetalJS component. This file is also used for any additional JavaScript logic your view may have. For example, here is a `View.es.js` component for a `View.soy` template:

```
import Component from 'metal-component/src/Component';
import Footer from './Footer.es';
import Header from './Header.es';
import Soy from 'metal-soy/src/Soy';
import templates from './View.soy';

/**
 * View Component
 */
class View extends Component {}

// Register component
Soy.register(View, templates);

export default View;
```

Now that you understand how to configure a Soy template view, you can learn how to use portlet parameters in your Soy templates next.

Using Portlet Template Parameters in the Soy Template

As mentioned above, the `template.put()` method exposes portlet parameters to the Soy templates. Once a parameter is exposed, you can access it in the Soy template by defining it at the top with the `{@param name: type}` declaration. For instance, the `hello-soy-web` portlet's `View` Soy template defines the `navigationURL` parameter with the code below:

```
{@param navigationURL: string}
```

It is then used to navigate between portlet views:

```
<a href="{$navigationURL}">{msg desc=""}
    click-here-to-navigate-to-another-view
{/msg}</a>
```

Some Java theme object variables are available as well. For example, to access the `ThemeDisplay` object in a Soy template, use the following syntax:

```
{$themeDisplay}
```

You can also access the `Locale` object by using `{$locale}`. Here is the full `View.soy` template for the `com.liferay.hello.soy.web` portlet, which demonstrates the features covered in this section:

```
{namespace View}

/**
 * Prints the portlet main view.
 */
{template .render}
    {@param id: string}
    {@param layouts: list<[
        friendlyURL: string,
        nameCurrentValue: string
    ]>}
    {@param navigationURL: string}

    <div id="{$id}">
        {call Header.render data="all"}{/call}

        <p>{msg desc=""}here-you-will-find-how-easy-it-is-to-do-things-like{/msg}</p>

        <h3>{msg desc=""}listing-pages{/msg}</h3>

        <div class="list-group">
            <div class="list-group-heading">{msg desc=""}navigate-to{/msg}</div>

            {foreach $layout in $layouts}
                <a class="list-group-item" href="{$layout.friendlyURL}">{$layout.nameCurrentValue}</a>
            {/foreach}
        </div>

        <h3>{msg desc=""}navigating-between-views{/msg}</h3>

        <a href="{$navigationURL}">{msg desc=""}click-here-to-navigate-to-another-view{/msg}</a>

        {call Footer.render data="all"}{/call}
    </div>
{/template}
```

Now you know how to create a Soy Portlet!

Related Topics

Liferay MVC Portlet

JSF Portlets with Liferay Faces

CHAPTER 62

THE STATE OBJECT

MetalJS's `component` class, which your view component extends, extends MetalJS's `state` class. The `state` class provides a `STATE` object that contains state properties, as well as watches these properties for changes. Any template parameters defined in your portlet classes are automatically added as properties to the portlet's `STATE` object. The `component` class provides additional rendering logic, such as automatically re-rendering the component when the `state` class detects a change in a state property. This means that you can change a state property on the client-side and automatically see that change reflected in the component's UI!

This section of tutorials covers how to configure and use your Soy portlet's `STATE` object.

62.1 Understanding The State Object's Architecture

An example `STATE` object configuration appears below:

```
View.STATE {
  myStateProperty: {
    setter: 'setterFunction',
    validator: val => val === expected value,
    value: default value,
    valueFn: val => default value,
    writeOnce: true
  }
}
```

State properties have these configuration options:

setter: Normalizes the state key's value. The setter function receives the new value that was set and return the value that should be stored.

validator: Validates the state key's value. When it returns `false`, the new value is ignored.

value: The state key's default value. Alternatively, you can set the default value with the `valueFn` property. Setting this to an object causes all class instances to use the same reference to the object. To have each instance use a different reference for objects, use the `valueFn` option instead. Note that the portlet template parameter's value (if applicable) has priority over this value.

valueFn: A function that returns the state key's default value. Alternatively, you can set the default value with the `value` property. Note that the portlet template parameter's value (if applicable) has priority over this value.

writeOnce: Whether the state key is read-only, meaning the initial value is the final value.

Now you know the `STATE` object's architecture and how to configure it!

Related Topics

Configuring Portlet Template Parameter State Properties

Configuring Soy Portlet Template Parameters on the Client Side

62.2 Configuring Portlet Template Parameter State Properties

Portlet template parameters are added automatically as state properties to the view component's `STATE` object. Therefore, you can provide additional configuration options for them in the `STATE` object. The example below sets the default value for the portlet template parameter `color` in its `*MVCRenderCommand` class:

```
Template template = (Template)renderRequest.getAttribute(
            WebKeys.TEMPLATE);

String color = "red";

template.put("color", color);
```

The configuration above has the implicit state property configuration shown below in the view's component file (`View.es.js` for example):

```
   View.STATE {
     color: {
       value: 'red'
     }
   }
```

You can provide additional settings by configuring the state Property in the `View` component. The example below defines a `setter` function that transforms the color's string to upper case before adding it to the `STATE` object:

```
function setColor(color) {
  return color.toUpperCase();
}

View.STATE = {
  color: {
    setter: 'setColor'
  }
}
```

Now you know how to configure portlet template parameter state properties!

Related Topics

Understanding the State Object's Architecture

Configuring Soy Portlet Template Parameters on the Client Side

62.3 Configuring Soy Portlet Template Parameters on the Client Side

Portlet template parameters are set in the Soy Portlet's server-side code. MetalJS's `state` class provides a `STATE` object that exposes these parameters as properties so you can access them on the client side. This tutorial covers how to configure your view component's `STATE` object and its properties on the client side so you can update the UI.

This tutorial references the example below.

An Example Header State Portlet

This tutorial references the example portlet covered in this section. It includes one view with a header that reads *Hello Soy* by default.

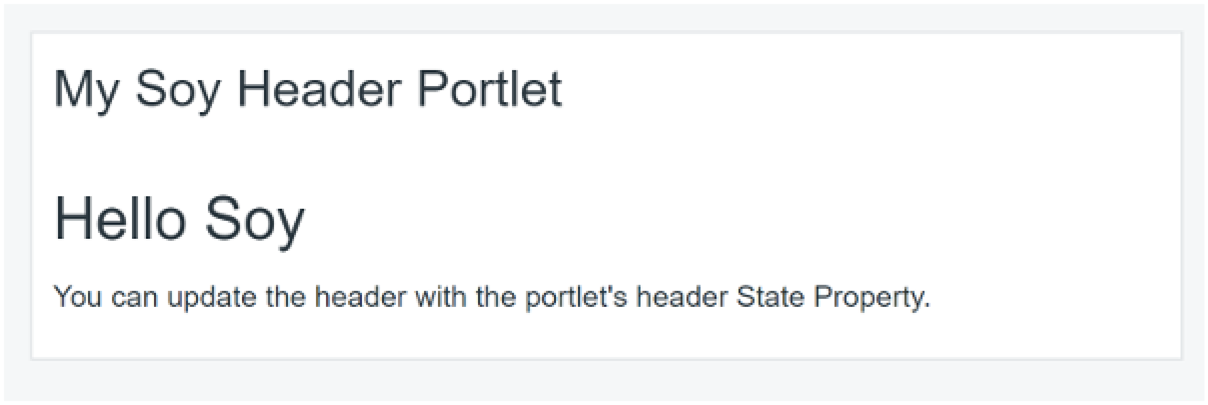

My Soy Header Portlet
Hello Soy
You can update the header with the portlet's header State Property.

Figure 62.1: The example Soy portlet has a configurable header.

The text in the header following *Hello* is provided by the `header` state property defined in its `*mvcRenderCommand` class. Deploy the provided `com.liferay.docs.state.soy-1.0.0.jar` file to follow along with this tutorial. The example portlet's *MVCRenderCommand class and Soy template appear below for reference:

`*MVCRenderCommand` class:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=MyStateSoyPortlet", "mvc.command.name=View",
        "mvc.command.name=/"
    },
    service = MVCRenderCommand.class
)
public class MyStateSoyPortletViewMVCRenderCommand
    implements MVCRenderCommand {

    @Override
    public String render(
        RenderRequest renderRequest, RenderResponse renderResponse) {

        Template template = (Template)renderRequest.getAttribute(
            WebKeys.TEMPLATE);

        String header = "Soy";

        template.put("header", header);

        return "View";
    }

}
```

`View.soy`:

```
{namespace View}

/**
 * Prints the portlet main view.
 */
```

```
{template .render}
    {@param id: string}
    {@param header: string}

    <div id="{$id}">

        <h1>Hello {$header}</h1>

        <p>You can update the header with the portlet's header State properties.</p>
    </div>
{/template}
```

Configuring the State properties

Soy Portlets are registered automatically using the `Liferay.component` API, so you can use this API to retrieve your portlet and update its state properties. You can test this in your browser's developer console.

Follow these steps:

1. Open the console in your web browser.

2. Retrieve your portlet's component by passing the Soy portlet's ID in the method `Liferay.component()`. For example, you can access the example portlet's component with the code below:

   ```
   Liferay.component('_MyStateSoyPortlet_');
   ```

   This returns the Soy portlet's component Object containing the state properties along with properties inherited from the prototype. Alternatively, you can access the `STATE` object directly by calling the `getState()` method:

   ```
   Liferay.component("_MyStateSoyPortlet_").getState();
   ```

```
**Note:** The `Liferay.component()` method only returns the `STATE` object
information for components currently on the page. These are the state
properties defined for the current view.
```

---

3. Now that you retrieved your Soy portlet's component, you can access its state properties the same way you would access any object's properties: the dot notation or the bracket notation. For example, you can use the code below to retrieve the `com.liferay.docs.state.soy` portlet's header state property:

   ```
   Liferay.component("_MyStateSoyPortlet_").header;
   ```

   or

   ```
   Liferay.component("_MyStateSoyPortlet_")["header"]
   ```

4. Update the state property's value:

   ```
   Liferay.component("portletID").stateProperty = "new value";
   ```

or

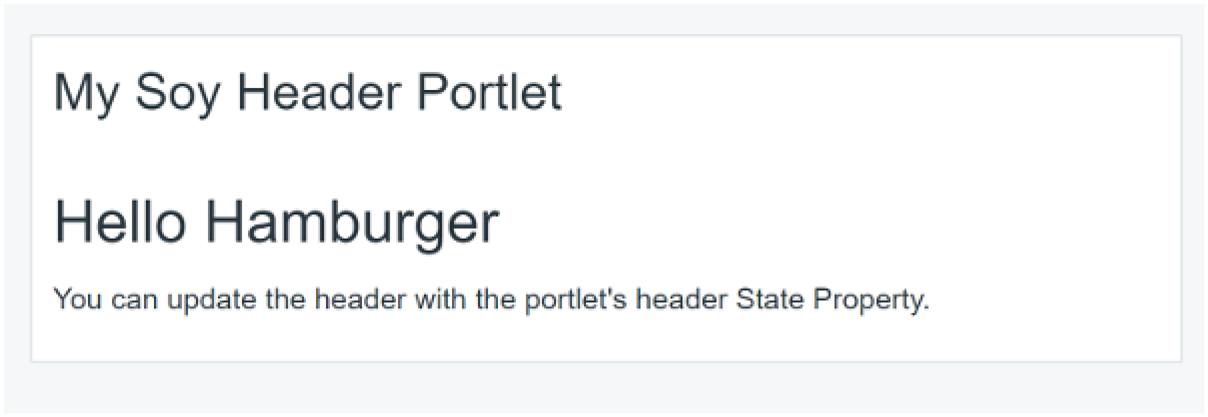
```
Liferay.component("portletID")["stateProperty"] = "new value";
```

or you can pass a configuration object with the `setState()` method:

```
Liferay.component("portletID").setState({stateProperty: new value});
```

For example, you can change the example portlet's header to read *Hello Hamburger* instead, if you don't like soy:

```
Liferay.component('_MyStateSoyPortlet_').setState({header: 'Hamburger'});
```


My Soy Header Portlet

Hello Hamburger

You can update the header with the portlet's header State Property.

Figure 62.2: You can change the example portlet's header state property on the client side.

Now you know how to configure Soy portlet state properties on the client side!

Related Topics

Understanding the State Object's Architecture
Configuring Portlet Template Parameter State Properties

Chapter 63

Spring MVC

Liferay is an open platform in an ecosystem of open platforms. Just because Liferay has its own MVC framework, therefore, doesn't mean you have to use it. It is perfectly valid to bring the tools and experience you have from other development projects over to Liferay. In fact, we expect you to. Liferay's development platform is standards-based, making it an ideal choice for applications of almost any type.

If you're already a wizard with Spring MVC, you can use it instead of Liferay's `MVCPortlet` class with no limitations whatsoever. Since Spring MVC replaces only your application's web application layer, you can still use Service Builder for your service layer.

So what does it take to implement a Spring MVC application in Liferay? Start by considering how to package a Spring MVC application for 7.0.

63.1 Packaging a Spring MVC Portlet

Developers creating portlets for 7.0 can usually deploy their portlet as Java EE-style Web Application ARchive (WAR) artifacts or as Java ARchive (JAR) OSGi bundle artifacts. Spring MVC portlet developers don't have that flexibility. Spring MVC portlets must be packaged as WAR artifacts because the Spring MVC framework is designed for Java EE. Therefore, it expects a WAR layout and requires Java EE resources such as the `WEB-INF/web.xml` descriptor.

Because Liferay supports the OSGi WAB (Web Application Bundler) standard for deployment, you can deploy your WAR and it runs as expected in the OSGi runtime. Here are the high points on why that works in 7.0:

- The Liferay auto-deploy process runs, adding the `PortletServlet` and `PlugincontextListener` configurations to the `WEB-INF/web.xml` file.
- The Liferay WAB Generator automatically creates an OSGi-ready `META-INF/MANIFEST.MF` file. If you want to affect the content of the manifest file, you can place BND directives and OSGi headers directly into the `WEB-INF/liferay-plugin-package.properties` file.

You'll still need to provide the Liferay descriptor files `liferay-display.xml` and `liferay-portlet.xml`, and you'll need a `portlet.xml` descriptor.

Develop a Spring MVC portlet WAR file with the appropriate descriptor files.

Import class packages your portlet's descriptor files reference by adding the packages to an `Import-Package` header in your `liferay-plugin-package.properties` file.

Here's an example Import-Package header:

```
Import-Package:\
    org.springframework.beans.factory.xml,\
    org.springframework.context.config,\
    org.springframework.security.config,\
    org.springframework.web.servlet.config
```

The auto-deploy process and Liferay's WAB generator convert your project to a Liferay-ready WAB. The WAB generator detects your class's import statements and adds all external packages to the WAB's Import-Package header. The generator merges packages from your plugin's liferay-plugin-package.properties into the header also.

If you depend on a package from Java's rt.jar other than a java.* package, override portal property org.osgi.framework.bootdelegation and add it to the property's list. Go here for details.

Note: Spring MVC portlets whose embedded JARs contain properties files (e.g., spring.handlers, spring.schemas, spring.tooling) might be affected by issue LPS-75212. The last JAR that has properties files is the only JAR whose properties are added to the resulting WAB's classpath. Properties in other JARs aren't added.

For example, suppose that a portlet has several JARs containing these properties files:

- WEB-INF/src/META-INF/spring.handlers
- WEB-INF/src/META-INF/spring.schemas
- WEB-INF/src/META-INF/spring.tooling

The properties from the last JAR processed are the only ones added to the classpath. The properties files must be on the classpath in order for the module to use them.

To add all the properties files to the classpath, you can combine them into one of each type (e.g., one spring.handlers, one spring.schemas, and one spring.tooling) and add them to WEB-INF/src.

Here's a shell script that combines these files:

```
cat /dev/null > docroot/WEB-INF/src/META-INF/spring.handlers
cat /dev/null > docroot/WEB-INF/src/META-INF/spring.schemas
cat /dev/null > docroot/WEB-INF/src/META-INF/spring.tooling
for jar in $(find docroot/WEB-INF/lib/ -name '*.jar'); do
for file in $(unzip -l $jar | grep -F META-INF/spring. | awk '
{ print $4 }
'); do
if [ "META-INF/spring.tld" != "$file" ]; then
unzip -p $jar $file >> docroot/WEB-INF/src/$file
echo >> docroot/WEB-INF/src/$file
fi
done
done
```

You can modify and use the shell script to add your JAR's properties files to the classpath.

Note: If you want to use a Spring Framework version different from the version Liferay DXP provides, you must name your Spring Framework JARs differently from the ones portal property module.framework.web.generator.excluded.paths excludes. If you don't rename your Spring Framework JARs, the WAB generator assumes you're using Liferay DXP's Spring Framework JARs and excludes yours from the generated WAB. Understanding Excluded JARs explains how to detect Liferay DXP's Spring Framework version.

Now get into the details of configuring a Spring MVC portlet for Liferay.

63.2 Spring MVC Portlets in Liferay

This isn't a comprehensive guide to configuring a Spring MVC portlet. It covers the high points, assuming you already have familiarity with Spring MVC. If you don't, you should consider using Liferay's MVC framework.

What does a Liferay Spring MVC portlet look like? Almost identical to any other Spring MVC portlet. To configure a Spring MVC portlet, start with the `<portlet-class>` element in `portlet.xml`. In it you must declare Spring's `DispatcherPortlet`:

```
<portlet-class>org.springframework.web.portlet.DispatcherPortlet</portlet-class>
```

The Spring front controller needs to know where the application context file is, so specify it as an initialization parameter in `portlet.xml` (update the path as needed):

```
<init-param>
    <name>contextConfigLocation</name>
    <value>/WEB-INF/spring/portlet-context.xml</value>
</init-param>
```

Provide an application context file (`portlet-context.xml` in the example above), specified as you normally would for your Spring MVC portlet.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

If you're configuring a WAB yourself, the `web.xml` file in your Spring MVC project needs to be fully ready for deployment. In addition to any `web.xml` configuration for Spring MVC, you need to include a listener for `PluginContextListener` and a `servlet` and `servlet-mapping` for `PortletServlet`:

```
<listener>
    <listener-class>com.liferay.portal.kernel.servlet.PluginContextListener</listener-class>
</listener>
<servlet>
    <servlet-name>Portlet Servlet</servlet-name>
    <servlet-class>com.liferay.portal.kernel.servlet.PortletServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Portlet Servlet</servlet-name>
    <url-pattern>/portlet-servlet/*</url-pattern>
</servlet-mapping>
```

If you're letting Liferay generate the WAB for you (this is the recommended approach), the above is not necessary, as it is added automatically during auto-deployment.

Your application must be able to convert `javax.portlet.PortletRequests` to `javax.servlet.ServletRequests` and back again. Add this to `web.xml`:

```
<servlet>
    <servlet-name>ViewRendererServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.ViewRendererServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>ViewRendererServlet</servlet-name>
    <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

To configure the Spring view resolver, add a bean in your application context file:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/views/" />
    <property name="suffix" value=".jsp" />
</bean>
```

Now the front controller, `org.springframework.web.portlet.DispatcherPortlet`, can get a request from the view layer, but there are no actual controller classes to delegate the request handling to.

With Spring MVC, your controller is conveniently separated into classes that handle the portlet modes (View, Edit, Help).

You'll use Spring's annotations to configure the controller and tell `DispatcherPortlet` which mode the controller supports.

A simple controller class supporting View mode might look like this:

```
@Controller("myAppController")
@RequestMapping("VIEW")
public class MyAppController {

    @RenderMapping
    public String processRenderRequest(RenderRequest request,
            RenderResponse response) {

        return "defaultView";
    }
}
```

The return `defaultView` statement should be understood in terms of the view resolver bean in the application context file, which gives the String `defaultView` a prefix of `WEB-INF/views/`, and a suffix of `.jsp`. That maps to the path `WEB-INF/views/defaultView.jsp`, where you would place your default view for the application.

With Spring MVC, you can only support one portlet phase in each controller.

An edit mode controller might contain render methods and action methods.

```
@Controller("myAppEditController")
@RequestMapping("EDIT")
public class MyAppEditController {

    @RenderMapping
    public String processRenderRequest(RenderRequest request,
            RenderResponse response) {

        return "thisView";
    }

    @ActionMapping(params="action=doSomething")
    public void doSomething(Actionrequest request, ActionResponse response){

        // Do something here

    }
}
```

You need to define any controller classes in your application context file by adding a <bean> tag for each one:

```
<bean class="com.liferay.docs.springmvc.portlet.MyAppController" />
<bean class="com.liferay.docs.springmvc.portlet.MyAppEditController" />
```

Develop your controllers and your views as you normally would in a Spring MVC portlet. You'll also need to provide some necessary descriptors for Liferay.

Liferay Descriptors

Liferay portlet plugins that are packaged as WAR files should include some Liferay specific descriptors.

The descriptor `liferay-display.xml` controls the category in which your portlet appears in Liferay DXP's *Add* menu. Find the complete DTD here.

Here's a simple example that just specifies the category the application will go under in Liferay's menu for adding applications:

```
<display>
    <category name="New Category">
        <portlet id="example-portlet" />
    </category>
</display>
```

The descriptor `liferay-portlet.xml` is used for specifying additional information about the portlet (like the location of CSS and JavaScript files or the portlet's icon. A complete list of the attributes you can set can be found here

```
<liferay-portlet-app>
    <portlet>
        <portlet-name>example-portlet</portlet-name>
        <instanceable>true</instanceable>
        <render-weight>0</render-weight>
        <ajaxable>true</ajaxable>
        <header-portlet-css>/css/main.css</header-portlet-css>
        <footer-portlet-javascript>/js/main.js</footer-portlet-javascript>
        <footer-portlet-javascript>/js/jquery.foundation.orbit.js</footer-portlet-javascript>
    </portlet>
    <role-mapper>
        <role-name>administrator</role-name>
        <role-link>Administrator</role-link>
    </role-mapper>
    <role-mapper>
        <role-name>guest</role-name>
        <role-link>Guest</role-link>
    </role-mapper>
    <role-mapper>
        <role-name>power-user</role-name>
        <role-link>Power User</role-link>
    </role-mapper>
    <role-mapper>
        <role-name>user</role-name>
        <role-link>User</role-link>
    </role-mapper>
</liferay-portlet-app>
```

Important: Make your portlet name unique, considering how Liferay DXP uses the name to create the portlet's ID.

You'll also notice the `role-mapper` elements included above. They're for defining the Liferay roles used in the portlet.

Then there's the `liferay-plugin-package.properties`. These properties describe the Liferay plugin, declare its resources, and specify its security related parameters. The DTD is found here.

```
name=example-portlet
module-group-id=liferay
module-incremental-version=1
tags=
short-description=
change-log=
page-url=http://www.liferay.com
author=Liferay, Inc.
licenses=LGPL
version=1
```

In the `liferay-plugin-package.properties` file, you can also add OSGi metadata which the Liferay WAB Generator adds to the `MANIFEST.MF` file when you deploy your WAR file.

Find all of Liferay's DTDs here.

63.3 Calling Services from Spring MVC

To call OSGi-based Service Builder services from your Spring MVC portlet, you need a mechanism that gives you access to the OSGi service registry.

Note: If you don't already have one, create a service builder project using Blade CLI.

```
 springmvc-service-builder/
     build.gradle
     springmvc-service-builder-api/
         bnd.bnd
         build.gradle
     springmvc-service-builder-service/
         bnd.bnd
         build.gradle
         service.xml
```

Design your model entity and write your service layer as normal (see the tutorials on Service Builder here). After that, add your service's API JAR as a dependency in your Spring MVC project.

Since you're in the context of a Spring MVC portlet, you can't look up a reference to the services published to the OSGi runtime using Declarative Services. So how do you call Service Builder services, or other services published in the OSGi service registry? One way is by calling the static utility methods.

```
FooServiceUtil.getFoosByGroupId()
```

While very simple, that's not a good way to call OSGi services because of the dynamic nature of the OSGi runtime. Service implementations could be removed and added at any time, and your plugin needs to be able to account for that. Additionally, you need a mechanism that lets your portlet plugin react gracefully to the possibility of the service implementation becoming unavailable entirely. That's why you should open a Service Tracker when you want to call a service that's in the OSGi service registry.

Service Trackers

Since you don't have the luxury of using Declarative Services to manage your service dependencies, you have a little bit of work to do if you want to gain some of the benefits OSGi gives you:

- Accounting for multiple service implementations, using the best service implementation available (taking into account the service ranking property)
- Accounting for no service implementations

The static utility classes don't let you do that, and that's sad. But be happy, because with a little code, you can regain those benefits. For the details on implementing a service tracker, read the Service Trackers tutorial.

To summarize, you'll need to do these things:

- Instantiate a new `org.osgi.util.tracker.ServiceTracker` to track the service of the type you need.
- Open the service tracker in an `@PostConstruct` method.
- Make sure the service tracker has something in it.
- If there is indeed something in the service tracker, get the service.
- Now you're ready to call the service. Here's what the `if` block might look like:

```
if (!someServiceTracker.isEmpty()) {
    SomeService someService = someServiceTracker.getService();
    someService.doSomethingCool();
}
```

- Close the service tracker in an `@PreDestroy` method.

That's probably not enough detail, so refer to the tutorial on Service Trackers for the details. As you'll see in the tutorial, there's some boilerplate code involved, but leveraging service trackers lets you look up services in the OSGi runtime.

If you are not required to use a Spring MVC portlet, consider using Liferay's MVC framework to design your portlets instead. Then you can take advantage of the Declarative Services `@Component` and `@Reference` annotations, which let you avoid the boilerplate code associated with service trackers.

63.4 Related Topics

Upgrading a Spring MVC Portlet

Using the WAB Generator

Chapter 64

JSF Portlets with Liferay Faces

Do you want to develop MVC-based portlets using the Java EE standard? Do you want to use a portlet development framework with a UI component model that makes it easy to develop sophisticated, rich UIs? Or have you been writing web apps using JSF that you'd like to use in Liferay DXP? If you answered *yes* to any of these questions, you're in luck! Liferay Faces provides all these capabilities and more.

Liferay Faces is an umbrella project that provides support for the JavaServer™ Faces (JSF) standard in Liferay DXP. It encompasses the following projects:

- **Liferay Faces Bridge** lets you deploy JSF web apps as portlets without writing portlet-specific Java code. It also contains innovative features that make it possible to leverage the power of JSF 2.x inside a portlet application. Liferay Faces Bridge implements the JSR 329/378 Portlet Bridge Standard.
- **Liferay Faces Alloy** lets you use AlloyUI components in a way that is consistent with JSF development.
- **Liferay Faces Portal** lets you leverage Liferay-specific utilities and UI components in JSF portlets.

For a comprehensive demo for the JSF component suite, visit the Liferay Faces Showcase.

If you're new to JSF, you may want to know its strengths, its weaknesses, and how it stacks up to developing portlets with CSS/JavaScript.

Here are some good reasons to use JSF and Liferay Faces:

- JSF is the Java EE standard for developing web applications that use the Model/View/Controller (MVC) design pattern. As a standard, the specification is actively maintained by the Java Community Process (JCP), and the Oracle reference implementation (Mojarra) has frequent releases. Software Architects often choose standards like JSF because they are supported by Java EE application server vendors and have a guaranteed service life according to Service Level Agreements (SLAs).
- JSF was first introduced in 2003 and is a mature technology for developing web applications that are (arguably) easy to maintain.
- JSF Portlet Bridges (like Liferay Faces Bridge) are also standardized by the JCP and make it possible to deploy JSF web applications as portlets without writing portlet-specific Java code.
- Support for JSF (via Liferay Faces) is included with Liferay DXP support.
- JSF is a unique framework in that it provides a UI component model that makes it easy to develop sophisticated, rich user interfaces.
- JSF has built-in Ajax functionality that provides automatic updates to the browser by replacing elements in the DOM.
- JSF is designed with many extension points that make a variety of integrations possible.
- There are several JSF component suites available including Liferay Faces Alloy, Primefaces, ICEfaces, and RichFaces. Each of these component suites fortify JSF with a variety of UI components and complimentary technologies such as Ajax Push.
- JSF is a good choice for server-side developers that need to build web user interfaces. This enables server-side developers to focus on their core competencies rather than being experts in HTML/CSS/JavaScript.
- JSF provides the Facelets templating engine which makes it possible to create reusable UI components that are encapsulated as markup.
- JSF provides good integration with HTML5 markup
- JSF provides the Faces Flows feature which makes it easy for developers to create wizard-like applications that flow from view-to-view.
- JSF has good integration with dependency injection frameworks such as CDI and Spring that make it easy for developers to create beans that are placed within a scope managed by a container: `@RequestScoped`, `@ViewScoped`, `@SessionScoped`, `@FlowScoped`
- Since JSF is a stateful technology, the framework encapsulates the complexities of managing application state so the developer doesn't have to write state management code. It is also possible to use JSF in a stateless manner, but some of the features of application state management become effectively disabled.

There are some reasons not to use JSF. For example, if you are a front-end developer who makes heavy use of HTML/CSS/JavaScript, you might find that JSF UI components render HTML in a manner that gives you less control over the overall HTML document. Sticking with JavaScript and leveraging AlloyUI or some other JavaScript framework may be better for you. Or, perhaps standards aren't a major consideration for you or you may simply prefer developing portlets using your current framework.

Whether you develop your next portlet application with JSF and Liferay Faces or with HTML/CSS/JavaScript is entirely up to you. But you probably want to learn more about Liferay Faces and try it out for yourself.

64.1 Generating a JSF Project from the Command Line

You can generate a Liferay Faces application without having to create your own folder structure, descriptor files, and such manually. If you really want to do that manually, you can examine the structure of a JSF application and create one from scratch in the Creating a JSF Project Manually tutorial.

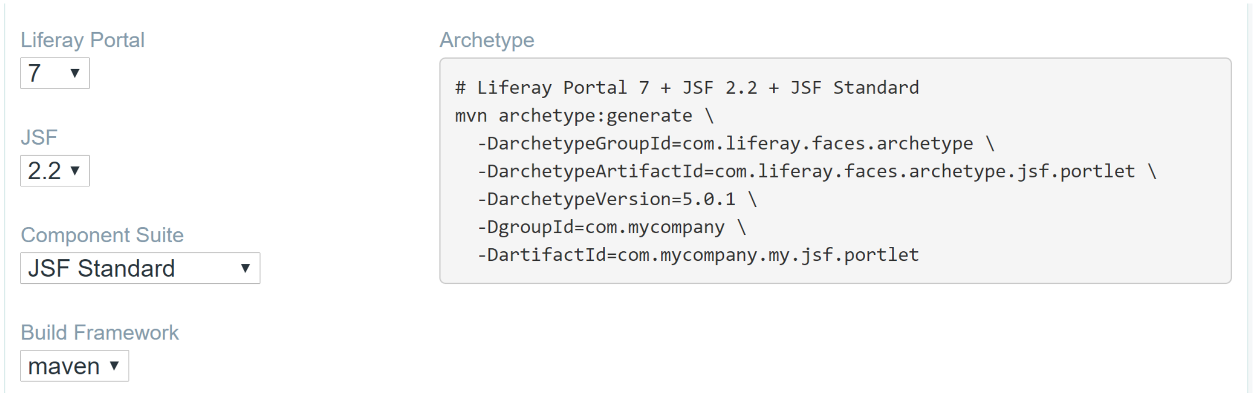
!PVideo Thumbnail

Before generating your JSF application, you should first visit liferayfaces.org, a great reference spot for JSF application development targeted for Liferay DXP. This site lets you choose the options for your JSF application and generates a Maven archetype command you can execute to generate an application with your chosen options. You can select the following archetype options:

- Liferay Portal Version
- JSF Version
- Component Suite

You can also choose a build framework (Gradle or Maven) and have a list of dependencies generated for you and displayed on the page. The dependencies are provided to you on the site page in a `pom.xml` or `build.gradle`, depending on the build type you selected. This is useful because it gives you an idea of what dependencies are required in your JSF application before generating it.

Note: Gradle developers can also use the `archetype:generate` command because it generates both a `build.gradle` and a `pom.xml` file for you to use.


Liferay Portal: 7
JSF: 2.2
Component Suite: JSF Standard
Build Framework: maven

Archetype

```
# Liferay Portal 7 + JSF 2.2 + JSF Standard
mvn archetype:generate \
  -DarchetypeGroupId=com.liferay.faces.archetype \
  -DarchetypeArtifactId=com.liferay.faces.archetype.jsf.portlet \
  -DarchetypeVersion=5.0.1 \
  -DgroupId=com.mycompany \
  -DartifactId=com.mycompany.my.jsf.portlet
```

Figure 64.1: You can select the Liferay Portal version, JSF version, and component suite for your archetype generation command.

Next you'll generate an example JSF application (e.g., Liferay Portal 7 + JSF 2.2 + JSF Standard) via command line using liferayfaces.org.

1. Navigate to liferayfaces.org and select the following options:

    - **Liferay Portal:** 7
    - **JSF:** 2.2
    - **Component Suite:** JSF Standard

2. Copy the archetype generation command and execute it. Make sure you've navigated to the folder where you want to generate your project.

That's it! Your JSF application is generated in the current folder!

You can also generate a Liferay JSF application using Maven's interactive archetype UI. To do this, execute `mvn archetype:generate -Dfilter=liferay` and select the JSF archetype you want to use. Then you'll step through each option and select the version, group ID, artifact ID, etc. To learn more about this, see the Generating New Projects Using Archetypes tutorial.

Once you have your JSF application generated, you can import it into Liferay @ide@ and develop it further. To deploy it to your Liferay DXP instance, drag and drop it onto the Liferay DXP server.

You can build the project and deploy it to Liferay DXP from the command line too! If you're using Gradle, run the following command to build your JSF application:

```
gradle build
```

For Maven, execute the following command:

```
mvn package
```

Then copy the generated WAR to Liferay DXP's `deploy` folder:

```
[cp|copy] ./com.mycompany.my.jsf.portlet.war LIFERAY_HOME/deploy
```

Awesome! You've generated your JSF application and deployed it using the command line.
!VVideo Tutorial

64.2 Generating a JSF Project Using @ide@

You can generate a Liferay Faces application without having to create your own folder structure and descriptor files manually using Liferay @ide@. If you're interested in creating the structure of a JSF application manually or if you want to examine a basic JSF application structure, visit the Creating a JSF Project Manually tutorial.

In this tutorial, you'll generate an example JSF project using Liferay @ide@. Open your @ide@ instance to get started.

1. Navigate to *File* → *New* → *Project...*. This opens a new project wizard.

2. Select the *Liferay* project and choose *Liferay JSF Project* from the listed subprojects. Then click *Next*.

3. Assign your JSF project a name, build framework (Gradle or Maven), and Component Suite. You have five component suites to choose from:

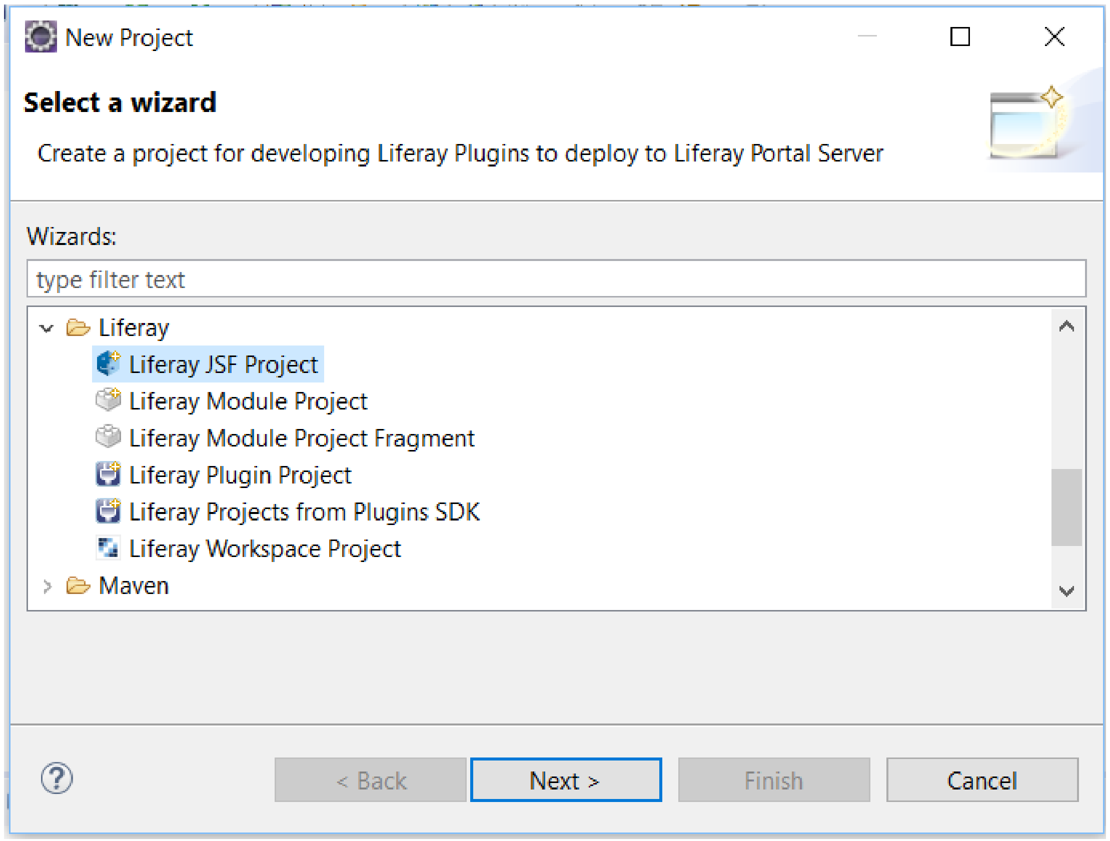
    - ICEFaces
    - JSF Standard
    - Liferay Faces Alloy
    - PrimeFaces
    - RichFaces

4. Click *Finish* to generate your Liferay JSF project.

You've generated a Liferay JSF project using @ide@! The project you generated contains a simple portlet that you can customize.

Note: There is another option in @ide@'s *File* → *New* menu named *Liferay JSF Portlet*. This is intended to add portlets to existing JSF projects. Currently, this is only configured to create Liferay Portal 6.2 JSF portlets. Do not use this option if you're developing for 7.0.

To deploy the new JSF project to your Liferay DXP instance, drag and drop it onto the Liferay server.
Fantastic! You're now able to quickly generate your Liferay JSF project using Liferay @ide@!


New Project

Select a wizard

Create a project for developing Liferay Plugins to deploy to Liferay Portal Server

Wizards:

type filter text

- Liferay
  - Liferay JSF Project
  - Liferay Module Project
  - Liferay Module Project Fragment
  - Liferay Plugin Project
  - Liferay Projects from Plugins SDK
  - Liferay Workspace Project
- Maven

< Back | Next > | Finish | Cancel

Figure 64.2: Choose the *Liferay JSF Project* option to begin creating a JSF project in @ide@.

64.3 Creating a JSF Project Manually

Liferay DXP's modular architecture lends itself well to modular applications created using a multitude of different technologies. JSF applications are no different and can be developed to integrate seamlessly into the Liferay platform.

In this tutorial, you'll step through packaging and creating a JSF application that is deployable as an OSGi module at runtime. First, you'll learn how to package a JSF application as a module.

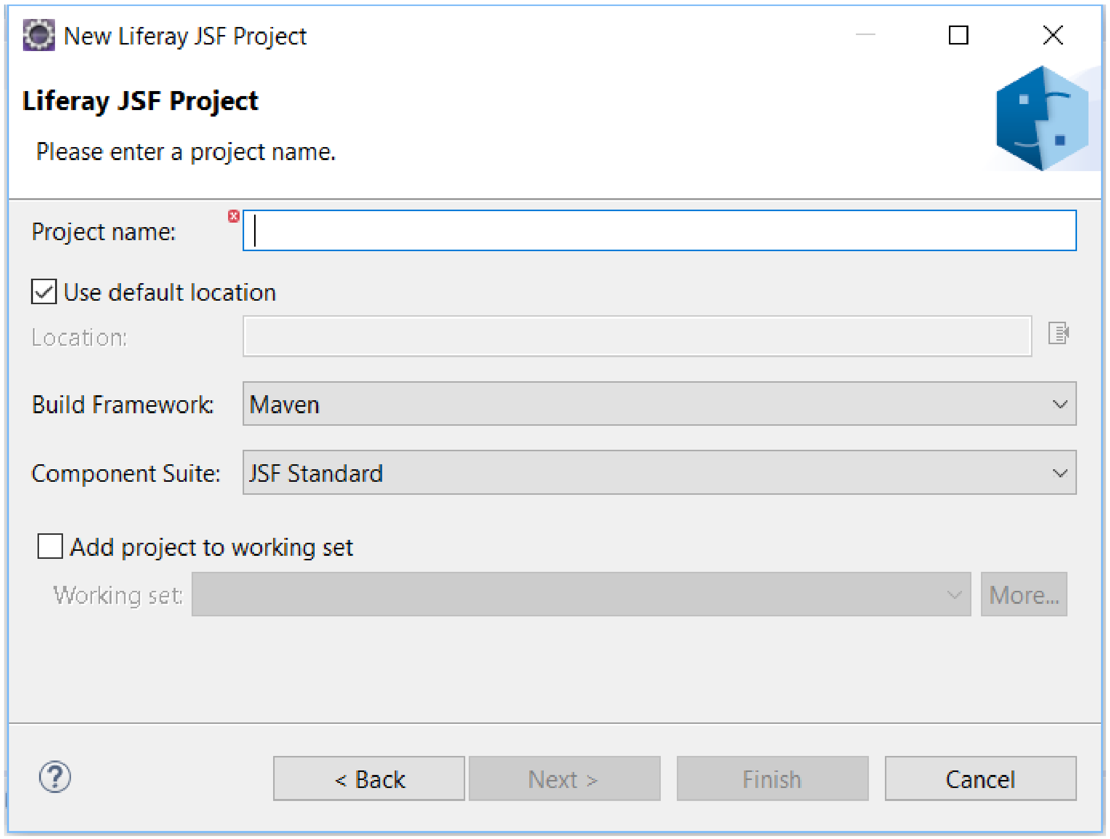
Packaging a JSF Application

Developers creating portlets for 7.0 can package their portlets as Java EE style Web Application ARchive (WAR) artifacts or as Java ARchive (JAR) OSGi bundle artifacts. JSF portlet developers, however, must package their portlets as WAR artifacts because the JSF framework expects a WAR layout and often requires the `WEB-INF/faces-config.xml` descriptor and other Java EE resources such as the `WEB-INF/web.xml` descriptor.

Liferay provides a way for these WAR-styled portlets to be deployed and treated like OSGi modules by Liferay's OSGi runtime. The WAB Generator does this automatically by converting your WAR artifact to a WAB at deployment time. You can learn more about WABs and the WAB Generator in the Using the WAB Generator tutorial.

This is how a JSF WAR artifact is structured:

- `jsf-portlet`
  - `src`


New Liferay JSF Project

Liferay JSF Project

Please enter a project name.

Project name:

☑ Use default location

Location:

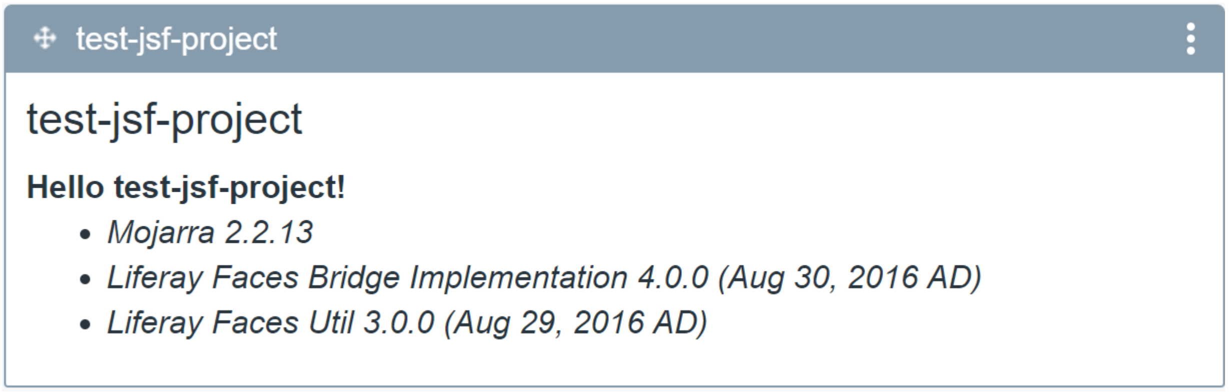
Build Framework: Maven

Component Suite: JSF Standard

☐ Add project to working set

Working set:

Figure 64.3: Choose your preferred options for your JSF project.


test-jsf-project

test-jsf-project

Hello test-jsf-project!

- *Mojarra 2.2.13*
- *Liferay Faces Bridge Implementation 4.0.0 (Aug 30, 2016 AD)*
- *Liferay Faces Util 3.0.0 (Aug 29, 2016 AD)*

Figure 64.4: The generated JSF portlet project displays basic build information.

* `main`

  - `java`
  - Java Classes

  - `resources`
  - Properties files

  - `webapp`
  - `WEB-INF/`
  - `classes/`
  - Class files and related properties

  - `lib/`
  - JAR dependencies

  - `resources/`
  - CSS, XHTML, PNG or other frontend files

  - `views/`
  - XHTML views

  - `faces-config.xml`
  - `liferay-display.xml`
  - `liferay-plugin-package.properties`
  - `liferay-portlet.xml`
  - `portlet.xml`
  - `web.xml`

Next, you'll begin creating a simple JSF application that is deployable to Liferay DXP.

Creating a JSF Application

JSF portlets are supported on Liferay Portal by using Liferay Faces Bridge. Liferay Faces Bridge makes developing JSF portlets as similar as possible to JSF web app development.

You'll create a simple *Hello User* application that asks for the user's name and then greets him or her with the name. You'll begin by creating the WAR-style folder structure, and then you'll configure dependencies like Liferay Faces Bridge.

1. Create a WAR-style folder structure for your module. Maven archetypes are available to help you get started quickly. They set the default configuration for you and contain boilerplate code so you can skip the file creation steps and get started right away. For your JSF application, you'll set up the folder structure manually. Follow the folder structure outline below:

```
- hello-user-jsf-portlet
    - src
        - main
            - java
            - resources
            - webapp
                - WEB-INF
                    - resources
                    - views
```

2. Make sure your module specifies the dependencies necessary for a Liferay JSF application. For instance, you must always specify the Faces API, Faces Reference Implementation (Mojarra), and Liferay Faces Bridge as dependencies in a Liferay-compatible JSF application. Also, an important, but not required, dependency is the Log4j logging utility. This is highly recommended for development purposes because it logs DEBUG messages in the console. You'll configure the logging utility later.

   For an example build file, the `pom.xml` file used for the Maven based Hello User JSF application is below. All the dependencies described above are configured in the Hello User JSF application's `pom.xml` file.

```
<?xml version="1.0"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.hello.user.jsf.portlet</artifactId>
    <packaging>war</packaging>
    <name>hello-user-jsf-portlet</name>
    <version>1.0-SNAPSHOT</version>
    <properties>
        <faces.api.version>2.2</faces.api.version>
        <liferay.faces.bridge.ext.version>5.0.0</liferay.faces.bridge.ext.version>
        <liferay.faces.bridge.version>4.0.0</liferay.faces.bridge.version>
        <mojarra.version>2.2.13</mojarra.version>
    </properties>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <encoding>UTF-8</encoding>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.3</version>
                <configuration>
                    <filteringDeploymentDescriptors>true</filteringDeploymentDescriptors>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>javax.faces</groupId>
            <artifactId>javax.faces-api</artifactId>
            <version>${faces.api.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.glassfish</groupId>
            <artifactId>javax.faces</artifactId>
            <version>${mojarra.version}</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>com.liferay.faces</groupId>
            <artifactId>com.liferay.faces.bridge.ext</artifactId>
            <version>${liferay.faces.bridge.ext.version}</version>
        </dependency>
        <dependency>
            <groupId>com.liferay.faces</groupId>
            <artifactId>com.liferay.faces.bridge.impl</artifactId>
            <version>${liferay.faces.bridge.version}</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.14</version>
        </dependency>
    </dependencies>
</project>
```

There are also two plugins the Hello User JSF application defined in its pom.xml: maven-compiler-plugin and maven-war-plugin. These two plugins are responsible for building and compiling the JSF application using Maven.

There are several UI component suites that a JSF application can use, which include *Liferay Faces Alloy*, *PrimeFaces*, *ICEfaces*, and *RichFaces*. Furthermore, you can take advantage of *Liferay Faces Portal* in order to use Liferay-specific utilities and UI components. These components can be used by specifying them as dependencies in your build file, as well.

Now that your build file is configured, you must define the JSF-specific configurations for your application. These fall into two convenient categories: general descriptors and Liferay descriptors. You'll start with creating the necessary general descriptors.

Defining JSF Portlet Descriptors

Since JSF portlets must follow a WAR-style folder structure, they must also have WAR-style portlet descriptors.

1. Create a portlet.xml file in the webapp/WEB-INF folder. All portlet WARs require this file. In this file, make sure to declare the following portlet class:

```
<portlet>
    ...
    <portlet-class>javax.portlet.faces.GenericFacesPortlet</portlet-class>
    ...
</portlet>
```

   The javax.portlet.faces.GenericFacesPortlet class handles invocations to your JSF portlet and makes your portlet, since it relies on Liferay Faces Bridge, easy to develop by acting as a turnkey implementation.

2. Define a default view file as an init-param in the portlet.xml. This ensures your portlet is visible when deployed to Liferay DXP.

```
<init-param>
    <name>javax.portlet.faces.defaultViewId.view</name>
    <value>/WEB-INF/views/view.xhtml</value>
</init-param>
```

   You'll create this view later.

   The portlet.xml file holds other important details too, like portlet info and security settings. Look at the portlet.xml file for the example Hello User JSF application.

```
<?xml version="1.0"?>

<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd http://java.sun.com/xml/ns/portlet/portlet-
app_2_0.xsd" version="2.0">
    <portlet>
        <portlet-name>hello-user-jsf-portlet</portlet-name>
        <display-name>Hello User JSF Portlet</display-name>
        <portlet-class>javax.portlet.faces.GenericFacesPortlet</portlet-class>
        <init-param>
            <name>javax.portlet.faces.defaultViewId.view</name>
            <value>/WEB-INF/views/view.xhtml</value>
        </init-param>
        <expiration-cache>0</expiration-cache>
        <supports>
            <mime-type>text/html</mime-type>
        </supports>
        <portlet-info>
            <title>Hello User JSF Portlet</title>
            <short-title>Hello User</short-title>
            <keywords>com.liferay.hello.user.jsf.portlet</keywords>
        </portlet-info>
        <security-role-ref>
            <role-name>administrator</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>guest</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>power-user</role-name>
        </security-role-ref>
        <security-role-ref>
            <role-name>user</role-name>
        </security-role-ref>
    </portlet>
</portlet-app>
```

The above configuration sets your portlet's various names, MIME type, expiration cache, and security roles.

3. Create a `web.xml` file in your JSF application's `webapp/WEB-INF` folder. The `web.xml` file serves as a deployment descriptor that provides necessary configurations for your JSF portlet to deploy and function in Liferay DXP. Copy the XML code below into your Hello User JSF application.

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">

    <context-param>
        <param-name>javax.faces.PROJECT_STAGE</param-name>
        <param-value>${project.stage}</param-value>
    </context-param>
    <context-param>
        <param-name>javax.faces.WEBAPP_RESOURCES_DIRECTORY</param-name>
        <param-value>/WEB-INF/resources</param-value>
    </context-param>
    <servlet>
        <servlet-name>Faces Servlet</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <security-constraint>
        <display-name>Prevent direct access to Facelet XHTML</display-name>
        <web-resource-collection>
```

```
            <web-resource-name>Facelet XHTML</web-resource-name>
            <url-pattern>*.xhtml</url-pattern>
        </web-resource-collection>
        <auth-constraint/>
    </security-constraint>
</web-app>
```

First, you set the `javax.faces.PROJECT_STAGE` parameter to the `${project.stage}` variable, which is defined in your build file (e.g., `pom.xml`) as `Development`. When set to `Development`, the JSF implementation will perform the following steps at runtime:

1. Log more verbose messages.
2. Render tips and/or warnings in the view markup.
3. Cause the default `ExceptionHandler` to display a developer-friendly error page.

The `javax.faces.WEBAPP_RESOURCES_DIRECTORY` parameter sets the resources folder inside the `WEB-INF` folder. This setting makes the resources in that folder (e.g., CSS, JavaScript, XHTML) secure from non-JSF calls. You'll create resources for your app later.

The Faces Servlet configuration is required to initialize JSF and should be defined in all JSF portlets deployed to Liferay DXP.

Finally, a security restraint is set on Facelet XHTML, which prevents direct access to XHTML files in your JSF application.

4. Create a `faces-config.xml` file in your JSF application's `webapp/WEB-INF` folder. The `faces-config.xml` descriptor is a JSF portlet's application configuration file, which is used to register and configure objects and navigation rules. The Hello User portlet's `faces-config.xml` file has the following contents:

```
<?xml version="1.0"?>

<faces-config version="2.2"
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd"
>
    <lifecycle>
        <phase-listener>com.liferay.faces.util.lifecycle.DebugPhaseListener</phase-listener>
    </lifecycle>
</faces-config>
```

Many auto-generated `faces-config.xml` files have the following configuration:

```
<lifecycle>
    <phase-listener>com.liferay.faces.util.lifecycle.DebugPhaseListener</phase-listener>
</lifecycle>
```

This configures your JSF portlet to log the before/after phases of the JSF lifecycle to your console in debug mode. Remove this declaration before deploying to production.

Great! You now have a good idea of how to specify and define general descriptor files for your JSF portlet. JSF portlets also use Liferay descriptors, which you can learn more about in the Liferay Descriptors sub-section.

Now that your portlet descriptors are defined, you should begin working on your JSF application's resources.

Defining Resources for a JSF Application

If you look back at the Hello User portlet's structure, you'll notice two `resources` folders defined. Why are there two of these folders for one portlet? These two folders have distinct differences in how they're used and what should be placed in them.

The `resources` folder in the application's `src/main` folder is intended for resources that need to be on the classpath. Files in this folder are usually properties files. For this portlet, you'll create two properties files to reside in this folder.

1. Create the `i18n.properties` file in the `src/main/resources` folder. Add the following property to this file:

   ```
   enter-your-name=Enter your name:
   ```

   This is a language key your JSF portlet displays in its view XHTML. The messages in the `i18n.properties` file can be accessed via the Expression Language using the implicit `i18n` object provided by the Liferay Faces Util class. The `i18n` object can access messages both from a resource bundle defined in the portlet's `portlet.xml` file, and from Liferay DXP's `Language.properties` file.

2. Create the `log4j.properties` file in the `src/main/resources` folder. This file sets properties for the Log4j logging utility defined in your JSF application (i.e., `faces-config.xml`). Insert the properties below into your JSF application's `log4j.properties` file.

   ```
   log4j.rootLogger=INFO, CONSOLE

   log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
   log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
   log4j.appender.CONSOLE.layout.ConversionPattern=%d{ABSOLUTE} %-5p [%c{1}:%L] %m%n

   log4j.logger.com.liferay.faces.util.lifecycle.DebugPhaseListener=DEBUG
   ```

The second `resources` folder in your JSF application is located in the `src/main/webapp/WEB-INF` folder. This folder holds CSS/JS/XHTML resources that shouldn't be accessed directly by the browser. For the Hello User JSF application, create a `css` folder with a `main.css` file inside. In the `main.css` file, add the following style:

```
.com.liferay.hello.user.jsf.portlet {
    font-weight: bold;
}
```

This file gives your JSF portlet a bold font.

Now that your resources are defined, it's time to begin developing the Hello User application's behavior and UI.

Developing a JSF Application's Behavior and UI

Your current JSF application satisfies the requirements for portlet descriptors and WAR-style structure, but it doesn't do anything yet. You'll learn how to develop a JSF application's back-end and give it a simple UI next.

The first thing to do is create a Java class for your module. Your JSF portlet's behavior is defined here. In the case of the Hello User portlet, you should provide Java methods that can get/set a name and facilitate the submission process.

1. Create a unique package name in the module's `src/main/java` folder and create a new public Java class named `ExampleBacking.java` in that package. For example, the class's folder structure could be `src/main/java/com/liferay/example/ExampleBacking.java`. Make sure the class is annotated with @RequestScoped and @ManagedBean:

   ```
   @RequestScoped
   @ManagedBean
   public class ExampleBacking {
   ```

   Managed beans are Java beans that are managed by the JSF framework. Managed beans annotated with `@RequestScoped` are usually responsible for handling actions and listeners. JSF *manages* these beans by creating and removing the bean object from the server. Visit the linked annotations above for more details.

2. Add the following methods and field to your `ExampleBacking.java` class:

   ```
   public String getName() {
       return name;
   }

   public void setName(String name) {
       this.name = name;
   }

   public void submit(ActionEvent actionEvent) {
       FacesContextHelperUtil.addGlobalSuccessInfoMessage();
   }

   private String name;
   ```

   You've provided a getter and setter method for the private `name` field. You've also provided a `submit(...)` method, which is called when the *Submit* button is selected. A success info message is displayed once the method is invoked.

   You've defined your Hello User portlet's Java behavior; now create its UI!

3. Create a `view.xhtml` file in the `webapp/WEB-INF/views` folder. Add the following logic to that file:

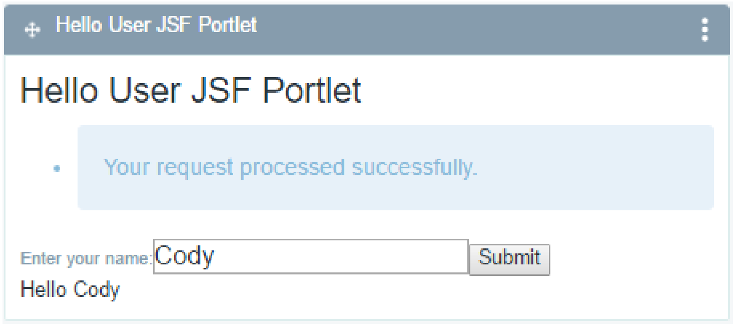
   ```
   <?xml version="1.0"?>

   <f:view
       xmlns="http://www.w3.org/1999/xhtml"
       xmlns:f="http://java.sun.com/jsf/core"
       xmlns:h="http://java.sun.com/jsf/html"
   >
       <h:head>
           <h:outputStylesheet library="css" name="main.css" />
       </h:head>
       <h:form>
           <h:messages globalOnly="true" />
           <h:outputLabel value="#{i18n['enter-your-name']}" />
           <h:inputText value="#{exampleBacking.name}" />
           <h:commandButton actionListener="#{exampleBacking.submit}" value="#{i18n['submit']}">
               <f:ajax execute="@form" render="@form" />
           </h:commandButton>
           <br />
           <h:outputText value="Hello #{exampleBacking.name}" />
       </h:form>
   </f:view>
   ```

The first thing to notice is the main.css file you created is specified here, which makes your portlet's heading typeface bold. Next, your form is defined within the <h:form> tags. The form asks the user to enter his or her name, and then sets that value to the name field in your Java class. The form uses the <h:commandButton> tag to execute the Submit button and render the form after submission.

Notice the i18n object call for the enter-your-name and submit properties. The enter-your-name property was set in the i18n.properties file you specified, but what about the submit property? This was not defined in your portlet's i18n.properties file, so how does your portlet know to use the string *Submit* for your button? If you recall, the i18n object can also access messages in Liferay DXP's Language.properties file. This is where the submit language key comes from.

Finally, the <h:outputText> tag prints the submitted name on the page, prefixed with *Hello*.

Awesome! Your Hello User JSF application is complete! Deploy your WAR to Liferay DXP. Remember, when your WAR-style portlet is deployed, it's converted to a WAB via the WAB Generator. Visit the Using the WAB Generator tutorial for more information on this process and your portlet's resulting folder structure.



Your request processed successfully.

Enter your name: Cody

Hello Cody

Figure 64.5: After submitting the user's name, it's displayed with a greeting.

To recap, you created your JSF application in the following steps:

- Construct the WAR-style folder structure.
- Specify the necessary dependencies in a build file of your choice.
- Create JSF portlet descriptors and Liferay descriptors.
- Add resource files in the two designated resources folders.
- Define the portlet's behavior using a Java class.
- Design a view XHTML form to let the user interact with the portlet.

You can view the finished version of the Hello User JSF application by downloading its ZIP file.
Now you have the knowledge to create your own JSF applications!

Related Topics

Fundamentals
    Internationalization
    Configuration

64.4 Services in JSF

Creating services works the same in a JSF portlet as it would in any other standard WAR-style MVC portlet; generate custom services as separate API and Impl JARs and deploy them as individual modules to Liferay

DXP. You can generate custom services for your JSF portlet using Service Builder. To learn more about how Service Builder works in Liferay DXP, visit the Service Builder tutorials.

The JSF WAR can then rely on the API module as a *provided* dependency. The main benefit for packaging your services this way is to allow multiple WARs to utilize the same custom service API without packaging it inside every WAR's `WEB-INF/lib` folder. This practice also enforces a separation of concerns, or *modularity*, between the UI layer and service layer of a system.

To call OSGi-based Service Builder services from your JSF portlet, you need a mechanism that gives you access to the OSGi service registry, because you can't look up services published to the OSGi runtime using Declarative Services. Instead, you should open a ServiceTracker when you want to call a service that's in the OSGi service registry.

To implement a service tracker in your JSF portlet, you can add a type-safe wrapper class that extends `org.osgi.util.tracker.ServiceTracker`. For example, this is done in a demo JSF portlet as follows

```
public class UserLocalServiceTracker extends ServiceTracker<UserLocalService, UserLocalService> {

    public UserLocalServiceTracker(BundleContext bundleContext) {
        super(bundleContext, UserLocalService.class, null);
    }
}
```

After extending the `ServiceTracker`, just call the constructor and the service tracker is ready to use in your managed bean.

In a managed bean, whenever you need to call a service, open the service tracker. For example, this is done in the same demo JSF portlet to open the service tracker, using the @PostContruct annotation:

```
@PostConstruct
public void postConstruct() {
    Bundle bundle = FrameworkUtil.getBundle(this.getClass());
    BundleContext bundleContext = bundle.getBundleContext();
    userLocalServiceTracker = new UserLocalServiceTracker(bundleContext);
    userLocalServiceTracker.open();
}
```

Then the service can be called:

```
UserLocalService userLocalService = userLocalServiceTracker.getService();
...

userLocalService.updateUser(user);
```

When it's time for the managed bean to go out of scope, you must close the service tracker using the @PreDestroy annotation:

```
@PreDestroy
public void preDestroy() {
    userLocalServiceTracker.close();
}
```

For more information on service trackers and how to use them in WAR-style portlets, see the Service Trackers tutorial.

Related Topics

Fundamentals
    Internationalization
    Configuration

Chapter 65

Making URLs Friendlier

This is a story of two URLs who couldn't be more different. One was full of himself, and always wanted to show everyone (users and SEO services alike) just how smart he was, by openly displaying all of the parameters he carried. He was happiest when he met new people and could tell they were intimidated and confused by him.

```
http://localhost:8080/group/guest/~/control_panel/manage?p_p_id=com_liferay_blogs_web_portlet_BlogsAdminPortlet&p_p_lifecycle=0&p_p_state=maximized&p_p_mode
```

The other was just, well, friendly. You knew only the important things about her, because she was less concerned about looking smart, and more concerned about those she interacted with. She didn't need to look fancy and complicated. She aspired to be simple and kind to all the users and SEO services she encountered.

```
http://localhost:8080/web/guest/home/-/blogs/lunar-scavenger-hunt
```

If you want your application to be friendly to your users and to SEO services, make your URLs friendlier. It only takes a couple steps, after all.

65.1 Creating Friendly URL Routes

1. First create a `routes.xml` file in your application's web module (if a multi-module build, in the pattern of Liferay's native Service Builder applications). It's recommended to put it in a `src/main/resources/META-INF/friendly-url-routes/` folder.

2. Add friendly URL routes, using as many <route> tags as you need friendly URLs, like this:

```
 <?xml version="1.0"?>
 <!DOCTYPE routes PUBLIC "-//Liferay//DTD Friendly URL Routes 7.0.0//EN" "http://www.liferay.com/dtd/liferay-friendly-url-
routes_7_0_0.dtd">

 <routes>
     <route>
         <pattern></pattern>
         <implicit-parameter name="mvcRenderCommandName">/blogs/view</implicit-parameter>
         <implicit-parameter name="p_p_lifecycle">0</implicit-parameter>
         <implicit-parameter name="p_p_state">normal</implicit-parameter>
     </route>
     <route>
         <pattern>/maximized</pattern>
         <implicit-parameter name="mvcRenderCommandName">/blogs/view</implicit-parameter>
         <implicit-parameter name="p_p_lifecycle">0</implicit-parameter>
```

```
            <implicit-parameter name="p_p_state">maximized</implicit-parameter>
        </route>
        <route>
            <pattern>/{entryId:\d+}</pattern>
            <implicit-parameter name="categoryId"></implicit-parameter>
            <implicit-parameter name="mvcRenderCommandName">/blogs/view_entry</implicit-parameter>
            <implicit-parameter name="p_p_lifecycle">0</implicit-parameter>
            <implicit-parameter name="p_p_state">normal</implicit-parameter>
            <implicit-parameter name="tag"></implicit-parameter>
        </route>
        ...
    </routes>
```

Use <pattern> tags to define placeholder values for the parameters that normally appear in the generated URL. This is just a mask. The beastly URL still lurks beneath it.

The pattern value /{entryId:\d+} matches a / followed by an entryId variable that matches the Java regular expression \d+—one or more numeric digits. For example, a URL /entryId, where the entryId value is 123 results in a URL value /123, which matches the pattern.

Warning: Make sure your pattern values don't end in a slash /. A trailing slash character prevents the request from identifying the correct route.

Important: If your portlet is instanceable, you must use a variant of the instanceId in the pattern value. If the starting value is render-it, for example, use one of these patterns:

```
<pattern>/{userIdAndInstanceId}/render-it</pattern>
```

or

```
<pattern>/{instanceId}/render-it</pattern>
```

or

```
<pattern>/{p_p_id}/render-it</pattern>
```

Use <implicit-parameter> tags to define parameters that will always be the same for the URL. For example, if you're dealing with a render URL, you can be certain that the p_p_lifecycle parameter will always be 0. There's no need for this to be generated. You don't have to define these type of implicit parameters, but it's a best practice. If you happen to forget one, or decide not to define any of them, they'll just be generated as usual.

The implicit parameters with the name mvcRenderCommandName are very important. If you're using an MVCPortlet with MVCRenderCommand classes, that parameter comes from the mvc.command.name property in the @Component of your MVCRenderCommand implementation. Basically, this determines what will be rendered (for example, view.jsp).

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS, "mvc.command.name=/",
        "mvc.command.name=/blogs/view"
    },
    service = MVCRenderCommand.class
)
```

65.2 Implementing a Friendly URL Mapper

Once you have your URLs mapped in a `routes.xml` file, you need to provide an implementation of the `FriendlyURLMapper` service. Just create a component that specifies a `FriendlyURLMapper` service, with two properties:

1. One that sets the path to your `routes.xml` file in the property `com.liferay.portlet.friendly-url-routes` property.

2. A `javax.portlet.name` property.

```
@Component(
    property = {
        "com.liferay.portlet.friendly-url-routes=META-INF/friendly-url-routes/routes.xml",
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS
    },
    service = FriendlyURLMapper.class
)
```

After that, implement the `FriendlyURLMapper` service. For your convenience, the `DefaultFriendlyURLMapper` class provides a default implementation. If you extend `DefaultFriendlyURLMapper` you only need to override one method, `getMapping()`. In this method you just need to return a String that defines the first part of your Friendly URLs. It's smart to name it after your application. Here's what it looks like for Liferay's Blogs application:

```
public class BlogsFriendlyURLMapper extends DefaultFriendlyURLMapper {

    @Override
    public String getMapping() {
        return _MAPPING;
    }

    private static final String _MAPPING = "blogs";

}
```

All of the Blogs application's friendly URLs begin with the String set here (`blogs`).

65.3 Friendly URLs in Action

Let's look at one of these Friendly URLs in action. If you add a blog entry in Liferay, and then add the Blogs application to a page, click on the title of the entry to see it. After that, look at the URL:

```
http://localhost:8080/web/guest/home/-/blogs/lunar-scavenger-hunt
```

As specified in the friendly URL mapper class, `blogs` is the first part of the friendly URL that comes after the Liferay part of the URL. The next part is determined by a specific URL route in `routes.xml`:

```
<route>
    <pattern>/{urlTitle}</pattern>
    <implicit-parameter name="categoryId"></implicit-parameter>
    <implicit-parameter name="mvcRenderCommandName">/blogs/view_entry</implicit-parameter>
    <implicit-parameter name="p_p_lifecycle">0</implicit-parameter>
    <implicit-parameter name="p_p_state">normal</implicit-parameter>
    <implicit-parameter name="tag"></implicit-parameter>
</route>
```

Here, the `urlTitle` is a database field that's generated from the title the author gives their blog post, and it's meant to be used in a URL. Since it's already a parameter in the URL (see below), it's available for use in the friendly URL.

```
<portlet:renderURL var="viewEntryURL">
    <portlet:param name="mvcRenderCommandName" value="/blogs/view_entry" />
    <portlet:param name="urlTitle" value="<%= entry.getUrlTitle() %>" />
</portlet:renderURL>
```

When a render URL for viewing a blog entry is invoked, the String defined in the friendly URL mapper teams up with the `pattern` tag in your friendly URL routes file, and you get a very friendly URL indeed, instead of some nasty, conceited, unfriendly URL that's despised by users and SEO services alike.

65.4 Automatic Single Page Applications

A good user experience is the measure of a well-designed site. A user's time is highly valuable. The last thing you want is for someone to grow frustrated with your site because of constant page reloads. A Single Page Application (SPA) is the solution. Single Page Applications drastically cut down on load times by loading only a single HTML page that is dynamically updated as the user interacts and navigates through the site. This provides a native-app experience by eliminating page loads. In Liferay DXP, **SPA is enabled by default in your apps and sites and requires no changes to your workflow or code!**

This tutorial covers these key topics:

- The benefits of SPAs
- What is SennaJS?
- How to enable SPA in Liferay DXP
- How to configure SPA settings
- How to listen to SPA lifecycle events

The Benefits of SPAs

Let's say you're surfing the web and you find a really rad site that happens to be SPA enabled. Alright! Page load times are blazin' fast. You're deep into the site, scrolling along, when you find this great post that just speaks to you. You copy the URL from the address bar and email it to all of your friends with the subject: 'Your Life Will Change Forever.' They must experience this awe-inspiring work!

You get a response back almost immediately. "This is a rad site, but what post are you talking about?" it reads.

"What!? Do my eyes deceive me?" you exclaim. You were in so much of a hurry to share this life-changing content that you neglected to notice that the URL never updated when you clicked the post. You click the back button, hoping to get back to the post, but it takes you to the site you were on before you ever visited this one.

What a bummer! "Why? Why have you failed me site?" you cry.

If only there was a way to have a Single Page Application, but also be able to link to the content you want. Well, don't despair my friend. You can have your cake and eat it too, thanks to SennaJS.

What is SennaJS?

SennaJS is Liferay DXP's SPA engine. SennaJS handles the client-side data, and AJAX loads the page's content dynamically. While there are other JavaScript frameworks out there that may provide some of the same features, Senna's only focus is SPA, ensuring that your site provides the best user experience possible.

SennaJS provides the following key enhancements to SPA:

SEO & Bookmarkability: Sharing or bookmarking a link displays the same content you are viewing. Search engines are able to index this content.

Hybrid rendering: Ajax + server-side rendering lets you disable pushState at any time, allowing progressive enhancement. You can use your preferred method to render the server side (e.g. HTML fragments or template views).

State retention: Scrolling, reloading, or navigating through the history of the page takes you back to where you were.

UI feedback: The UI indicates to the user when some content is requested.

Pending navigations: UI rendering is blocked until data is loaded, and the content is displayed all at once.

Timeout detection: If the request takes too long to load or the user tries to navigate to a different link while another request is pending, the request times out.

History navigation: The browser history is manipulated via the History API, so you can use the back and forward history buttons to navigate through the history of the page.

Cacheable screens: Once a surface is loaded, the content is cached in memory and is retrieved later without any additional request, speeding up your application.

Page resources management: Scripts and stylesheets are evaluated from dynamically loaded resources. Additional content can be appended to the DOM using XMLHttpRequest. For security reasons, some browsers won't evaluate `<script>` tags from content fragments. Therefore, SennaJS extracts scripts from the content and parses them to ensure that they meet the browser loading requirements.

You can read more about SennaJS as well as see examples at http://sennajs.com/.

Now that you have a better understanding of how SennaJS benefits SPA, you can learn how to enable and configure options for SPA within Liferay DXP next.

Enabling SPA

Enabling SPA is easy. Deploy `com.liferay.frontend.js.spa.web-[version]` module deployed and enabled, and you're all set to use SPA. Since this module is included with Liferay DXP by default, you shouldn't have to do anything.

SPA is enabled by default in your apps and sites, and requires no changes to your workflow or existing code!

Next you can learn how to customize SPA settings to meet your own needs.

Customizing SPA Settings

Depending on what behaviors you need to customize, you can configure SPA options in one of two places. SPA caching and SPA timeout settings can be configured in System Settings. If you wish to disable SPA for a certain link, page, or portlet in your site, you can do so within the corresponding element itself. All SPA configuration options are covered here.

Configuring SPA System Settings

To configure system settings for SPA, follow these steps:

1. In the Control Panel, navigate to *Configuration → System Settings*.

2. Select the *Foundation* tab at the top of the page.

3. Click *Frontend SPA Infrastructure*.

Note: In prior versions of Liferay, all SPA render requests that didn't belong to a portlet (no `p_p_id` in the URL) were cached indefinitely. This can confuse users, as the content they view is cached rather than the latest fresh content. Since Liferay Portal CE 7.0 GA2 and Liferay DXP 7.0 GA1, administrators can use the **Cache Expiration Time** property to set an expiration time for the Senna cache.

The following configuration options are available:

Cache Expiration Time: The time, in minutes, in which the SPA cache is cleared. A negative value means the cache should be disabled.

Request Timeout Time: The time, in milliseconds, in which a SPA request times out. A zero value means the request should never timeout.

User Notification Time: The time, in milliseconds, in which a notification is shown to the user stating that the request is taking longer than expected. A zero value means no notification should be shown.

Now that you know how to configure system settings for SPA, you can learn how to disable SPA for elements in your site next.

Disabling SPA

Certain elements of your page may require a regular navigation to work properly. For example, you may have downloadable content that you want to share with the user. In these cases, SPA must be disabled for those specific elements.

To disable SPA on a portal wide basis, you can add the following line to your `portal-ext.properties`:

```
javascript.single.page.application.enabled = false
```

If there is a portlet or element that you don't want to be part of the SPA, you have some options:

- Blacklist the portlet to disable SPA for the entire portlet
- Use the `data-senna-off` annotation to disable SPA for a specific form or link

To blacklist a portlet from SPA, follow these steps:

1. Open your portlet class.

2. Set the `com.liferay.portlet.single-page-application` property to false:

   ```
   com.liferay.portlet.single-page-application=false
   ```

   If you prefer, you can set this property to false in your `liferay-portlet.xml` instead by adding the following property to the `<portlet>` section:

   ```
   <single-page-application>false</single-page-application>
   ```

3. Alternatively, you can override the `isSinglePageApplication` method of the portlet to return `false`.

To disable SPA for a form or link follow these steps:

1. Add the `data-senna-off` attribute to the element.

2. Set the value to `true`.

For example `<a data-senna-off="true" href="/pages/page2.html">Page 2</a>`

That's all you need to do to disable SPA in your app.

Now that you know how to disable SPA, you can learn how to specify how resources are loaded during navigation.

Specifying How Resources Are Loaded During Navigation

By default, Liferay DXP unloads CSS resources from the `<head>` element on navigation. JavaScript resources in the `<head>`, however, are not removed on navigation. This functionality can be customized by setting the resource's `data-senna-track` attribute. Follow these steps to customize your resources:

1. Select the resource you want to modify the default behavior for.
2. Add the `data-senna-track` attribute to the resource.
3. Set the `data-senna-track` attribute to `permanent` to prevent a resource from unloading on navigation.

   Alternatively, set the `data-senna-track` attribute to `temporary` to unload the resource on navigation.

```
**Note:** the `data-senna-track` attribute can be added to resources loaded
outside of the `<head>` element as well to specify navigation behavior.
```

The example below ensures that the JS resource isn't unloaded during navigation:

```
<script src="myscript.js" data-senna-track="permanent" />
```

Next you can learn about the available SPA lifecycle events next.

Listening to SPA Lifecycle Events

During development, you may need to know when navigation has started or stopped in your SPA. SennaJS makes this easy by exposing lifecycle events that represent state changes in the application. The following events are available:

beforeNavigate: Fires before navigation starts. This event passes a JSON object with the path to the content being navigated to and whether to update the history. Below is an example event payload:

```
{ path: '/pages/page1.html', replaceHistory: false }
```

startNavigate: Fires when navigation begins. Below is an example event payload:

```
{ form: '<form name="form"></form>', path: '/pages/page1.html',
replaceHistory: false }
```

endNavigate: Fired after the content has been retrieved and inserted onto the page. This event passes the following JSON object:

```
{ form: '<form name="form"></form>', path: '/pages/page1.html' }
```

These events can be leveraged easily by listening for them on the Liferay global object.

For example, the JavaScript below alerts the user to "Get ready to navigate to" the URL that has been clicked, just before SPA navigation begins.

```
Liferay.on('beforeNavigate', function(event) {
    alert("Get ready to navigate to " + event.path);
});
```

The alert takes advantage of the payload for the `beforeNavigate` event, retrieving the URL from the `path` attribute of the JSON payload object.

The above code results in the behavior shown below:

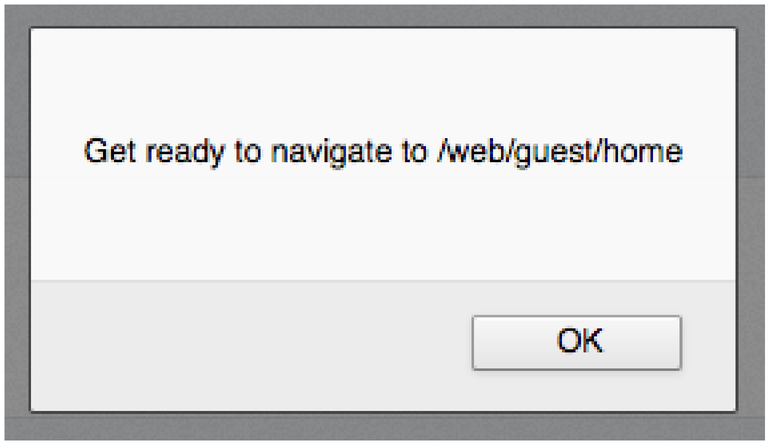

Get ready to navigate to /web/guest/home

Figure 65.1: You can leverage SPA lifecycle events in your apps.

Due to the nature of SPA navigation, global listeners that you create can become problematic over time if not handled properly. You'll learn how to handle these listeners next.

Detaching Global Listeners

SPA provides several improvements that highly benefit your site and users, but there is potentially some additional maintenance as a consequence. In a traditional navigation scenario, every page refresh resets everything, so you don't have to worry about what's left behind. In a SPA scenario, however, global listeners such as `Liferay.on` or `Liferay.after` or body delegates can become problematic. Every time you execute these global listeners, you add yet another listener to the globally persisted `Liferay` object. The result is multiple invocations of those listeners. This can obviously cause problems if not handled.

To prevent this, you need to listen to the navigation event in order to detach your listeners. For example, you would use the following code to detach the event listeners of a global `category` event:

```
var onCategory = function(event) {...};

var clearPortletHandlers = function(event) {
    if (event.portletId === '<%= portletDisplay.getRootPortletId() %>') {
        Liferay.detach('onCategoryHandler', onCategory);
        Liferay.detach('destroyPortlet', clearPortletHandlers);
    }
};


Liferay.on('category', onCategory);
Liferay.on('destroyPortlet', clearPortletHandlers);
```

Now you know how to configure and use SPA in Liferay DXP!

Related Topics

Configuring Modules for Liferay Portal's Loaders
Preparing your JavaScript Files for ES2015
Using ES2015 Modules in Your Portlet

65.5 Creating Layouts inside Custom Portlets

Page layout tags let you create layouts using Bootstrap 3 within your portlets.

This tutorial explains the `<aui:*>` tags that developers can use to create layouts.

AUI Container

The `<aui:container>` tag creates a container `<div>` tag to wrap `<aui:row>` components and offer additional styling.

It supports the following attributes:

Attribute	Type	Description
cssClass	String	A CSS class for styling the component
dynamicAttributes	Map<String, Object>	Map of `data-` attributes for your container
fluid	boolean	Whether to enable the container to span the entire width of the viewport. The default value is true
id	String	An ID for the component instance

AUI Row

The `<aui:row>` tag creates a row to hold `<aui:col>` components.

It supports the following attributes:

Attribute	Type	Description
cssClass	String	A CSS class for styling the component
id	String	An ID for the component instance

AUI Col

The `<aui:col>` tag creates a column to display content in an `<aui:row>` component.

It supports the following attributes:

Attribute	Type	Description
cssClass	String	A CSS class for styling the component.
id	String	An ID for the component instance.
lg	String	Comma separated string of numbers 1-12 to be used for Boostrap grid `col-lg-`
md	String	Comma separated string of numbers 1-12 to be used for Boostrap grid `col-md-`
sm	String	Comma separated string of numbers 1-12 to be used for Boostrap grid `col-sm-`
xs	String	Comma separated string of numbers 1-12 to be used for Boostrap grid `col-xs-`
span	int	The width of the column in the containing row as a fraction of 12. For example, a span of 4 would result in a column width 4/12 (or 1/3) of the total width of the containing row.
width	int	The width of the column in the containing row as a percentage, overriding the span attribute. The width is then converted to a span expressed as ((width/100) x 12), rounded to the nearest whole number. For example, a width of 33 would be converted to 3.96, which would be rounded up to a span value of 4.

Example JSP

Below is an example layout created in a portlet:

```
<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>

<aui:container cssClass='super-awesome-container'>
        <aui:row>
                <aui:col md="4" sm="6">
                        <h2>Some fun content using the 'md' and 'sm' attributes</h2>
                </aui:col>

                <aui:col md="8" sm="6">
                        <p>
                                Some text here.
                </p>
            </aui:col>
        </aui:row>

        <aui:row>
            <aui:col width="<%= 40 %>">
                <h2>Some fun content using the 'width' attribute</h2>
            </aui:col>

            <aui:col width="<%= 60 %>">
                <p>
                    Cool text here.
                </p>
            </aui:col>
        </aui:row>

        <aui:row>
            <aui:col span="<%= 4 %>">
                <h2>Some fun content using the 'span' attribute</h2>
            </aui:col>

            <aui:col span="<%= 8 %>">
                <p>
                    Nice text here.
                </p>
            </aui:col>
        </aui:row>
</aui:container>
```

Now you know how to create layouts inside your portlets!

Related Topics

Layout Templates with the Liferay Theme Generator

Chapter 66

Using JavaScript in Your Portlets

Would you like to use the latest ECMAScript features in your JavaScript files and portlets? Do you wish you could use npm and npm packages in your portlets?

In this section of tutorials you'll learn how to prepare your JavaScript files to leverage these features in your portlets.

Chapter 67

Using ES2015 in Your Portlets

You can now write JavaScript that adheres to the new ECMAScript 2015 (ES2015) syntax, leverage ES2015 advanced features in your modules, and publish them. To do these things, you need make only minor adjustments to your JavaScript files and projects.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

This section of tutorials shows how to prepare modules for ES2015 and use them in your portlets.

67.1 Preparing Your JavaScript Files for ES2015

To use the ES2015 syntax in a JavaScript file, add the extension `.es` to its name. For example, you rename file `filename.js` to `filename.es.js`. The extension indicates it uses ES2015 syntax and must therefore be transpiled by Babel before deployment.

ES2015 advanced features, such as generators, are available to you if you import the `polyfillBabel` class from the `polyfill-babel` module found in 7.0:

```
import polyfillBabel from 'polyfill-babel'
```

The Babel Polyfill emulates a complete ES6 environment. Use it at your own discretion, as it loads a large amount of code. You can inspect https://github.com/zloirock/core-js#core-js to see what's polyfilled.

Once you've completed writing your module, you can expose it by creating a `package.json` file that specifies your bundle's name and version. Make sure to create this in your module's root folder. The js-logger module, for example, specifies the following values in its `package.json` file:

```
{
    "name": "js-logger",
    "version": "1.0.0"
}
```

The Module Config Generator creates the module based on this information. There you have it! In just a few steps you can prepare your module to leverage the latest JavaScript standard features and publish it.

Related Topics

Using ES2015 Modules in Your Portlet
OSGi and Modularity for Liferay Portal 6 Developers

67.2 Using ES2015 Modules in your Portlet

Once you've exposed your modules via your `package.json` file, you can use them in your portlets. The `aui:script` tag's `require` attribute makes it easy.

This tutorial covers how to access in your portlets the modules you've exposed. The example module `logger.es` was written inside the Console Logger Portlet. Once the portlet is deployed, and added to a page, you'll notice a printout in the console.

Follow the steps below to use your exposed modules in your portlets.

1. Declare the aui taglib in a JSP in your view:

   ```
    <%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
   ```

   Note: if you created the portlet using Blade, the aui taglib is already provided for you in the `init.jsp`.

2. Add an `aui:script` tag to the JSP and set the `require` attribute to the relative path for your module.

   Since 7.0, the `require` attribute allows you to include your exposed modules in your JSP. The AMD Loader fetches the specified module and its dependencies. For example, the Console Logger Portlet's `view.jsp` includes the module `logger.es`:

   ```
   <aui:script require="js-logger/logger.es">
       var Logger = jsLoggerLoggerEs.default;

       var loggerOne = new Logger('*** -> ');
       loggerOne.log('Hello');

       var loggerDefault = new Logger();
       loggerDefault.log('World');
   </aui:script>
   ```

   References to the module within the script tag are named after the `require` value, in camel-case and with all invalid characters removed. The `logger.es` module's reference `jsLoggerLoggerEs` is derived from the module's relative path value `js-logger/logger.es`. The value is stripped of its dash and slash characters and converted to camel case.

Thanks to the `aui:script` tag and its `require` attribute, using your modules in your portlet is a piece of cake!

Related Topics

Overriding a Module's JSPs
Web Services

Chapter 68

Using npm in Your Portlets

npm is a powerful tool, and almost a necessity for Front-End development. Since Liferay DXP 7.0 Fix Pack 30 and Liferay Portal 7.0 CE GA5, you can use npm as your JavaScript package manager tool—including npm and npm packages—while developing portlets in your normal, everyday workflow.

Deployed portlets leverage Liferay AMD Loader to share JavaScript modules and take advantage of semantic versioning when resolving modules among portlets on the same page. The liferay-npm-bundler helps prepare your npm modules for the Liferay AMD Loader.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

This section of tutorials covers how to set up npm-based portlet projects and how Liferay DXP supports them.

68.1 liferay-npm-bundler

The liferay-npm-bundler is a bundler (like Webpack or Browserify) that targets Liferay DXP as a platform and assumes you're using your npm packages from portlets (as opposed to typical web applications). It is just one of the pieces of the Liferay JS Bundle Toolkit. Liferay JS Bundle Toolkit is an abstract umbrella term that refers to the following tools:

- liferay-npm-bundler
- Babel plugins
- liferay-npm-bundler plugins
- frontend-js-loader-modules-extender
- Javascript AMD loader

Note: The Liferay JS Bundle Toolkit is supported for Liferay DXP 7.0 Fix Pack 39 and up.

The workflow for running npm packages inside portlets is slightly different from standard bundlers. Instead of bundling the JavaScript in a single file, you must *link* all packages together in the browser when the full web page is assembled. This lets portlets share common versions of modules instead of each one loading its own copy. The liferay-npm-bundler handles this for you. You can learn how it works next.

How it Works Internally

The liferay-npm-bundler takes a Liferay DXP portlet project and outputs its files (including npm packages) to a build folder, so the standard portlet build (Gradle) can produce an OSGi bundle. You can learn more about the build folder's structure in The Structure of OSGi Bundles Containing NPM Packages tutorial.

Here's what happens to produce the OSGi bundle:

1. Copy the project's `package.json` file to the output directory.
2. Traverse the project's dependency tree to determine its dependencies.
3. For each npm package dependency:

    a. Copy the npm package to the output folder and prefix the bundle's name to it. Note that the bundler stores packages in plain *bundle-name$package@version* format, rather than the standard node_modules tree format).

    b. Pre-process the npm package with any configured plugins.

    c. Run Babel with configured plugins for each `.js` file inside the npm package.

    d. Post-process the npm package with any configured plugins.

4. For the project:

    a. Pre-process the project's package with any configured plugins.

    b. Run Babel with configured plugins for each `.js` file inside the project.

    c. Post-process the project package with any configured plugins.

The only difference between the pre-process and post-process steps are when they are run (before or after Babel is run, respectively). During this workflow, liferay-npm-bundler calls all the configured plugins so they can perform transformations on the npm packages (for instance, modifying their `package.json` files, or deleting or moving files).

Now you understand how the liferay-npm-bundler works!

Related Topics

The Structure of OSGi Bundles Containing NPM Packages

How Liferay DXP Publishes NPM Packages

Liferay JavaScript APIs

68.2 Adding liferay-npm-bundler to Your Portlet

Adding liferay-npm-bundler to your portlet involves installing the package via npm and adding it to your npm build process.

Installing liferay-npm-bundler

Note: liferay-npm-bundler 1.x performs aggressive semantic version resolution, which can potentially lead to unstable results. To avoid issues, we recommend that you use the latest version of the bundler (2.x at the time of this writing).

Follow these steps to install liferay-npm-bundler:

1. Install NodeJS >= v6.11.0 if you don't have it installed.

2. Navigate to your portlet's `src/main/resources/META-INF/resources` folder.

   If you don't have a portlet already, create an empty MVC portlet. For convenience, you can use Blade CLI to create an empty portlet with the mvc portlet blade template.

3. Run the following command to install the liferay-npm-bundler:

   ```
   npm install --save-dev liferay-npm-bundler
   ```

Note: Use npm from within your portlet project's root folder (where the `package.json` file lives), as you normally do on a typical web project.

Now that you have the liferay-npm-bundler installed, you can add it to your npm build process.

Adding liferay-npm-bundler to Your Build Process

Once your source is transpiled (if necessary) to ECMAScript and converted to AMD format for the Liferay AMD Loader, you must run liferay-npm-bundler in `package.json` to pack the needed npm packages and transform them to AMD. This lets Liferay AMD Loader grab the packages from the Portal.

Add `liferay-npm-bundler` to your `package.json`'s build script:

```
"scripts": {
     "build": "[... && ] liferay-npm-bundler"
}
```

The `[...&&]` refers to any previous steps you need to perform (for example, transpiling your sources with Babel).

You can use any languages you like as long as they can be transpiled to Ecmascript 5 or higher (the only requirement is that Babel can process it and your browser can execute it). When you deploy your portlet using Gradle, the build script is called as part of the process.

Now you know how to add the liferay-npm-bundler to your portlet!

Related Topics

Configuring liferay-npm-bundler

Understanding How liferay-npm-bundler Formats JavaScript Modules for AMD

68.3 Configuring liferay-npm-bundler

The liferay-npm-bundler is configured via a `.npmbundlerrc` file placed in the portlet project's root folder. You can create a complete configuration manually or extend a configuration preset (via Babel).

This tutorial explains the `.npmbundlerrc` file's structure and shows how the default preset configures the liferay-npm-bundler.

Understanding the .npmbundlerrc File's Structure

The `.npmbundlerrc` file has four possible phase definitions: *copy-process*, *pre-process*, *post-process*, and *babel*. These phase definitions are explained in more detail below:

Copy-Process: Defined with the `copy-plugins` property (only available for dependency packages). Specifies which files should be copied or excluded from each given package.

Pre-Process: Defined with the `plugins` property. Specifies plugins to run before the Babel phase is run.

Babel: Defined with the `.babelrc` definition. Specifies the `.babelrc` file to use when running Babel through the package's `.js` files.

Note: During this phase, Babel transforms package files (for example, to convert them to AMD format, if necessary), but doesn't transpile them. In theory, you could also transpile them by configuring the proper plugins. We recommend transpiling before running the bundler, to avoid mixing both unrelated processes.

Post-Process: Defined with the `post-plugins` property. An alternative to using the *pre-process* phase, this specifies plugins to run after the Babel phase has completed.

Here's an example of a `.npmbundlerrc` configuration:

```
{
    "exclude": {
        "*": [
            "test/**/*"
        ],
        "some-package-name": [
            "test/**/*",
            "bin/**/*"
        ],
        "another-package-name@1.0.10": [
            "test/**/*",
            "bin/**/*",
            "lib/extras-1.0.10.js"
        ]
    },
    "include-dependencies": [
        "isobject", "isarray"
    ],
    "output": "build",
    "process-serially": false,
    "verbose": false,
    "dump-report": true,
    "config": {
        "imports": {
            "npm-angular5-provider": {
                "@angular/common": "^5.0.0",
                "@angular/core": "^5.0.0"
            }
        }
    },
    "/": {
        "plugins": ["resolve-linked-dependencies"],
        ".babelrc": {
            "presets": ["liferay-standard"]
        },
        "post-plugins": [
            "namespace-packages",
            "inject-imports-dependencies"
        ]
    },
    "*": {
        "copy-plugins": ["exclude-imports"],
        "plugins": ["replace-browser-modules"],
        ".babelrc": {
            "presets": ["liferay-standard"]
        },
        "post-plugins": [
            "namespace-packages",
            "inject-imports-dependencies",
            "inject-peer-dependencies"
        ]
    }
    "packages": {
        "a-package-name": [
        "copy-plugins": ["exclude-imports"],
            "plugins": ["replace-browser-modules"],
            ".babelrc": {
                "presets": ["liferay-standard"]
            },
            "post-plugins": [
                "namespace-packages",
                "inject-imports-dependencies",
                "inject-peer-dependencies"
            ]
        ],
        "other-package-name@1.0.10": [
          "copy-plugins": ["exclude-imports"],
            "plugins": ["replace-browser-modules"],
            ".babelrc": {
                "presets": ["liferay-standard"]
            },
            "post-plugins": [
                "namespace-packages",
                "inject-imports-dependencies",
                "inject-peer-dependencies"
            ]
        ]
    }
```

Note: Not all definition formats (*, `some-package-name`, and `some-package-name@version`) shown above are required. In most cases, the wildcard definition (*) is enough. The non-wildcard formats (`some-package-name` and `some-package-name@version`) are rare exceptions for packages that require a more specific configuration than the wildcard definition provides.

Below are the configuration options for the `.npmbundlerrc` file:

exclude: defines glob expressions of files to exclude from bundling from all or specific packages.

include-dependencies: defines packages to include in bundling, even if they are not listed under the `dependencies` section of `package.json`. These packages must be available in the `node_modules` folder (i.e. installed manually, without saving them to `package.json`, or listed in the `devDependencies` section).

output: by default the bundler writes packages to the standard Gradle resources folder: `build/resources/main/META-INF/resources`. Set this value to override the default output folder.

process-serially: Process packages in parallel, leveraging Node.js asynchronous model, or one by one. The default value is `false`, (parallel), but if you get EMFILE errors, you can disable this.

verbose: Sets whether to output detailed information about what the tool is doing to the commandline.

dump-report: Sets whether to generate a debugging report. If `true`, an HTML file is generated in the project directory with information such as what the liferay-npm-bundler is doing with each package.

config: global configuration which is passed to all bundler and Babel plugins. Please refer to each plugin's documentation to find the available options for each specific plugin.

"/": plugins' configuration for the project's package.

"": plugins' configuration for dependency packages.

Note: Plugins' configuration specifies the options for configuring plugins in all the possible phases, as well as the `.babelrc` file to use when running Babel (see Babel's documentation for more information on that file format).

Note: Prior to version 1.4.0 of the liferay-npm-bundler, package configurations were placed next to the tools options (*, `output`, `exclude`, etc.) To prevent package name collisions, package configurations are now namespaced and placed under the `packages` section. To maintain backwards compatibility, the liferay-npm-bundler falls back to the root section outside `packages` for package configuration, if no package configurations (`package-name@version`, `package-name`, or *) are found in the `packages` section.

Now that you know the structure of the `.npmbundlerrc` file, you can learn about the default configuration preset.

How the Default Preset Configures the liferay-npm-bundler

The liferay-npm-bundler comes with a default configuration preset: `liferay-npm-bundler-preset-standard` in your `.npmbundlerrc` file. This preset configures several plugins for the build process and is automatically used (even if the `.npmbundlerrc` is missing), unless you override it with one of your own. Running the liferay-npm-bundler with this preset applies the config file from `liferay-npm-bundler-preset-standard`:

```
{
    "/": {
        "plugins": ["resolve-linked-dependencies"],
        ".babelrc": {
            "presets": ["liferay-standard"]
        },
        "post-plugins": ["namespace-packages", "inject-imports-dependencies"]
    },
    "*": {
        "copy-plugins": ["exclude-imports"],
        "plugins": ["replace-browser-modules"],
        ".babelrc": {
            "presets": ["liferay-standard"]
        },
        "post-plugins": [
            "namespace-packages",
            "inject-imports-dependencies",
            "inject-peer-dependencies"
        ]
    }
}
```

Note: You can override configuration preset values by adding your own configuration to your project's `.npmbundlerrc` file. For instance, using the configuration preset example above, you can define your own `.babelrc` value in `.npmbundlerrc` file to override the defined "liferay-standard" babelrc preset.

The `liferay-standard` preset applies the following plugins to packages:

- exclude-imports: Exclude packages declared in the `imports` section from the build.
- inject-imports-dependencies: Inject dependencies declared in the `imports` section in the dependencies' `package.json` files.
- inject-peer-dependencies: Inject declared peer dependencies (as they are resolved in the project's `node_modules` folder) in the dependencies' `package.json` files.
- namespace-packages: Namespace package names based on the root project's package name to isolate packages per project and avoid collisions. This prepends `<project-package-name>$` to each package name appearance in `package.json` files.
- replace-browser-modules: Replace modules listed under `browser/unpkg/jsdelivr` section of `package.json` files.
- resolve-linked-dependencies: Replace linked dependencies versions appearing in `package.json` files (those obtained from local file system or GitHub, for example) by their real version number, as resolved in the project's `node_modules` directory.

In addition, the bundler runs Babel with the babel-preset-liferay-standard preset, that invokes the following plugins:

- babel-plugin-name-amd-modules: Name AMD modules based on package name, version, and module path.
- babel-plugin-namespace-amd-define: Add a prefix to AMD `define()` calls (by default `Liferay.Loader.`).
- babel-plugin-namespace-modules: Namespace modules based on the root project's package name, prepending `<project-package-name>$`. Wrap modules inside an AMD `define()` module for each module name appearance (in `define()` or `require()` calls) so that the packages are localized per project and don't clash.
- babel-plugin-normalize-requires: Normalize AMD `require()` calls.
- babel-plugin-wrap-modules-amd: Wrap modules inside an AMD `define()` module.
- babel-plugin-transform-node-env-inline: Inline the `NODE_ENV` environment variable, and if it's part of a binary expression (eg. `process.env.NODE_ENV === "development"`), then statically evaluate and replace it.

Now you know how to configure the liferay-npm-bundler!

Related Topics

Adding liferay-npm-bundler to Your Portlet

The Structure of OSGi Bundles Containing NPM Packages

68.4 The Structure of OSGi Bundles Containing npm Packages

To deploy JavaScript modules, you must create an OSGi bundle with the npm dependencies extracted from the project's `node_modules` folder and modify them to work with the Liferay AMD Loader. The liferay-npm-bundler automates this process for you, creating a bundle similar to the one below:

- `my-bundle/`
  - `META-INF/`
    - `resources/`
      - `package.json`
      - name: my-bundle-package
      - version: 1.0.0
      - main: lib/index
      - dependencies:
      - my-bundle-package$isarray: 2.0.0
      - my-bundle-package$isobject: 2.1.0

      - ...

      - `lib/`
      - `index.js`
      - ...

      - ...
      - `node_modules/`
      - `my-bundle-package$isobject@2.1.0/`
      - `package.json`
      - name: my-bundle-package$isobject
      - version: 2.1.0
      - main: lib/index
      - dependencies:
      - my-bundle-package$isarray: 1.0.0

      - ...

      - ...

      - `my-bundle-package$isarray@1.0.0/`
      - `package.json`
      - name: my-bundle-package$isarray
      - version: 1.0.0
      - ...

      - ...

      - `my-bundle-package$isarray@2.0.0/`
      - `package.json`

            · name: my-bundle-package$isarray
            · version: 2.0.0
            · ...

            · ...

The packages inside `node_modules` are the same format as the npm tool and can be copied (after a little processing for things like converting to AMD, for example) from a standard `node_modules` folder. The `node_modules` folder can hold any number of npm packages (even different versions of the same package), or no npm packages at all.

Now that you know the structure for OSGi bundles containing npm packages, you can learn how the liferay-npm-bundler handles inline JavaScript packages.

Inline JavaScript packages

The resulting OSGi bundle that the liferay-npm-bundler creates lets you deploy one inline JavaScript package (named `my-bundle-package` in the example) with several npm packages that are placed inside the `node_modules` folder, one package per folder.

The inline package is nested in the OSGi standard `META-INF/resources` folder and is defined by a standard npm `package.json` file.

The inline package is optional, but only one inline package is allowed per OSGi bundle. The inline package usually provides the JavaScript code for a portlet, when the OSGi bundle contains one. Note that the architecture does not differentiate between inline and npm packages once they are published. The inline package is only used for organizational purposes.

Now you know the liferay-npm-bundler creates OSGi bundles for npm packages!

Related Topics

Configuring liferay-npm-bundler

liferay-npm-bundler

Adding liferay-npm-bundler to Your Portlet

68.5 Understanding How liferay-npm-bundler Formats JavaScript Modules for AMD

Liferay AMD Loader is based on the AMD specification. All modules inside an npm OSGi bundle must be in AMD format. This is done for CommonJS modules by wrapping the module code inside a `define` call. The liferay-npm-bundler helps automate this process by wrapping the module for you. This tutorial references the OSGi structure below as an example. You can learn more about this structure in The Structure of OSGi Bundles Containing NPM Packages tutorial.

- `my-bundle/`
    - `META-INF/`
        * `resources/`
            · `package.json`
            · name: my-bundle-package

- version: 1.0.0
- main: lib/index
- dependencies:
- my-bundle-package$isarray: 2.0.0
- my-bundle-package$isobject: 2.1.0

- ...

- `lib/`
- `index.js`
- ...

- ...
- `node_modules/`
- `my-bundle-package$isobject@2.1.0/`
- `package.json`
- name: my-bundle-package$isobject
- version: 2.1.0
- main: lib/index
- dependencies:
- my-bundle-package$isarray: 1.0.0

- ...

- ...

- `my-bundle-package$isarray@1.0.0/`
- `package.json`
- name: my-bundle-package$isarray
- version: 1.0.0
- ...

- ...

- `my-bundle-package$isarray@2.0.0/`
- `package.json`
- name: my-bundle-package$isarray
- version: 2.0.0
- ...

- ...

For example, the `my-bundle-package$isobject@2.1.0` package's `index.js` file contains the following code:

```
'use strict';

var isArray = require('my-bundle-package$isarray');

module.exports = function isObject(val) {
    return val ≠ null && typeof val === 'object' && isArray(val) === false;
};
```

The updated module code configured for AMD format is shown below:

```
define(
    'my-bundle-package$isobject@2.1.0/index',
    ['module', 'require', 'my-bundle-package$isarray'],
    function (module, require) {
        'use strict';

        var define = undefined;

        var isArray = require('my-bundle-package$isarray');

        module.exports = function isObject(val) {
            return val != null && typeof val === 'object'
            && isArray(val) === false;
        };
    }
);
```

Note: The module's name must be based on its package, version, and file path (for example `my-bundle-package$isobject@2.1.0/index`), otherwise Liferay AMD Loader can't find it.

Note the module's dependencies: `['module', 'require', 'my-bundle-package$isarray']`.

`module` and `require` must be used to get a reference to the `module.exports` object and the local `require` function, as defined in the AMD specification.

The subsequent dependencies state the modules on which this module depends. Note that `my-bundle-package$isarray` in the example is not a package but rather an alias of the `my-bundle-package$isarray` package's main module (thus, it is equivalent to `my-bundle-package$isarray/index`).

Also note that there is enough information in the `package.json` files to know that `my-bundle-package$isarray` refers to `my-bundle-package$isarray/index`, but also that it must be resolved to version `1.0.0` of such package, i.e., that `my-bundle-package$isarray/index` in this case refers to `my-bundle-package$isarray@1.0.0/index`.

You may also have noted the `var define = undefined;` addition to the top of the file. This is introduced by `liferay-npm-bundler` to make the module think that it is inside a CommonJS environment (instead of an AMD one). This is because some npm packages are written in UMD format and, because we are wrapping it inside our AMD `define()` call, we don't want them to execute their own `define()` but prefer them to take the CommonJS path, where the exports are done through the `module.exports` global.

You can leverage liferay-npm-bundler with the correct presets to process your npm modules for AMD. All liferay-npm-bundler presets (*liferay-npm-bundler-preset-*) found in the liferay-npm-build-tools repository include some or all of the following Babel plugins to accomplish the AMD conversion:

- babel-plugin-wrap-modules-amd
- babel-plugin-name-amd-modules
- Babel-plugin-namespace-amd-define

Now you have a better understanding of how liferay-npm-bundler formats JavaScript modules for AMD!

Related Topics

How Liferay DXP Publishes NPM Packages

Configuring liferay-npm-bundler

68.6 How Liferay DXP Publishes npm Packages

When you deploy an OSGi bundle with the specified structure, as explained in The Structure of OSGi Bundles Containing NPM Packages tutorial, its modules are made available for consumption through canonical URLs. To better illustrate resolved modules, the example structure below is the standard structure that the liferay-npm-bundler 1.x generates, and therefore doesn't have the namespaced packages that the 2.x version generates. Please refer to the last sections of this tutorial to know how liferay-npm-bundler 2.0 overrides this de-duplication mechanism to implement isolated dependencies and imports.

- `my-bundle/`
  - `META-INF/`
    - `resources/`
      - `package.json`
      - name: my-bundle-package
      - version: 1.0.0
      - main: lib/index
      - dependencies:
      - isarray: 2.0.0
      - isobject: 2.1.0

      - …

      - `lib/`
      - `index.js`
      - …

      - …
      - `node_modules/`
      - `isobject@2.1.0/`
      - `package.json`
      - name: isobject
      - version: 2.1.0
      - main: lib/index
      - dependencies:
      - isarray: 1.0.0

      - …

      - …

      - `isarray@1.0.0/`
      - `package.json`
      - name: isarray
      - version: 1.0.0
      - …

      - …

- `isarray@2.0.0/`
- `package.json`
- name: isarray
- version: 2.0.0
- …

- …

If you deploy the example OSGi bundle shown above, the following URLs are made available (one for each module):

- http://localhost/o/js/module/598/my-bundle-package@1.0.0/lib/index.js
- http://localhost/o/js/module/598/isobject@2.1.0/index.js
- http://localhost/o/js/module/598/isarray@1.0.0/index.js
- http://localhost/o/js/module/598/isarray@2.0.0/index.js

NOTE: The OSGi bundle ID (598) may vary.

You can learn about package de-duplication next.

Package De-duplication

Since two or more OSGi modules may export multiple copies of the same package and version, Liferay Portal must de-duplicate such collisions. To accomplish de-duplication, a new concept called resolved module was created.

A resolved module is the reference package exported to Liferay Portal's front-end, when multiple copies of the same package and version exist. It's randomly referenced from one of the several bundles exporting the same copies of the package.

Using the example from the previous section, for each group of canonical URLs referring to the same module inside different OSGi bundles, there's another canonical URL for the resolved module. The example structure has the resolved module URLs shown below:

- http://localhost/o/js/resolved-module/my-bundle-package@1.0.0/lib/index.js
- [http://localhost/o/js/resolved-module/my-bundle-package$isobject@2.1.0/index.js](http: //localhost/o/js/resolved - module/my - bundle - package$isobject@2.1.0/index.js)
- [http://localhost/o/js/resolved-module/my-bundle-package$isarray@1.0.0/index.js](http: //localhost/o/js/resolved - module/my - bundle - package$isarray@1.0.0/index.js)
- [http://localhost/o/js/resolved-module/my-bundle-package$isarray@2.0.0/index.js](http: //localhost/o/js/resolved - module/my - bundle - package$isarray@2.0.0/index.js)

NOTE: The OSGi bundle ID (598 in the example) is removed and module is replaced by `resolved-module`.

Next you can learn how the bundler (since version 2.0.0) isolates package dependencies. See What Changed Between liferay-npm-bundler 1.x and 2.x for more information on why this change was made.

Isolated Package Dependencies

A typical OSGi bundle structure generated with liferay-npm-bundler 2.x is shown below:

- `my-bundle/`
  - `META-INF/`
    - `resources/`
      - `package.json`
      - name: my-bundle-package
      - version: 1.0.0
      - main: lib/index
      - dependencies:
      - my-bundle-package$isarray: 2.0.0
      - my-bundle-package$isobject: 2.1.0

      - ...

      - `lib/`
      - `index.js`
      - ...

      - ...
      - `node_modules/`
      - `my-bundle-package$isobject@2.1.0/`
      - `package.json`
      - name: my-bundle-package$isobject
      - version: 2.1.0
      - main: lib/index
      - dependencies:
      - my-bundle-package$isarray: 1.0.0

      - ...

      - ...

      - `my-bundle-package$isarray@1.0.0/`
      - `package.json`
      - name: my-bundle-package$isarray
      - version: 1.0.0
      - ...

      - ...

      - `my-bundle-package$isarray@2.0.0/`
      - `package.json`
      - name: my-bundle-package$isarray
      - version: 2.0.0

- ...
- ...

Note that each package dependency is namespaced with the bundle's name (`my-bundle-package$` in the example structure). This lets each project load its own dependencies and avoid potential collisions with projects that export the same package. For example, consider the two portlet projects below:

```
- `my-portlet`
    - package.json
        - dependencies:
            - a-library 1.0.0
            - a-helper 1.0.0
    - node_modules
        - a-library
            - version: 1.0.0
            - dependencies:
                - a-helper ^1.0.0
        - a-helper
            - version: 1.0.0

- `another-portlet`
    - package.json
        - dependencies:
            - a-library 1.0.0
            - a-helper 1.2.0
    - node_modules
        - a-library
            - version: 1.0.0
            - dependencies:
                - a-helper ^1.0.0
        - a-helper
            - version: 1.2.0
```

In this example, `a-library` depends on `a-helper` at version 1.0.0 or higher (note the caret ^ expression in the dependencies). The bundler implements isolated dependencies by prefixing the name of the bundle to the modules, so that `my-portlet` gets its `a-helper` at 1.0.0, while `another-portlet` gets its `a-helper` at 1.2.0.

The dependencies isolation not only avoids collisions between bundles, but also makes peer dependencies behave deterministically as each portlet gets what it had in its `node_modules` folder when it was developed.

Now that you understand how namespacing modules isolates bundle dependencies, avoiding collisions, you can learn about de-duplication next.

De-duplication through Importing

Isolated dependencies are very useful, but there are times when sharing the same package between modules would be more beneficial. To do this, the liferay-npm-bundler lets you import packages from an external OSGi bundle, instead of using your own. This lets you put shared dependencies in one project and reference them from the rest.

Imagine that you have three portlets that compose the homepage of your site: `my-toolbar`, `my-menu`, and `my-content`. These portlets depend on the fake, but awesome, Wonderful UI Components (WUI) framework. This quite limited framework is composed of only three packages:

1. `component-core`
2. `button`
3. `textfield`

Since the bundler namespaces each dependency package with the portlet's name by default, you would end up with three namespaced copies of the WUI package on the page. This is not what you want. Since they share the same package, instead you can create a fourth bundle that contains the WUI package, and import the WUI package in the three portlets. This results in the structure below:

- `my-toolbar/`
  - `.npmbundlerrc`
    - config:
      - imports:
      - wui-provider:
      - component-core: ^1.0.0
      - button: ^1.0.0
      - textfield: ^1.0.0
- `my-menu/`
  - `.npmbundlerrc`
    - config:
      - imports:
      - wui-provider:
      - component-core: ^1.0.0
      - button: ^1.0.0
      - textfield: ^1.0.0
- `my-content/`
  - `.npmbundlerrc`
    - config:
      - imports:
      - wui-provider:
      - component-core: ^1.0.0
      - button: ^1.0.0
      - textfield: ^1.0.0
- `wui-provider/`
  - `.package.json`
    - name: wui-provider
    - dependencies:
      - component-core: 1.0.0
      - button: 1.0.0

- textfield: 1.0.0

The bundler switches the namespace of certain packages, thus pointing them to an external bundle. Say that you have the following code in `my-toolbar` portlet:

```
var Button = require('button');
```

By default, the bundler 2.x transforms this into the following when not imported from another bundle:

```
var Button = require('my-toolbar$button');
```

But, because `button` is imported from `wui-provider`, it is instead changed to the value below:

```
var Button = require('wui-provider$button');
```

Also, a dependency on `wui-provider$button` at version `^1.0.0` is included in `my-toolbar`'s `package.json` file so that the loader finds the correct version. That's all you need. Once `wui-provider$button` is required at runtime, it jumps to `wui-provider`'s context and loads the subdependencies from there on, even if code is executed from `my-toolbar`. This works because, as you can imagine, `wui-provider`'s modules are namespaced too, and once you load a module from it, it keeps requiring `wui-provider$` prefixed modules all the way down.

Next, you will learn possible strategies for importing.

Strategies When Importing Packages

De-duplication by importing is a powerful tool, but you must design a versioning strategy suitable for you so that you don't run into errors.

First of all, you must decide if you want to declare imported dependencies only in the `.npmbundlerrc` file or in the `package.json` too. Listing an imported dependency in `.npmbundlerrc` is enough, even if it isn't present in your `node_modules` folder because during runtime the loader will find it. Listing an imported dependency in `.npmbundlerrc` is enough, even if it isn't present in your `node_modules` folder, because during runtime the loader finds it. If you have previous experience with dynamic linking support in standard operating systems, think of it as a DLL or shared object.

You may need to install your dependencies in `node_modules` too if you use them for tests, or if they contain types needed to compile (like in Typescript), etc. If that is the case, then you can place them in the `dependencies` or `devDependencies` section of your `package.json`. If you list them under the latter, they are automatically excluded from the output bundle by the liferay-npm-bundler. Otherwise, you need to exclude them in the `.npmbundlerrc` file so they don't redundantly appear in the output.

If you list dependencies both in `package.json` and `.npmbundlerrc`, decide how to keep versions in sync. The best advice is to use the same version constraints in both files, but you may decide not to do so if it is necessary. For example, imagine that you import one of your dependencies from another bundle during runtime to run tests. Say you are using version constraint ^1.5.1. It would be desirable that if you have tested your code with a version >=1.5.1 and <2.0.0 (that's what ^1.5.1 means), you get a compatible version during runtime. Thus, you would declare the dependency with ^1.5.1 in `.npmbundlerrc` too.

However, there are times when you may want to be more lenient, and you may need to get a lower version (1.4.0 for example) during runtime, even if you are developing against ^1.5.1. In that case, you can declare ^1.5.1 in your `package.json` and ^1.0.0 in `.npmbundlerrc`.

In the end, it's up to you to decide how you want to handle your dependencies:

1. `package.json` (While developing)
2. `.npmbundlerrc` (During runtime)

we recommend that you choose a versioning strategy and stick to it, to ensure dependencies are satisfied at runtime.

Now you know how Liferay DXP publishes npm packages!

Related Topics

Understanding How liferay-npm-bundler Formats JavaScript Modules for AMD

Understanding How Liferay DXP Exposes Configuration for Liferay AMD Loader

68.7 Understanding How Liferay DXP Exposes Configuration For Liferay AMD Loader

NOTE: This tutorial is for users who know how Liferay AMD Loader works under the hood. You can learn more about Liferay AMD Loader in the Liferay AMD Module Loader tutorial.

With de-duplication in place, JavaScript modules are made available to Liferay AMD Loader through the configuration returned by the `/o/js_loaded_modules` URL.

The OSGi bundle shown below is used for reference in this tutorial:

- `my-bundle/`
  - `META-INF/`
    - `resources/`
      - `package.json`
      - name: my-bundle-package
      - version: 1.0.0
      - main: lib/index
      - dependencies:
      - isarray: 2.0.0
      - isobject: 2.1.0

      - ...

      - `lib/`
      - `index.js`
      - ...

      - ...
      - `node_modules/`
      - `isobject@2.1.0/`
      - `package.json`
      - name: isobject
      - version: 2.1.0
      - main: lib/index
      - dependencies:
      - isarray: 1.0.0

      - ...

- ...

- isarray@1.0.0/
- package.json
- name: isarray
- version: 1.0.0
- ...

- ...

- isarray@2.0.0/
- package.json
- name: isarray
- version: 2.0.0
- ...

- ...

For example, for the specified structure (shown above), as explained in The Structure of OSGi Bundles Containing npm Packages tutorial, the following configuration is published for Liferay AMD loader to consume:

```
Liferay.PATHS = {
  ...
  'my-bundle-package@1.0.0/lib/index': '/o/js/resolved-module/my-bundle-package@1.0.0/lib/index',
  'isobject@2.1.0/index': '/o/js/resolved-module/isobject@2.1.0/index',
  'isarray@1.0.0/index': '/o/js/resolved-module/isarray@1.0.0/index',
  'isarray@2.0.0/index': '/o/js/resolved-module/isarray@2.0.0/index',
  ...
}
Liferay.MODULES = {
  ...
  "my-bundle-package@1.0.0/lib/index.es": {
    "dependencies": ["exports", "isarray", "isobject"],
    "map": {
      "isarray": "isarray@2.0.0",
      "isobject": "isobject@2.1.0"
    }
  },
  "isobject@2.1.0/index": {
    "dependencies": ["module", "require", "isarray"],
    "map": {
      "isarray": "isarray@1.0.0"
    }
  },
  "isarray@1.0.0/index": {
    "dependencies": ["module", "require"],
    "map": {}
  },
  "isarray@2.0.0/index": {
    "dependencies": ["module", "require"],
    "map": {}
  },
  ...
}
Liferay.MAPS = {
  ...
  'my-bundle-package@1.0.0': { value: 'my-bundle-package@1.0.0/lib/index', exactMatch: true}
  'isobject@2.1.0': { value: 'isobject@2.1.0/index', exactMatch: true},
  'isarray@2.0.0': { value: 'isarray@2.0.0/index', exactMatch: true},
  'isarray@1.0.0': { value: 'isarray@1.0.0/index', exactMatch: true},
  ...
}
```

Note:

- The `Liferay.PATHS` property describes paths to the JavaScript module files.
- The `Liferay.MODULES` property describes the dependency names and versions of each module.
- The `Liferay.MAPS` property describes the aliases of the package's main modules.

Now you know how Liferay DXP exposes configuration for Liferay AMD Loader!

68.8 Related Topics

How Liferay DXP Publishes npm Packages

Understanding How liferay-npm-bundler Formats JavaScript Modules for AMD

Chapter 69

Using the NPMResolver API in Your Portlets

If you're developing an npm-based portlet, your OSGi bundle's `package.json` is a treasure-trove of information. It contains everything that's stored in the npm registry about your bundle: default entry point, dependencies, modules, package names, versions, and more. Since Liferay DXP 7.0 Fix Pack 37 and Liferay Portal 7.0 CE GA6, Liferay DXP's `NPMResolver` APIs expose this information so you can access it in your portlet. If it's defined in the OSGi bundle's `package.json`, you can retrieve the information in your portlet with the `NPMResolver` API. For instance, you can use this API to reference an npm package's static resources (such as CSS files) and even to make your code more maintainable.

To enable the `NPMResolver` in your portlet, use the `@Reference` annotation to inject the `NPMResolver` OSGi component into your portlet's Component class, as shown below:

```
import com.liferay.frontend.js.loader.modules.extender.npm.NPMResolver;

public class MyPortlet extends MVCPortlet {

  @Reference
  private NPMResolver `_npmResolver`;

}
```

Note: Because the `NPMResolver` reference is tied directly to the OSGi bundle's `package.json` file, it can only be used to retrieve npm module and package information from that file. You can't use the `NPMResolver` to retrieve npm package information for other OSGi bundles.

Now that the `NPMResolver` is added to your portlet, the tutorials in this section describe retrieving your OSGi bundle's npm package and module information.

69.1 Referencing an npm Module's Package to Improve Code Maintenance

Once you've exposed your modules, you can use them in your portlet via the `aui:script` tag's `require` attribute. By default, Liferay DXP automatically composes an npm module's JavaScript variable based on its name. For example, the module `my-package@1.0.0` translates to the variable `myPackage100` for the `<aui:script>` tag's `require` attribute. This means that each time a new version of the OSGi bundle's npm package is released, you must update your code's variable to reflect the new version. You can use the JSPackage interface to obtain the module's package name and create an alias to reference it, so the variable name always reflects the latest version number!

Follow these steps:

1. Retrieve a reference to the OSGi bundle's npm package using the `getJSPackage()` method:

```
JSPackage jsPackage = _npmResolver.getJSPackage();
```

2. Grab the npm package's resolved ID (the current package version, in the format `<package name>@<version>`, defined in the OSGi module's `package.json`) using the `getResolvedId()` method and alias it with the `as myVariableName` pattern. The example below retrieves the npm module's resolved ID, sets it to the `bootstrapRequire` variable, and assigns the entire value to the attribute `bootstrapRequire`. This ensures that the package version is always up to date:

```
renderRequest.setAttribute(
  "bootstrapRequire",
  jsPackage.getResolvedId() + " as bootstrapRequire");
```

3. Include the reference to the `NPMResolver`:

```
@Reference
private NPMResolver _npmResolver;
```

4. Resolve the `JSPackage` and `NPMResolver` imports:

```
import com.liferay.frontend.js.loader.modules.extender.npm.JSPackage;
import com.liferay.frontend.js.loader.modules.extender.npm.NPMResolver;
```

5. In the portlet's JSP, retrieve the aliased attribute (`bootstrapRequire` in the example):

```
<%
String bootstrapRequire =
    (String)renderRequest.getAttribute("bootstrapRequire");
%>
```

6. Finally, use the attribute as the `<aui:script>` require attribute's value:

```
<aui:script require="<%= bootstrapRequire %>">
    bootstrapRequire.default();
</aui:script>
```

Below is the full example `*Portlet` class:

```
public class MyPortlet extends MVCPortlet {

    @Override
    public void doView(
            RenderRequest renderRequest, RenderResponse renderResponse)
        throws IOException, PortletException {

        JSPackage jsPackage = _npmResolver.getJSPackage();

        renderRequest.setAttribute(
            "bootstrapRequire",
            jsPackage.getResolvedId() + " as bootstrapRequire");

        super.doView(renderRequest, renderResponse);
    }

    @Reference
    private NPMResolver _npmResolver;

}
```

And here is the corresponding example `view.jsp`:

```
<%
String bootstrapRequire =
  (String)renderRequest.getAttribute("bootstrapRequire");
%>

<aui:script require="<%= bootstrapRequire %>">
  bootstrapRequire.default();
</aui:script>
```

Now you know how to reference an npm module's package!

Related Topics

Obtaining an OSGi bundle's Dependency npm Package Descriptors

liferay-npm-bundler

How Liferay DXP Publishes npm Packages

69.2 Obtaining an OSGi bundle's Dependency npm Package Descriptors

While writing your npm portlet, you may need to reference a dependency package or its modules. For instance, you can retrieve an npm dependency package module's CSS file and use it in your portlet. The `NPMResolver` OSGi component provides two methods for retrieving an OSGi bundle's dependency npm package descriptors: `getDependencyJSPackage()` to retrieve dependency npm packages and `resolveModuleName()` to retrieve dependency npm modules. This tutorial references the `package.json` below to help demonstrate these methods:

```
{
    "dependencies": {
        "react": "15.6.2",
        "react-dom": "15.6.2"
    },
    .
    .
    .
}
```

To obtain an OSGi bundle's npm dependency package, pass the package's name in as the `getDependencyJSPackage()` method's argument. The example below resolves the `react` dependency package:

```
String reactResolvedId = npmResolver.getDependencyJSPackage("react");
```

`reactResolvedId`'s resulting value is `react@15.6.2`.

You can use the `resolveModuleName()` method To obtain a module in an npm dependency package. To do this, pass the module's relative path in as the `resolveModuleName()` method's argument. The example below resolves a module named `react-with-addons` for the `react` dependency package:

```
String resolvedModule =
npmResolver.resolveModuleName("react/dist/react-with-addons");
```

The `resolvedModule` variable evaluates to `react@15.6.2/dist/react-with-addons`. You can also use this to reference static resources inside npm packages (like CSS or image files), as shown in the example below:

```
String cssPath = npmResolver.resolveModuleName(
      "react/lib/css/main.css");
```

Now you know how to obtain an OSGi bundle's dependency npm packages descriptors!

Related Topics

Obtaining an OSGi bundle's npm Package Descriptors
The Structure of OSGi Bundles Containing npm Packages
How Liferay DXP Publishes npm Packages

Chapter 70

Applying Lexicon Styles to your App

It's important to have a consistent user experience across your apps. Liferay DXP's built-in apps achieve this through Liferay's Lexicon Experience Language and its web implementation, Lexicon.

Lexicon provides a consistent, user-friendly UI for Liferay DXP apps, and is included in all themes that are based on the `_styled` base theme, making all the components documented on the Lexicon site accessible.

This means you can use Lexicon markup and components in your Liferay DXP apps. These tutorials explain how to apply Lexicon's design patterns to achieve the same look and feel as Liferay DXP's built-in apps.

The tutorials in this section cover the following topics:

- Configuring your portlet title and back link
- Applying Lexicon patterns to your forms, navigation, and more
- Using the Add Button pattern
- Implementing the Management Bar
- Configuring your admin app's actions menu
- Setting search container animations
- Using Lexicon icons in your app

70.1 Configuring Your Application's Title and Back Link

For 7.0 administration applications, the title should be moved to the inner views of the app and the associated back link should be moved to the portlet titles.

If you open the Blogs Admin application in the Control Panel and add a new blog entry, you'll see this behavior in action:

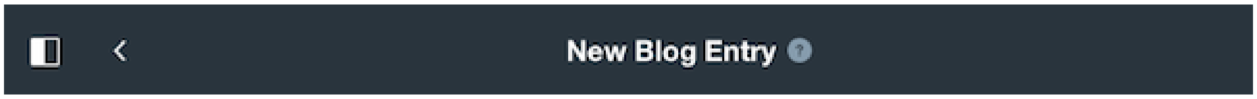


Figure 70.1: Adding a new blog entry displays the portlet title at the top, along with a back link.

This tutorial uses the Blogs Admin application's `edit_entry.jsp` as an example.

Follow these steps to configure your app's title and back URL:

1. Use `ParamUtil` to retrieve the redirect for the URL:

   ```
   String redirect = ParamUtil.getString(request, "redirect");
   ```

2. Display the back icon and set the back URL to the `redirect`:

   ```
   portletDisplay.setShowBackIcon(true);
   portletDisplay.setURLBack(redirect);
   ```

3. Finally, set the title using the `renderResponse.setTitle()` method, as shown in the example configuration below:

   ```
   renderResponse.setTitle((entry ≠ null) ? entry.getTitle() :
   LanguageUtil.get(request, "new-blog-entry"));
   %>
   ```

The example above provides a title for two scenarios:

- If an existing blog entry is being updated, the blog's title is displayed.
- Otherwise it defaults to *New Blog Entry* since a new blog entry is being created.

You should also update any back links in the view to use the `redirect`. For example the `edit_entry.jsp` form's cancel button redirects the user:

```
<aui:button cssClass="btn-lg" href="<%= redirect %>" name="cancelButton"
type="cancel" />
```

Now you know how to configure your app's title and back URL!

Related topics

Applying Lexicon Patterns to your Forms, Navigation, and Search
Setting Search Container Animations

70.2 Applying Lexicon Patterns to Your Forms, Navigation, and Search

This tutorial covers how to leverage Lexicon patterns in your app's forms, navigation, and search results to make them more user-friendly.

You can learn how to update your navigation next.

Applying Lexicon to the Navigation Bar

All administration apps in 7.0 have a navigation bar. Applying Lexicon to your existing navigation bar takes only one additional attribute.

If your app already has a navigation bar implemented with the `aui:nav-bar` tag, you can reuse it by adding the attribute `markupView="lexicon"`.

For example, Liferay's Trash app has the configuration below:

```
<aui:nav-bar cssClass="collapse-basic-search" markupView="lexicon">
```

Note: The `markupView="lexicon"` attribute ensures that the Lexicon markup is used for the UI components, rather than the standard markup. This attribute tells the app to use the `lexicon` folder in the taglib to render the HTML, rather than the default rendering. For example, `<aui:fieldset markupView="lexicon" />` renders the HTML using `/portal/portal-web/docroot/html/taglib/aui/fieldset/lexicon/` instead of the `end.jsp` and `start.jsp` files in `/portal/portal-web/docroot/html/taglib/aui/fieldset/`.

Alternatively, you can use non-bordered tabs with the `liferay-ui:tabs` taglib as the Lexicon Guidelines state.

Sweet! Now you know how to style a navigation bar with Lexicon. Next, you'll learn how to apply Lexicon to your forms.

Applying Lexicon Patterns to the Application Body

To ensure that your application uses all available screen real state from left to right, make the application body fluid in all portlet views. This helps provide a consistent user-experience across all app views.

To make your app's content fluid, add the `container-fluid-1280` class in a `<div>` (or equivalent) element that contains all the portlet's content (excluding the Nav Bar and Management Bar).

If your app's view (or views) are already contained within a `<div>` element, add the `container-fluid-1280` class to it. Otherwise add an uppermost `<div>` element for this purpose:

```
<div class="container-fluid-1280">
...
</div>
```

Next, you can learn how to apply Lexicon to your forms.

Improving your Forms with Lexicon

Follow these steps to apply Lexicon to your forms:

1. Encapsulate your `fieldsets` with the following taglib:

   ```
   <aui:fieldset-group markupView="lexicon">

   </aui:fieldset-group>
   ```

2. The `fieldset` inside `fieldset-group` should be collapsible, so you can hide it when it's not being used. Add the `collapsed` and `collapsible` attributes to your `aui:fieldset` taglib:

```
<aui:fieldset collapsed="<%= true %>" collapsible="<%= true %>"
label="permissions">
      ...
</aui:fieldset>
```

3. Finally, add the `btn-lg` CSS class to your form's buttons to increase the click area:

```
<aui:button-row>

    <aui:button cssClass="btn-lg" type="submit" />

    <aui:button cssClass="btn-lg" href="<%= redirect %>"
    type="cancel" />

</aui:button-row>
```

Your forms are now configured to use Lexicon! Next, you can learn how to apply Lexicon to the Actions menu for your entities.

Applying Lexicon to Your Entity's Actions Menus

Your Actions menus can also benefit from Lexicon patterns. Learn how to apply Lexicon patterns to your Admin app's actions in the Configuring Your Admin app's Actions Menu tutorial. For regular apps, follow these steps:

1. Open your module's actions JSP (`guestbook_actions.jsp` for example) and update the `<liferay-ui:icon-menu>` to use Lexicon's markup with the `markupView` attribute:

```
<liferay-ui:icon-menu
    direction="left-side"
    icon="<%= StringPool.BLANK %>"
    markupView="lexicon"
    message="<%= StringPool.BLANK %>"
    showWhenSingleIcon="<%= true %>"
>
```

2. To follow the Lexicon guidelines, the Actions menu should only display an icon if it is one action. If the Actions menu contains multiple actions, remove the icon's `image` attribute and replace it with the `message` attribute displaying the action's title. Below is an example configuration:

```
<liferay-ui:icon
    message="Edit"
    url="<%= editURL.toString() %>"
/>
```

Next you can update your search iterator.

Applying Lexicon to your Search iterator

To apply Lexicon to your search iterator, add the `markupView="lexicon"` attribute:

```
<liferay-ui:search-iterator

    displayStyle="<%= displayStyle %>"
    markupView="lexicon"
    searchContainer="<%= searchContainer %>"
/>
```

The `displayStyle` attribute specifies which display style is set for the management bar. You can learn how to configure display styles in the Implementing Management Bar Display Styles tutorial.

If the results contain different sets of entries (folders and documents, categories and threads, etc.), you must use a class that implements `*ResultRowSplitter` to divide the results. This is covered next.

Creating a Result Row Splitter

Classes that implement the `*ResultRowSplitter` class divide and categorize the results based on the different entry types. Follow these steps to create a result row splitter:

1. Create a Java class that implements the `ResultRowSplitter` interface. For example, the `com.liferay.bookmarks.web` module has the following `BookmarksResultRowSplitter` class to split its folder and bookmark results:

   public class BookmarksResultRowSplitter implements ResultRowSplitter {

2. Override the `split()` method:

```
    @Override
    public List<ResultRowSplitterEntry> split(List<ResultRow> resultRows) {
        List<ResultRowSplitterEntry> resultRowSplitterEntries =
            new ArrayList<>();
```

3. Create an `ArrayList` for each type of entity, as shown in the example below:

```
        List<ResultRow> entryResultRows = new ArrayList<>();
        List<ResultRow> folderResultRows = new ArrayList<>();
```

4. Loop through the results and add your entities to the proper `ArrayList`:

```
        for (ResultRow resultRow : resultRows) {
            Object object = resultRow.getObject();

            if (object instanceof BookmarksFolder) {
                folderResultRows.add(resultRow);
            }
            else {
                entryResultRows.add(resultRow);
            }
    }
```

5. Create a new `ResultRowSplitterEntry` for each entity, passing the name of the entity and the ArrayList:

```
    if (!folderResultRows.isEmpty()) {
        resultRowSplitterEntries.add(
            new ResultRowSplitterEntry("folders", folderResultRows));
    }

    if (!entryResultRows.isEmpty()) {
        resultRowSplitterEntries.add(
            new ResultRowSplitterEntry("bookmarks", entryResultRows));
    }
```

6. Return the List of `resultRowSplitter` Entries.

7. Use the `resultRowSplitter` attribute in your `liferay-ui:search-iterator` taglib to create a new instance of your `*ResultRowSplitter` as shown in the example below:

```
<liferay-ui:search-iterator
    displayStyle="<%= displayStyle %>"
    markupView="lexicon"
    resultRowSplitter="<%= new BookmarksResultRowSplitter() %>"
    searchContainer="<%= bookmarksSearchContainer %>"
/>
```

Now you know how to apply Lexicon patterns to your app's forms, navigation, and search results!

Related topics

Configuring Your Application's Title and Back Link
Using Lexicon Icons in Your App

70.3 Applying the Add Button Pattern

Lexicon's add button pattern is for actions that add entities (for example a new blog entry button): it gives you a clean, minimal UI. You can use it in any of your app's screens. The add button pattern consists of an `add-menu` tag and at least one `add-menu-item` tag.

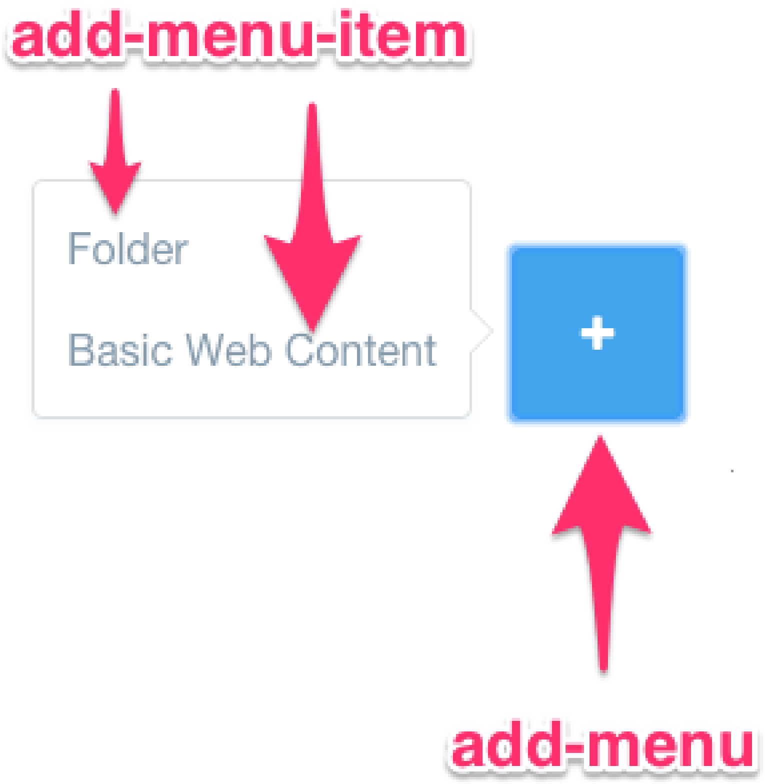


Figure 70.2: The add button pattern consists of an `add-menu` tag and at least one `add-menu-item` tag.

If there's only one item, the plus icon acts as a button that triggers the item. If there's more than one item, clicking the plus icon displays a menu containing them.

Add a `<liferay-frontend:add-menu-item>` tag for every menu item you have. Here's an example of the add button pattern with a single item:

```
<liferay-frontend:add-menu>
    <liferay-frontend:add-menu-item title='<%= LanguageUtil.get(request,
    "titleName") %>' url="<%= nameURL.toString() %>" />
</liferay-frontend:add-menu>
```

You can also find the add button pattern in Liferay DXP's built-in apps. For example, the Message Boards Admin application uses the following add button pattern:

```
<liferay-frontend:add-menu>
    ...
    <liferay-frontend:add-menu-item title='<%= LanguageUtil.get(request,
    "thread") %>' url="<%= addMessageURL.toString() %>" />
    ...
    <liferay-frontend:add-menu-item title='<%= LanguageUtil.get(request,
    (categoryId == MBCategoryConstants.DEFAULT_PARENT_CATEGORY_ID) ?
    "category[message-board]" : "subcategory[message-board]") %>'
    url="<%= addCategoryURL.toString() %>" />
    ...
</liferay-frontend:add-menu>
```

There you have it! Now you know how to use the add button pattern.

Related Topics

Setting Search Container Animations

Adding the Management Bar

Chapter 71

Adding the Management Bar

The Management Bar controls display options for search container results. You can use it to display content in a list or a grid, or to display a specific type of content. You can also customize your app's Management Bar.

The Management Bar is divided into a few key sections. Each section is grouped and configured using different taglibs:

The `<liferay-frontend:management-bar-buttons>` tag wraps the Management Bar's button elements:

The `<liferay-frontend:management-bar-sidenav-toggler-button>` tag implements slide-out navigation for the info button.

The `<liferay-frontend:management-bar-display-buttons>` tag renders the app's display style options:

The `<liferay-frontend:management-bar-filters>` tag wraps the app's filtering options. This filter should be included in all control panel applications. Filtering options can include sort criteria, sort ordering, and more:

Finally, the `<liferay-frontend:management-bar-action-buttons>` tag wraps the actions that you can execute over selected items. In 7.0, you can select multiple items between pages. The management bar keeps track of the number of selected items for you:

For example, here's the Management Bar configuration in Liferay's Trash app:

```
    <liferay-portlet:actionURL name="changeDisplayStyle"
    varImpl="changeDisplayStyleURL">
        <portlet:param name="redirect" value="<%= currentURL %>" />
    </liferay-portlet:actionURL>

    <liferay-frontend:management-bar-display-buttons
        displayViews='<%= new String[] {"descriptive", "icon",
        "list"} %>'
        portletURL="<%= changeDisplayStyleURL %>"
        selectedDisplayStyle="<%= trashDisplayContext.getDisplayStyle()
        %>"
    />
</liferay-frontend:management-bar-buttons>

<liferay-frontend:management-bar-filters>
    <liferay-frontend:management-bar-navigation
        navigationKeys='<%= new String[] {"all"} %>'
        portletURL="<%= trashDisplayContext.getPortletURL() %>"
    />

    <liferay-frontend:management-bar-sort
        orderByCol="<%= trashDisplayContext.getOrderByCol() %>"
        orderByType="<%= trashDisplayContext.getOrderByType() %>"
```

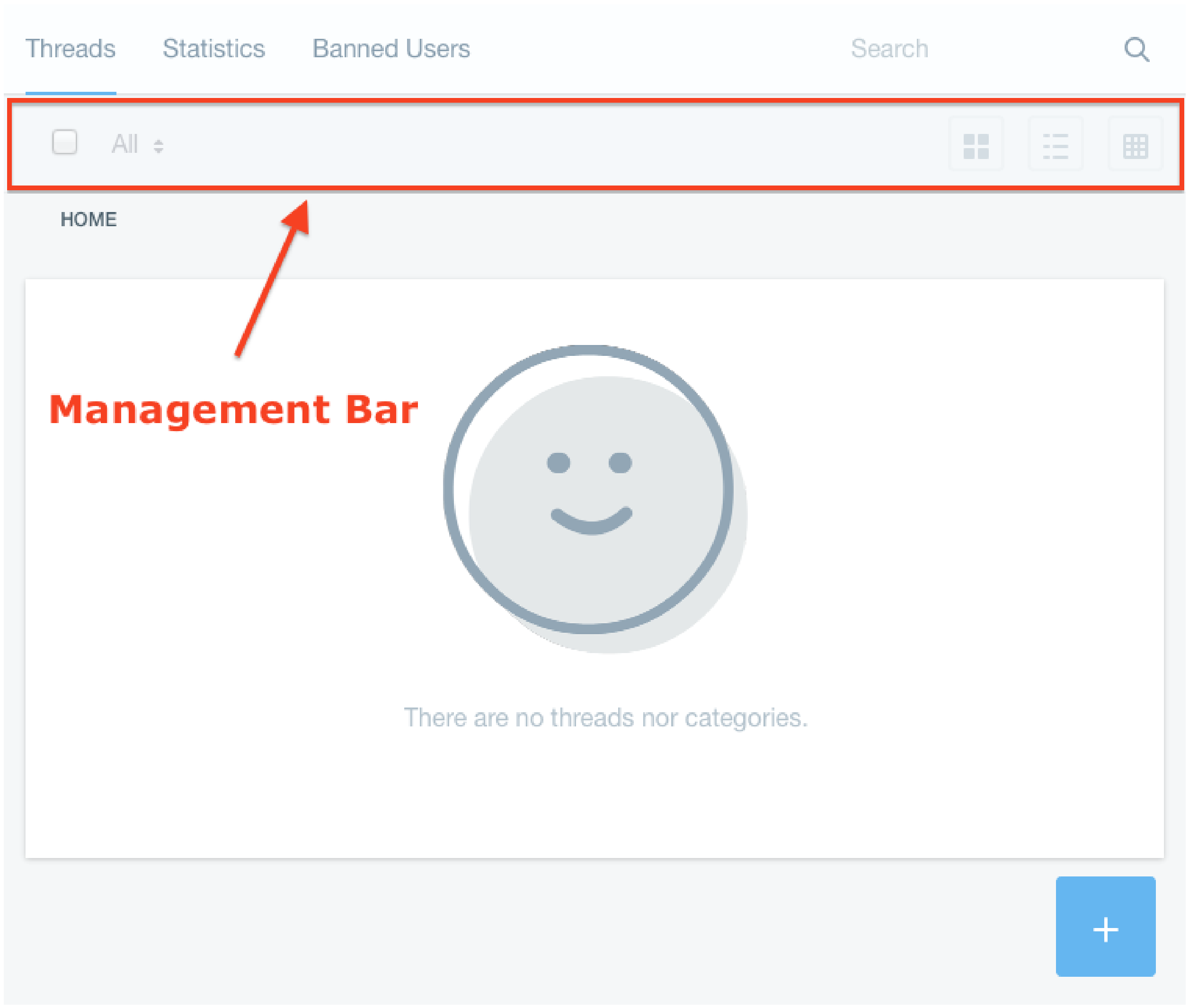


Figure 71.1: The Management Bar lets the user customize how the app displays content.

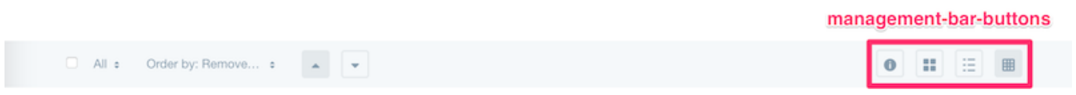


Figure 71.2: The `management-bar-buttons` tag contains the Management Bar's main buttons.

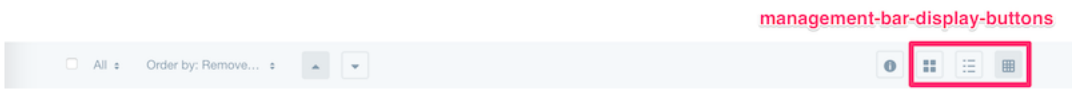


Figure 71.3: The `management-bar-display-buttons` tag contains the content's display options.

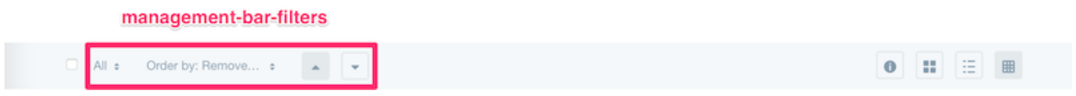


Figure 71.4: The `management-bar-filters` tag contains the content filtering options.

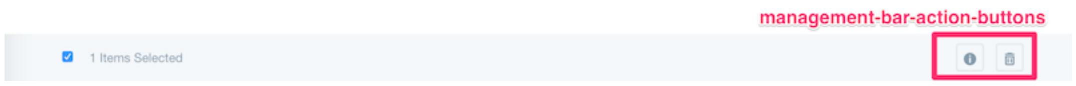


Figure 71.5: The management bar keeps track of the items selected and displays the actions to execute on them.

```
        orderColumns='<%= new String[] {"removed-date"} %>'
        portletURL="<%= trashDisplayContext.getPortletURL() %>"
    />
</liferay-frontend:management-bar-filters>

<liferay-frontend:management-bar-action-buttons>
    <liferay-frontend:management-bar-sidenav-toggler-button />

    <liferay-frontend:management-bar-button href="javascript:;"
    icon="trash" id="deleteSelectedEntries" label="delete" />
</liferay-frontend:management-bar-action-buttons>
```

In this section of tutorials, you'll learn how to add a management bar to your application.

71.1 Implementing the Management Bar Display Styles

The Management Bar offers a few display styles for your app's search container contents: descriptive, icon, and list. These views are standard in Liferay DXP's control panel apps. While you are not required to implement all these display styles in your app's management bar, they provide some additional control over how your app's information is displayed.

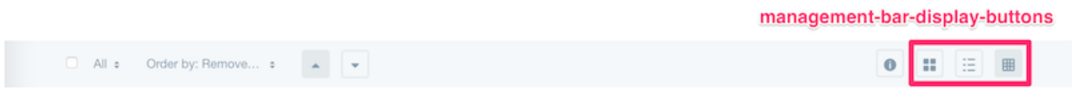


Figure 71.6: The `management-bar-display-buttons` tag contains the content's display options.

To provide these views in your app, you must make some updates to your search result columns. Follow the patterns covered in this tutorial to configure your app.

Note: You are not required to implement all the display views in your app. You must just at least have one display style implemented (`list` is the default). Views that are disabled in your app will render as greyed out buttons.

Start by configuring the Management Bar Display Buttons tag next.

Configuring the Management Bar Display Buttons Tag

Follow these steps to configure the management bar display button tags:

1. Add the `<liferay-frontend:management-bar>` taglib to your app's main view (`view.jsp` for example). If the management bar has a checkbox that needs to stay selected while the navigation is used, you can optionally provide a search container ID with the `searchContainerId` attribute and set the `includeCheckBox` attribute to `true`.

2. Add the management bar buttons using the `<liferay-frontend:management-bar-buttons>` and `<liferay-frontend:management-bar-display-buttons>` tags. The `<liferay-frontend:management-bar-display-buttons>` tag requires three attributes: `displayViews`, the display style views that are available;

`portletURL`, the URL to redirect to after an option is chosen; and `selectedDisplayStyle`, the view to display. Below is an example configuration that implements all three display views:

```
<liferay-frontend:management-bar>
    <liferay-frontend:management-bar-buttons>
      <liferay-frontend:management-bar-display-buttons
        displayViews='<%= new String[] {"icon", "descriptive", "list"} %>'
        portletURL="<%= myViewURL %>"
        selectedDisplayStyle="<%= displayStyle %>"
      />
    </liferay-frontend:management-bar-buttons>
</liferay-frontend:management-bar>
```

Your taglibs are configured for your display styles, but at the moment they don't do anything. You'll configure the views next.

Configuring the Display Views

Note that your management bar may not contain all three views. You only need to implement the views that you defined in your `<liferay-frontend:management-bar-display-buttons>` tag's `displayViews` attribute. Follow these steps to set the display views for the management bar:

1. Define a default display style. For example, the configuration below sets the default display style to list:

```
<%
String displayStyle = ParamUtil.getString(request, "displayStyle", "list");
%>
```

2. Wrap each display style configuration with the proper check:

```
<c:choose>
                <c:when test='<%= Objects.equals(displayStyle, "icon") %>'>
            <%-- icon display style configuration goes here --%>
        </c:when>
        ...
        <c:when test='<%= Objects.equals(displayStyle, "descriptive") %>'>
            <%-- descriptive display style configuration goes here --%>
        </c:when>
        ...
        <c:when test='<%= Objects.equals(displayStyle, "list") %>'>
            <%-- list display style configuration goes here --%>
        </c:when>
</c:choose>
```

Use cards to display the information. Use a vertical card to display assets like files or web content. Use horizontal cards to display folders or directories. You can add the display style configurations for each view next.

Implementing the Icon View

The icon view prominently displays an icon for the content, along with its name, status, and a condensed description.

Follow the steps below to create your icon view:

1. First, make your icon view responsive to different devices.

   For vertical cards use the following pattern:

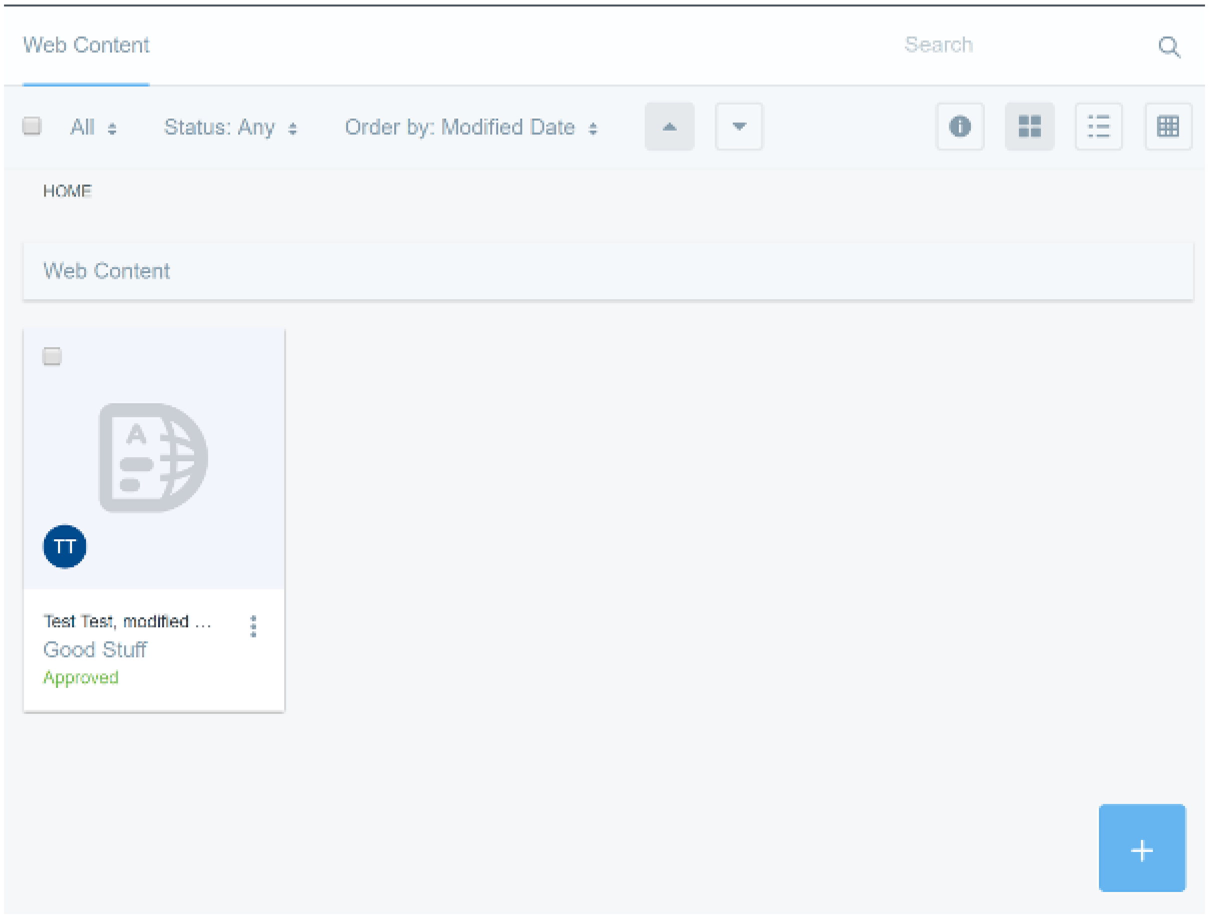


Figure 71.7: The Management Bar's icon display view gives a quick summary of the content's description and status.

```
<%
row.setCssClass("col-md-2 col-sm-4 col-xs-6");
%>
```

For horizontal cards use the following pattern:

```
<%
row.setCssClass("col-md-3 col-sm-4 col-xs-12");
%>
```

2. Once your cards are responsive, you must add search container column text using the pattern below:

```
<liferay-ui:search-container-column-text>
    <%-- include your vertical card or horizontal card here --%>
</liferay-ui:search-container-column-text>
```

3. Add the card to the `<liferay-ui:search-container-column-text>`:

   Use one of the following tags for your vertical card:

```
<liferay-frontend:vertical-card/>
```

```
<liferay-frontend:user-vertical-card/>

<liferay-frontend:icon-vertical-card/>
```

Below is an example from the `com.liferay.journal.web` module's `view_comments.jsp`:

```
<liferay-ui:search-container-column-text>
  <liferay-frontend:vertical-card
    cssClass="entry-display-style"
    imageUrl="<%= (userDisplay ≠ null) ? userDisplay.getPortraitURL(themeDisplay) : UserConstants.getPortraitURL(themeDisplay.getPathImage(), true,
    resultRow="<%= row %>"
  >
    <liferay-frontend:vertical-card-header>
      <liferay-ui:message arguments="<%= new String[] {LanguageUtil.getTimeDescription(locale, System.currentTimeMillis() - mbMessage.getModifiedDat
ago-by-x" translateArguments="<%= false %>" />
    </liferay-frontend:vertical-card-header>

    <liferay-frontend:vertical-card-footer>
      <%= HtmlUtil.extractText(content) %>
    </liferay-frontend:vertical-card-footer>
  </liferay-frontend:vertical-card>
</liferay-ui:search-container-column-text>
```

For horizontal cards you can use the tag below:

```
<liferay-frontend:horizontal-card/>
```

Now that your icon view is configured, you can move onto your descriptive view next.

Implementing the Descriptive View

The descriptive view displays the complete description, along with a small icon for the content, and its name. Your descriptive view should have three columns.

1. The first column usually contains an icon, image, or user portrait:

   For an icon use the following tag:

   ```
   <liferay-ui:search-container-column-icon/>
   ```

   For an image use the following tag:

   ```
   <liferay-ui:search-container-column-image/>
   ```

   For a user portrait use the following pattern:

   ```
   <liferay-ui:search-container-column-text>
       <liferay-ui:user-portrait/>
   </liferay-ui:search-container-column-text>
   ```

2. The second column should contain the descriptions. For example, the site teams application is configured with the settings below:

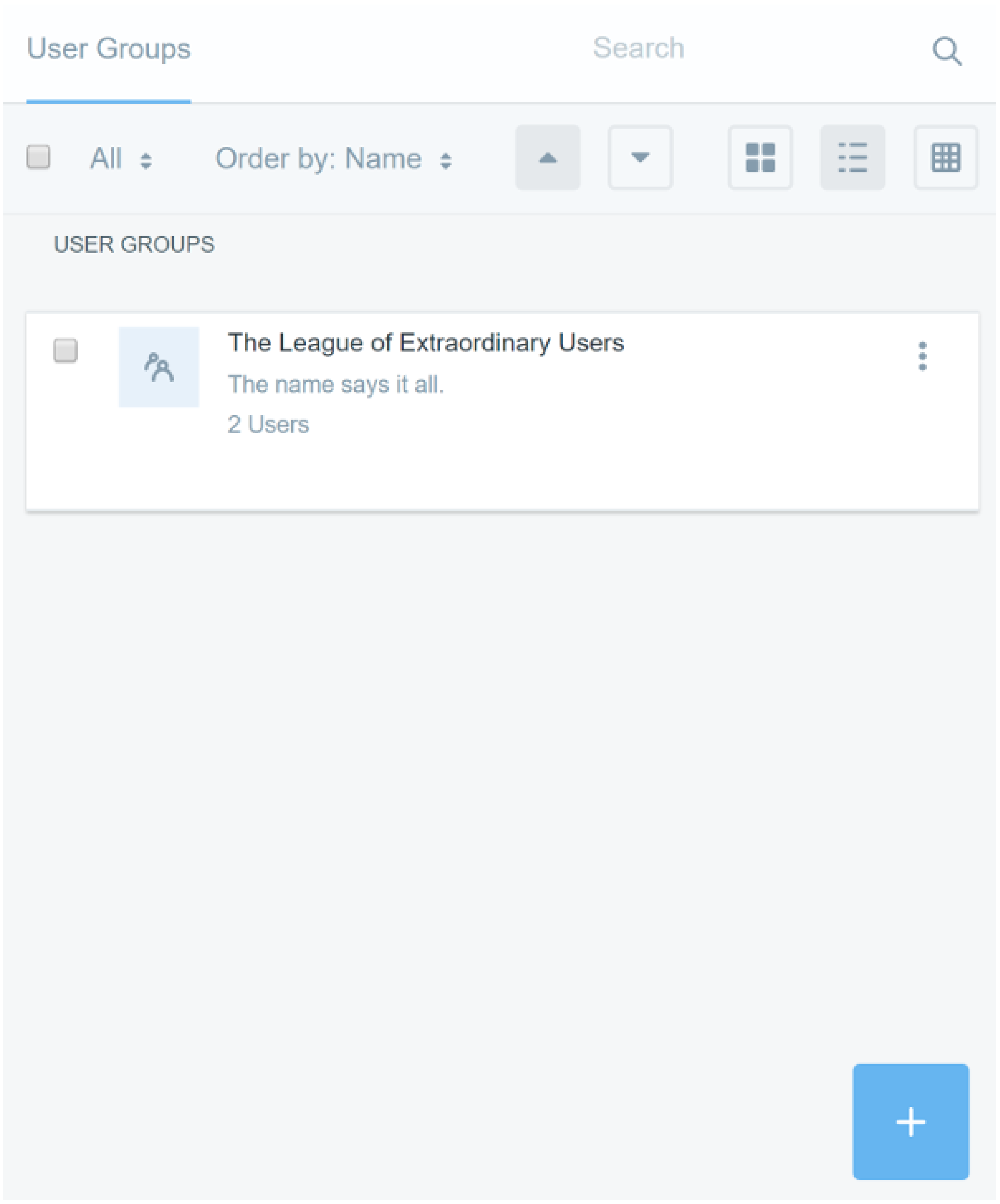


Figure 71.8: The Management Bar's descriptive display view gives the content's full description.

```
<liferay-ui:search-container-column-text
    colspan="<%=2%>"
>
    <h5><%= userGroup.getName() %></h5>

    <h6 class="text-default">
        <span><%= userGroup.getDescription() %></span>
    </h6>

    <h6 class="text-default">
        <span><liferay-ui:message arguments="<%= usersCount%>" key="x-users"
        /></span>
    </h6>

</liferay-ui:search-container-column-text>
```

3. Finally, the third column contains the actions. For example, the site teams application uses the configuration below:

```
<liferay-ui:search-container-column-jsp
    path="/edit_team_assignments_user_groups_action.jsp"
/>
```

Now that your descriptive view is configured you can implement your list view next.

Implementing the List View

The list view is the default view that is shown for most applications. This view lists the content's information in individual columns.

For example, the mobile device rules application configures its list view using the pattern below:

```
<liferay-ui:search-container-column-text
        cssClass="content-column name-column title-column"
        name="name"
        truncate="<%= true %>"
        value="<%= rule.getName(locale) %>"
/>

<liferay-ui:search-container-column-text
        cssClass="content-column description-column"
        name="description"
        truncate="<%= true %>"
        value="<%= rule.getDescription(locale) %>"
/>

<liferay-ui:search-container-column-date
        cssClass="create-date-column text-column"
        name="create-date"
        property="createDate"
/>

<liferay-ui:search-container-column-text
        cssClass="text-column type-column"
        name="type"
        translate="<%= true %>"
        value="<%= rule.getType() %>"
/>

<liferay-ui:search-container-column-jsp
        cssClass="entry-action-column"
        path="/rule_actions.jsp"
/>
```

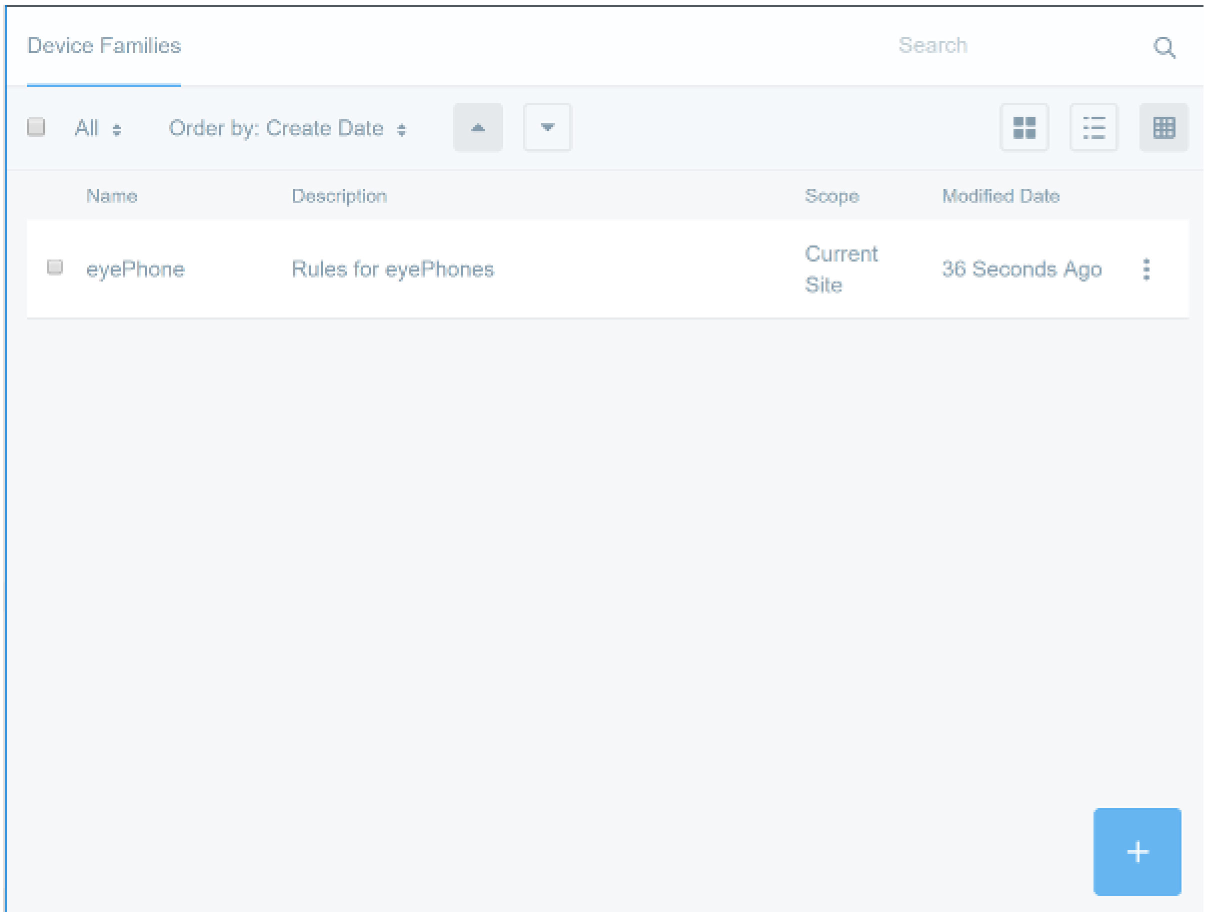


Figure 71.9: The Management Bar's list display view list the content's information in individual columns.

Finally, set the display style in your `liferay-ui:search-iterator` tag with the `displayStyle` attribute:

```
<liferay-ui:search-iterator

    displayStyle="<%= displayStyle %>"
    markupView="lexicon"
    searchContainer="<%= searchContainer %>"
/>
```

The `displayStyle` attribute is set to the `displayStyle` var which is set by the management bar display style buttons.

Your display views are configured!

Related Topics

Implementing a Management Bar Sort Filter

Implementing a Management Bar Navigation Filter

71.2 Implementing a Management Bar Navigation Filter

Navigation filters are used to create navigation menus in the Management Bar. You can add as many navigation filters to the Management Bar as your app requires.

You can learn how to configure the navigation filter next.

Configuring the Navigation Filter

Follow these steps to configure the navigation filter:

1. Add the `<liferay-frontend:management-bar-filters>` tag below the `<liferay-frontend:management-bar-buttons>` tags, to contain your management bar filters.

2. Use the `<liferay-frontend:management-bar-navigation>` tag to add as many navigation menus as your app requires. Use the `navigationKeys` attribute to set the navigation menu options. The `navigationParam` attribute identifies the parameter to use for the navigation filter value. The default value is `navigation`. If you have more than one navigation menu, you can specify a unique variable with the `navigationParam` to identify each menu. Finally, use the `portletURL` attribute to set the URL for the page. Below is an example configuration with one navigation menu:

   ```
   <liferay-frontend:management-bar-filters>

       <liferay-frontend:management-bar-navigation
       navigationKeys='<%= new String[] {"all", ["navigation-title"]...} %>'
       navigationParam="myCustomNavigationVariable"
       portletURL="<%= portletURL %>"
       />
   </liferay-frontend:management-bar-filters>
   ```

   If your app doesn't require any navigation filters, you can just provide the *all* filter to display everything. If, however, you need to let the user navigate between pages (JSPs) of your app , you can add additional strings to the `navigationKeys` attribute for each page you need.

3. Set the navigation filter's default value with `paramUtil`. For example, the configuration below sets the default navigation filter to *all*:

   ```
   String navigation = ParamUtil.getString(request, "navigation", "all");
   ```

4. If your app has multiple options in a navigation menu, use the `navigationParam` to check the current value. Below is an example code snippet from `com.liferay.wiki.web` module's `page_iterator.jsp` that checks the navigation menu's value to render the proper JSP content. Note that it uses the `navigationParam`'s default value `navigation` to check the current value:

   ```
   if (navigation.equals("all-pages")) {
       portletURL.setParameter("mvcRenderCommandName", "/wiki/view_pages");

       PortalUtil.addPortletBreadcrumbEntry(request, LanguageUtil.get(request,
     "all-pages"), portletURL.toString());
   }
   else if (navigation.equals("categorized-pages")) {
       portletURL.setParameter("mvcRenderCommandName",
     "/wiki/view_categorized_pages");

       portletURL.setParameter("categoryId", String.valueOf(categoryId));
   }
   else if (navigation.equals("draft-pages")) {
       portletURL.setParameter("mvcRenderCommandName", "/wiki/view_draft_pages");

       PortalUtil.addPortletBreadcrumbEntry(request, LanguageUtil.get(request,
     "draft-pages"), portletURL.toString());
   }
   ```

Now you know how to add navigation filters to a management bar!

Related Topics

Implementing a Management Bar Sort Filter

Implementing the Management Bar Display Styles

71.3 Implementing a Management Bar Sort Filter

The Management Bar Sort Filters let you compare entries for a search container field, and sort them by ascending or descending. To do this, you must create a comparator class for each field that you want to sort.

The sort filters are an implementation of the standard `Comparator` Interface, with some additional methods provided by the `OrderByComparator` class.

Once the class is created you can use it in your view to add the sort filters to the UI.

Go ahead and get started by creating the Comparator class next.

Creating the Comparator Class

The `OrderByComparator` class is a `Comparator` implementation that you can extend to create sort filters. Follow these steps to create the *`OrderByComparator` class:

1. Right-click on your API module's folder in the package explorer and select *New→Package* to create a new package.

2. Right-click the package you just created and select *New→Class*. Enter *EntryNameComparator* for the class Name, check the *Constructors from superclass* option, and click *Finish*.

3. Update the class declaration to extend the `OrderByComparator` class and use a proper asset type, <Entry> for example>. Below is an example configuration for an entry name field comparator:

```
 public class EntryNameComparator extends OrderByComparator<Entry>{
   ...
 }
```

4. Add variables for the ascending, descending, and column name field (name for example) sorters:

```
public static final String ORDER_BY_ASC = "[asset].[column name] ASC";

public static final String ORDER_BY_DESC = "[asset].[column name] DESC";

public static final String[] ORDER_BY_FIELDS = {"[field name]"};
```

Below is an example configuration for an Entry asset's name field:

```
public static final String ORDER_BY_ASC = "Entry.name ASC";

public static final String ORDER_BY_DESC = "Entry.name DESC";

public static final String[] ORDER_BY_FIELDS = {"name"};
```

5. Replace the public constructor with the following constructors:

```
public *Comparator() {
    this(false);
}

public *Comparator(boolean ascending) {
    _ascending = ascending;
}
```

Below is an example configuration for an EntryNameComparator:

```
public EntryNameComparator() {
  this(false);
}

public EntryNameComparator(boolean ascending) {
  _ascending = ascending;
}
```

6. Add the `compare()` method to compare search container asset entries. Below is an example configuration for entry assets:

```
@Override
public int compare(Entry entry1, Entry entry2) {
  String name1 = entry1.getName();
  String name2 = entry2.getName();

  int value = name1.compareTo(name2);

  if (_ascending) {
    return value;
  }
  else {
    return -value;
  }
}
```

7. Add the following code to return the order fields and check whether the order is ascending or descending:

```
@Override
public String getOrderBy() {
    if (_ascending) {
        return ORDER_BY_ASC;
    }
    else {
        return ORDER_BY_DESC;
    }
}

@Override
public String[] getOrderByFields() {
    return ORDER_BY_FIELDS;
}

@Override
public boolean isAscending() {
    return _ascending;
}

private final boolean _ascending;
```

8. Finally, resolve imports for the class:

```
import com.liferay.docs.guestbook.model.Entry;
import com.liferay.portal.kernel.util.OrderByComparator;
```

Now that your *Comparator class is written you must update the service layer to use it.

Updating the Service Layer

Follow these steps to update services:

1. Open your *EntryLocalServiceImpl class in your service module and import the OrderByComparator class:

```
import com.liferay.portal.kernel.util.OrderByComparator;
```

2. Update the getEntries() method with the start and end integers to include the OrderByComparator parameter. Below is an example configuraiton:

```
public List<Entry> getEntries(
    long groupId, long guestbookId, int start, int end,
    OrderByComparator<Entry> obc) {

    return entryPersistence.findByG_G(
        groupId, guestbookId, start, end, obc);
}
```

3. Rebuild services for your App. Right-click the service module in the Project Explorer and select *Liferay → build-service*.

4. Export the comparator package in the API module's BND.

Now that the services are updated and your exports are in order, you can configure the view to use the comparator next.

Configuring the View

Follow these steps to configure the view to use the Comparator:

1. Import the EntryNameComparator and Comparator classes into the web module project's init.jsp:

```
page import="com.liferay.docs.guestbook.util.comparator.EntryNameComparator"

page import="com.liferay.portal.kernel.util.OrderByComparator"
```

2. Open the view and add the comparator code below the displayStyle variable in the java scriplet at the top. Below is an example configuration that uses the EntryNameComparator class:

```
String orderByCol = ParamUtil.getString(request, "orderByCol", "name");

boolean orderByAsc = false;

String orderByType = ParamUtil.getString(request, "orderByType", "asc");

if (orderByType.equals("asc")) {
    orderByAsc = true;
}

OrderByComparator orderByComparator = null;

if (orderByCol.equals("name")) {
    orderByComparator = new EntryNameComparator(orderByAsc);
}
```

This sets up the configuration for the comparators.

3. Add `orderByCol` and `orderByType` portlet parameters for your order comparator to the view's render URL. The `orderByCol` parameter specifies the column to order by and the `orderByType` column specifies whether the order is ascending or descending. Below is the configuration for the EntryNameComparator:

```
<liferay-portlet:renderURL varImpl="viewPageURL">
    <portlet:param name="mvcPath" value="/html/guestbookmvcportlet/view.jsp" />
    <portlet:param name="guestbookId" value="<%= String.valueOf(guestbookId) %>" />
    <portlet:param name="displayStyle" value="<%= displayStyle %>" />
    <portlet:param name="orderByCol" value="<%= orderByCol %>" />
    <portlet:param name="orderByType" value="<%= orderByType %>" />
</liferay-portlet:renderURL>
```

4. Add the sort filters below the navigation filters, using the `<liferay-frontend:management-bar-sort />` taglib. Pass the name of the column you specified in the *Comparator class. Below is the example configuration for the EntryNameComparator class:

```
<liferay-frontend:management-bar-sort
  orderByCol="<%= orderByCol %>"
  orderByType="<%= orderByType %>"
  orderColumns='<%= new String[] {"name"} %>'
  portletURL="<%= viewPageURL %>"
/>
```

5. Finally, pass the `orderByComparator` in as an argument in the search container results to match the updated method signature you modified. Below is the configuration for the example EntryNameComparator:

```
<liferay-ui:search-container-results
    results="<%= EntryLocalServiceUtil.getEntries(scopeGroupId,
  guestbookId, searchContainer.getStart(), searchContainer.getEnd(),
  orderByComparator) %>"
/>
```

The Management Bar Sort Filters are finished!

Related Topics

Implementing a Management Bar Navigation Filter
Disabling the Management Bar

71.4 Disabling the Management Bar

When there's no content in the app, you should disable all the Management Bar's buttons, except the info button.

You can disable the Management Bar by adding the `disabled` attribute to the `liferay-frontend:management-bar` tag:

```
<liferay-frontend:management-bar
        disabled="<%= total == 0 %>"
        includeCheckBox="<%= true %>"
        searchContainerId="<%= searchContainerId %>"
>
```

You can also disable individual buttons by adding the `disabled` attribute to the corresponding tag. The example configuration below disables the display buttons when the search container displays 0 results:

```
<liferay-frontend:management-bar-display-buttons
        disabled="<%= total == 0 %>"
        displayViews='<%= new String[] {"descriptive", "icon", "list"} %>'
        portletURL="<%= changeDisplayStyleURL %>"
        selectedDisplayStyle="<%= trashDisplayContext.getDisplayStyle() %>"
/>
```

Now you know how to disable the Management Bar!

Related Topics

Implementing a Management Bar Sort Filter

Setting Search Container Animations

71.5 Configuring Your Admin App's Actions Menu

In versions prior to 7.0, it was common to have a series of buttons or menus with actions in the different views of the app. In 7.0 the proposed pattern is to move all of these actions to the upper right menu of the administrative portlet, leaving the primary action (often an "Add" operation) visible in the add menu, using the Add Button pattern. For example, the web content application has the actions menu shown below:

The changes covered in this tutorial do not refer to actions menus associated with entities. For those, see Applying Lexicon Patterns to Your Forms, Navigation, and Search.

This tutorial shows how to configure the actions menu in your admin app. The first step is to create the `PortletConfigurationIconFactory` class.

Creating the Icon Factory Class

To add an action to the upper right menu of the Admin portlet you must first create a `PortletConfigurationIcon` Component class. This class specifies the portlet for the action, the screen to show it on, and the order (by specifying a weight).

In this example, the action appears on the System Settings portlet. To make it appear in a secondary screen, you can use the `path` property as shown below:

```
@Component(
  immediate = true,
  property = {
     "javax.portlet.name=" +
        ConfigurationAdminPortletKeys.SYSTEM_SETTINGS,
```

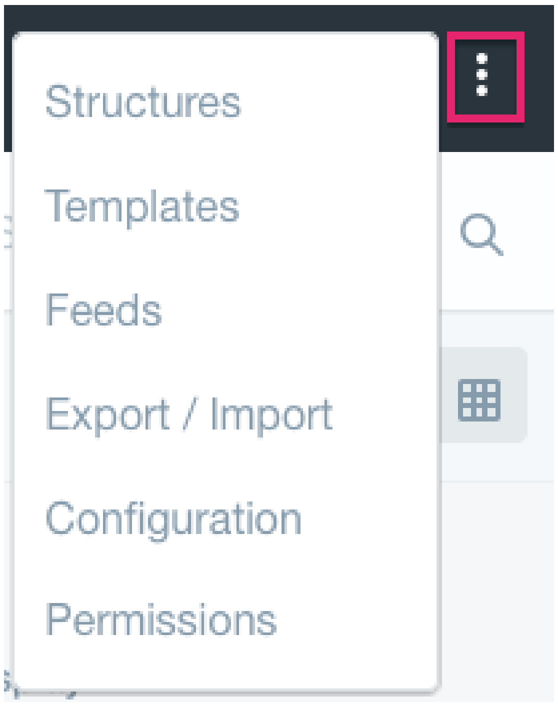


Figure 71.10: The upper right ellipsis menu contains most of the actions for the app.

```
        "path=/view_factory_instances"
    },
    service = PortletConfigurationIconFactory.class
)
public class ExportFactoryInstancesIconFactory
    extends BasePortletConfigurationIconFactory {

    @Override
    public PortletConfigurationIcon create(PortletRequest portletRequest) {
        return new ExportFactoryInstancesIcon(portletRequest);
    }

    @Override
    public double getWeight() {
        return 1;
    }

}
```

The value of the path property depends on the MVC framework used to develop the app.

For the MVCPortlet framework, provide the path (often a JSP) from the `mvcPath` parameter.

For MVCPortlet with MVC Commands, the path should contain the `mvcRenderCommandName` where the actions should be displayed (such as `/document_library/edit_folder` for example).

Now that your `PortletConfigurationIconFactory` class is written, you can write the `PortletConfigurationIcon` class next.

Writing the Configuration Icon Class

The second class that you must write is a class that extends the `BasePortletConfigurationIcon` class. This class specifies the action's label, whether it's invoked with a GET or POST method, and the URL (or `onClick` JavaScript method) to invoke when the action is clicked. It can also implement some custom code to determine whether the action should display for the current request. For example the class below creates a export-all-settings label and specifies the GET method for the action:

```
public class ExportAllConfigurationIcon extends BasePortletConfigurationIcon {

  public ExportAllConfigurationIcon(PortletRequest portletRequest) {
     super(portletRequest);
  }

  @Override
  public String getMessage() {
     return "export-all-settings";
  }

  @Override
  public String getMethod() {
     return "GET";
  }

  @Override
  public String getURL() {
     LiferayPortletURL liferayPortletURL =
        (LiferayPortletURL)PortalUtil.getControlPanelPortletURL(
           portletRequest, ConfigurationAdminPortletKeys.SYSTEM_SETTINGS,
           PortletRequest.RESOURCE_PHASE);

     liferayPortletURL.setResourceID("export");

     return liferayPortletURL.toString();
  }

  @Override
  public boolean isShow() {
     return true;
  }

}
```

By default, if the portlet uses mvcPath, the global actions (such as configuration, export/import, maximized, etc.) are displayed for the JSP indicated in the initialization parameter of the portlet javax.portlet.init-param.view-template=/view.jsp. The value indicates the JSP where the global actions should be displayed.

However, if the portlet uses MVC Command, the views for the global actions must be indicated with the initialization parameter javax.portlet.init-param.mvc-command-names-default-views=/wiki_admin/view and the value must contain the mvcRenderCommandName where the global actions should be displayed.

For portlets that can be added to a page, if the desired behavior is to always include the configuration options, the following initialization parameter should be added to the portlet:

```
javax.portlet.init-param.always-display-default-configuration-icons=true
```

Now you know how to configure your admin app's actions!

Related Topics

Applying Lexicon Patterns to your Forms, Navigation, and Search

Configuring Your Application's Title and Back Link

71.6 Setting Search Container Animations

If you've toured 7.0's UI, you've probably noticed animations in the search containers. These animations show the user when there is no available content.

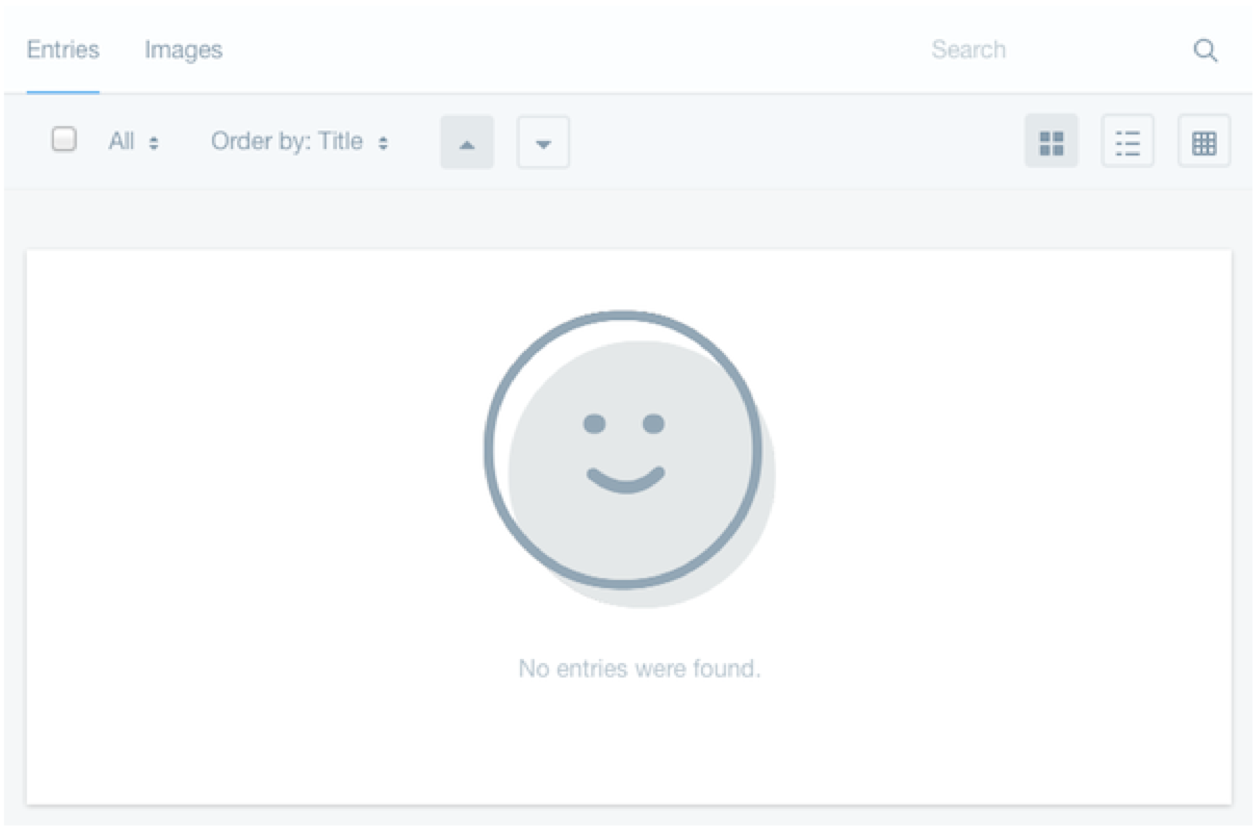


Figure 71.11: This is a still frame from the Blogs portlet's empty results animation.

This tutorial covers the following topics:

- Using the default animations in your search container
- Using custom animations in your search container

First, you'll learn how to use the default animations in your search container.

Using the Default Animations in Your Search Container

There are three built-in classes for the search container animation:

1. The default class is *taglib-empty-result-message-header*. This is used for most cases.
2. *taglib-empty-search-result-message-header* is used when there are no search results.
3. *taglib-empty-result-message-header-has-plus-btn* is used when there is no content, but you can use the add button to add an entity.

To use these animations, use the following method:

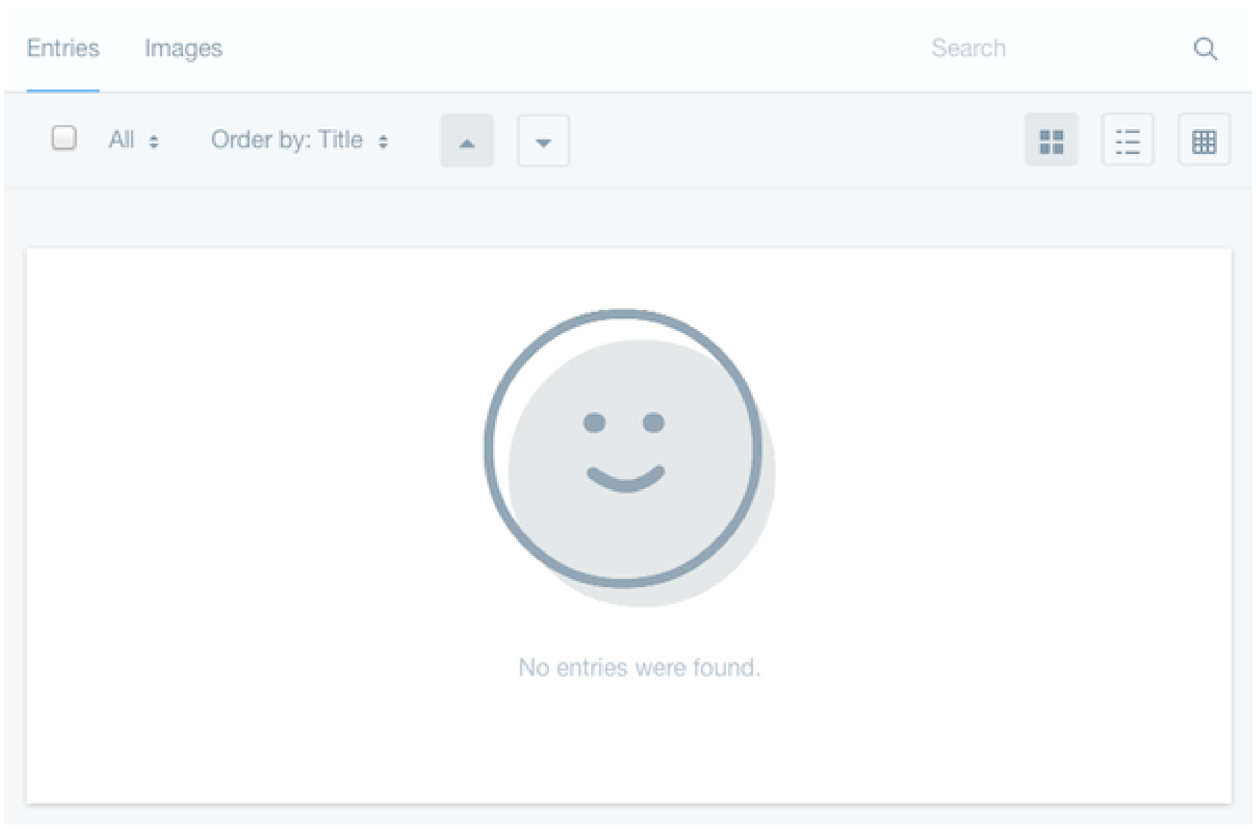


Figure 71.12: When no content is found, the default animation is usually shown.

```
SearchContainer.setEmptyResultsMessageCssClass()
```

For example, the Roles Admin application uses this code to set its animation in its `edit_role_assignments_sites.jsp` file:

```
if (!searchTerms.isSearch()) {
        searchContainer.setEmptyResultsMessageCssClass(
        "taglib-empty-result-message-header-has-plus-btn"
        );
}
```

Alternatively you can use the `emptyResultsMessageCssClass` attribute of the `liferay-ui:search-container` tag to set the animation. For example,

```
<liferay-ui:search-container
  emptyResultsMessage="no-results-were-found"
  emptyResultsMessageCssClass="taglib-empty-result-message-header-has-plus-btn"
  ...
>
```

If you don't want to use the default animations packaged with 7.0, you can use custom animations instead. This is covered next.

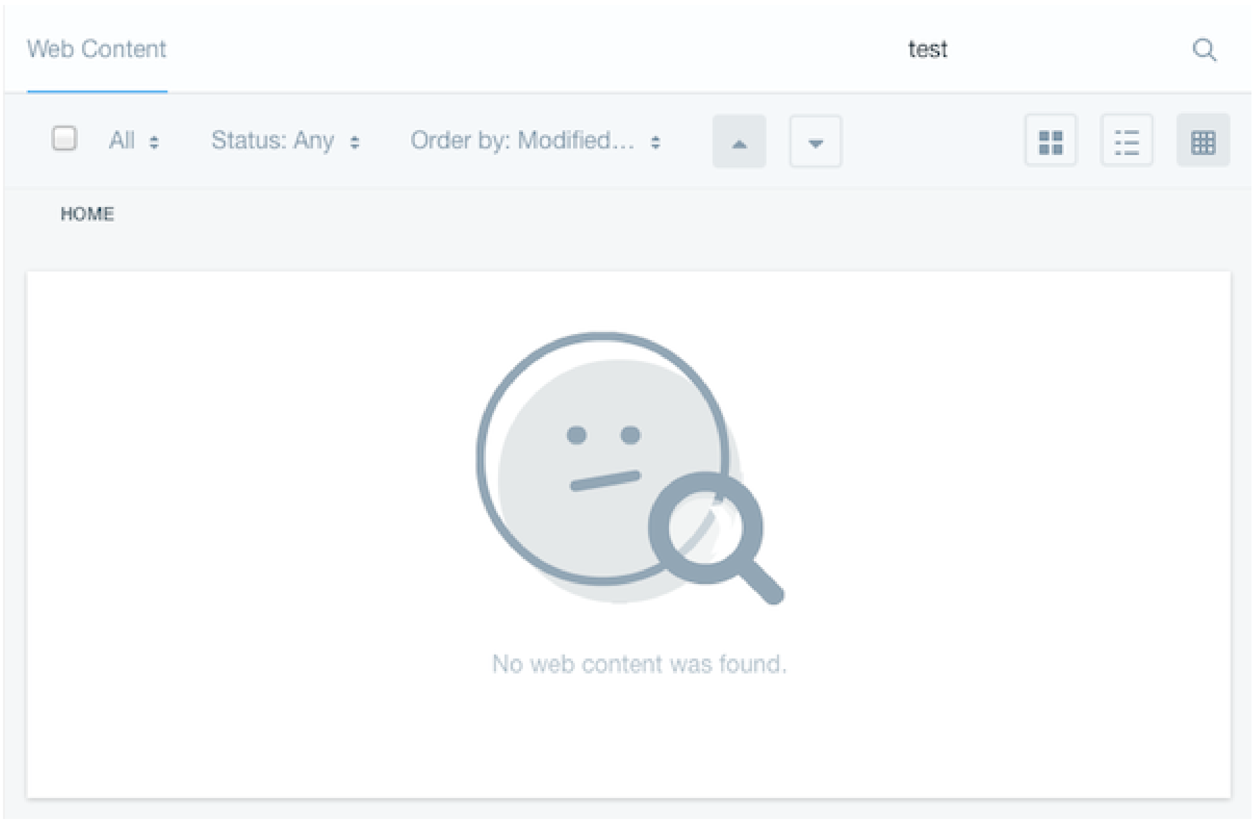


Figure 71.13: You can use the empty search result animation to show that no search results were found.

Using Custom Animations

As stated earlier, each animation has a matching CSS class that the search container uses. To use a custom animation, therefore, you must modify the existing styles.

There are two approaches you can take:

- Overwrite the existing styles to replace the default animations
- Create new styles to make the animation available to the search container

Regardless of the approach you choose, you must provide the CSS styles in a Theme, Themelet, or Theme Contributor. These styles point to the animation's source. You can provide the animation however you like: as long as you have a valid URL (relative or absolute) that points to the animation, you can use it.

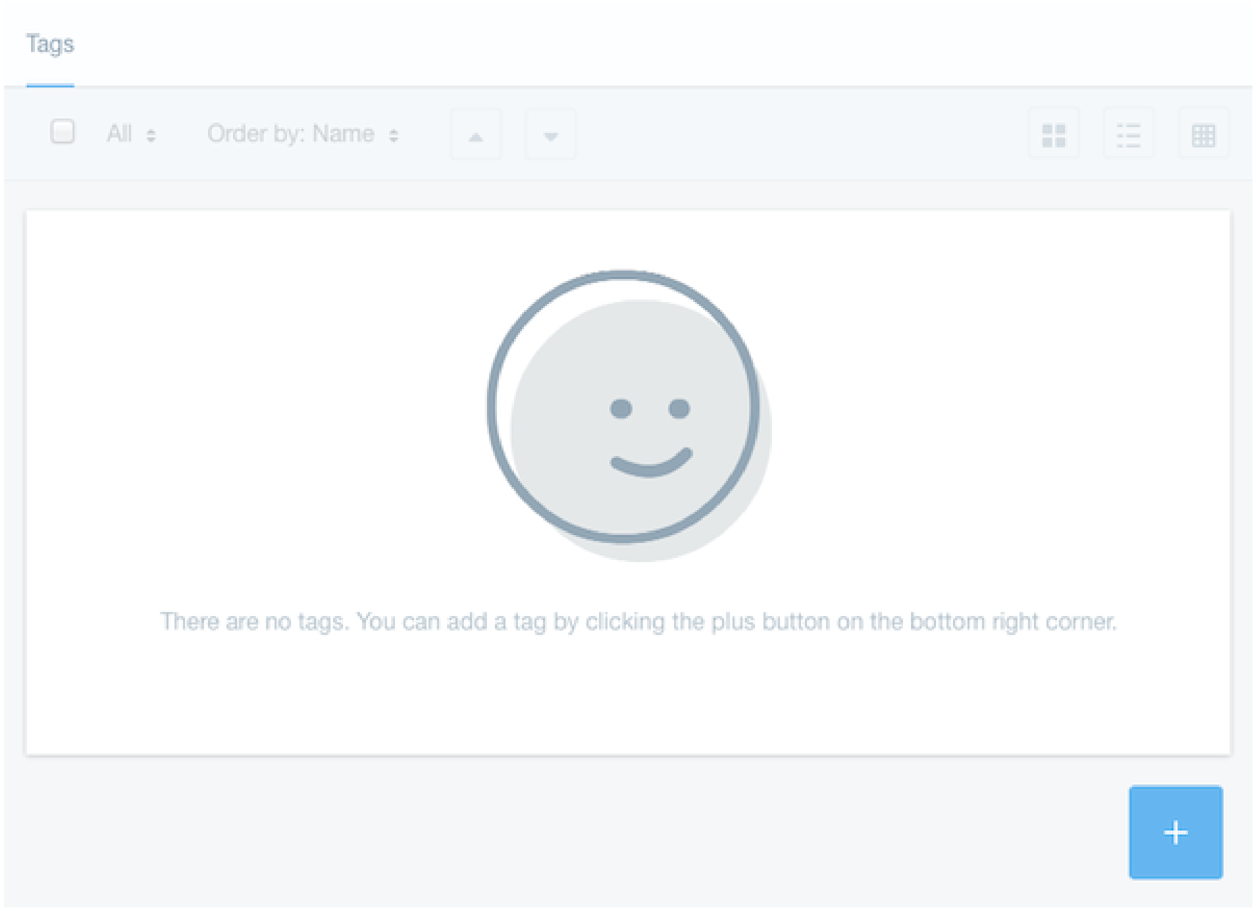
The default search container animation styles are provided by the _empty_result_message.scss file.

Note: Search containers can also contain static images for the no results message if you prefer. Just use a valid image type instead. All animations must be of type `GIF` though.

You can learn how to replace the default animations next.

Replacing Default Empty Results Message Animations

Follow these steps to replace the existing animations with your own:



There are no tags. You can add a tag by clicking the plus button on the bottom right corner.

Figure 71.14: If you can use the add button to add entities to the app, use the has plus button animation.

1. Make your custom animation available in your theme, themelet, Documents and Media repository, etc. For example, place the `.gif` in your theme's `images` folder.

2. Inside your CSS file (`_custom.scss` for example), override the animation styles that you want to replace. For example, to replace the default animation, include the following styles for a custom animation located in the `images/emoticons` folder of a theme:

```
.taglib-empty-result-message {
    .taglib-empty-result-message-header {
        background-image:
        url(@theme_image_path@/emoticons/[your_custom_animation].gif);
    }
}
```

3. Inside your app's search container, use the `emptyResultsMessageCssClass` attribute, or use the `SearchContainer.setEmptyResultsMessageCssClass()` method. Below is an example configuration that uses the `emptyResultsMessageCssClass` attribute:

```
<liferay-ui:search-container
    emptyResultsMessage="no-results-were-found"
    emptyResultsMessageCssClass="taglib-empty-result-message-header"
    total="<%= total %>"
>
```

Here is an example configuration that uses the `SearchContainer.setEmptyResultsMessageCssClass()` method:

```
SearchContainer.setEmptyResultsMessageCssClass("taglib-empty-result-message-header")
```

Your custom animation now appears in the search container instead of the default animation. If instead you want to add your custom animation to the default ones available, follow the steps in the next section.

Adding A New Empty Results Message Animation

Adding an animation to the empty results message involves the same steps as replacing a default animation. The only difference is you must add a new CSS class. Follow these steps to create a new style for your custom search container animation:

1. Make your custom animation available in your theme, themelet, Documents and Media repository, etc. For example, place the GIF in your theme's `images` folder.

2. Inside your CSS file (`_custom.scss` for example), add the styles for your new class, wrapped with `.taglib-empty-result-message`. For example, the styles below add a custom animation for a class called `my-custom-message-header`:

```
.taglib-empty-result-message {
    .my-custom-message-header {
        background-image:
        url(@theme_image_path@/emoticons/[my_custom_animation].gif);
    }
}
```

3. Use the `emptyResultsMessageCssClass` attribute in the Search Container, or use the `SearchContainer.setEmptyResultsMess` method to use the new CSS class you just added. Below is an example configuration that uses the `emptyResultsMessageCssClass` attribute:

```
<liferay-ui:search-container
    emptyResultsMessage="no-results-were-found"
    emptyResultsMessageCssClass="my-custom-message-header"
    total="<%= total %>"
>
```

Here is an example configuration that uses the `SearchContainer.setEmptyResultsMessageCssClass()` method:

```
SearchContainer.setEmptyResultsMessageCssClass("my-custom-message-header")
```

Now you know how to set search container animations in your app!

Related Topics

Using the Liferay UI Taglib

Introduction to Liferay Search

71.7 Using Lexicon Icons in Your App

Whether you're updating your app to 7.0 or writing a new 7.0 app, follow the process here to use Lexicon's icons. You can find the list of available Lexicon icons on the Lexicon site.

Lexicon icons are defined with the `icon` attribute. For example, you define the icon in the management bar, inside the `liferay-frontend:management-bar-sidenav-toggler-button` taglib:

```
<liferay-frontend:management-bar-sidenav-toggler-button
        disabled="<%= false %>"
        href="javascript:;"
        icon="info-circle"
        label="info"
        sidenavId='<%= liferayPortletResponse.getNamespace() + "infoPanelId" %>'
/>
```

To use Lexicon icons outside the management bar, you have two options:

1. You can use the `liferay-ui:icon` taglib. For example:

    ```
    <liferay-ui:icon

        icon="icon-name"

        markupView="lexicon"

        message="message-goes-here"

    />
    ```

2. You can use the `aui:icon` taglib. For example:

    ```
    <aui:icon

         cssClass="icon-monospaced"

         image="times"

         markupView="lexicon"

    />
    ```

Note the addition of the `markupView="lexicon"` attribute. This ensures that the HTML is rendered with Lexicon markup.

That's it! Now you know how to use Lexicon icons in your apps.

Related Topics

Setting Search Container Animations

Creating Layouts inside Custom Portlets

Chapter 72

Customizing

In Liferay DXP, portlets let you add functionality and themes let you style your sites. But how do you modify and add to existing functionality in Liferay DXP and portlets? How do you change their content? The *Customizing* tutorials answer these questions and demonstrate how to affect your site in the following ways:

- Add, modify, or remove content from Liferay DXP and portlets
- Modify behavior
- Perform actions that respond to events

CHAPTER 73

CUSTOMIZING JSPS

There are several different ways to customize JSPs in portlets and the core. Liferay DXP's API provides the safest ways to customize them. If you customize a JSP by other means, new versions of the JSP can render your customization invalid and leave you with runtime errors. It's highly recommended to use one of the API-based ways.

73.1 Using Liferay's API to Override a JSP

Here are API-based approaches to overriding JSPs in Liferay DXP:

Approach	Description	Cons/Limitations
Dynamic includes	Adds content at dynamic include tags.	Limited to JSPs that have `dynamic-include` tags (or tags whose classes inherit from `IncludeTag`). Only inserts content in the JSPs at the dynamic include tags.
Portlet filters	Modifies portlet requests and/or responses to simulate a JSP customization.	Although this approach doesn't directly customize a JSP, it achieves the effect of a JSP customization.

73.2 Overriding a JSP Without Using Liferay's API

It's strongly recommended to customize JSPs using Liferay DXP's API, as the previous section describes. As of Liferay 7.0, overriding a JSP using an OSGi fragment or a Custom JSP Bag are both deprecated. Since these approaches are not based on APIs there's no way to guarantee that they'll fail gracefully. Instead, if your customization is buggy (because of your code or because of a change in Liferay), you are most likely to find out at runtime, where functionality breaks and nasty log errors greet you.

If you're maintaining a JSP customization that uses one of these approaches, you should know how they work. This section describes them and links to their tutorials.

Here are ways to customize JSPs without using Liferay DXP's API:

Approach	Description	Cons/Limitations
OSGi fragment (deprecated as of Liferay 7.0)	Completely overrides a module's JSP using an OSGi fragment	Changes to the original JSP or module can cause runtime errors.
Custom JSP bag (deprecated as of Liferay 7.0)	Completely override a Liferay DXP core JSP or one of its corresponding `-ext.jsp` files.	For Liferay DXP core JSPs only. Changes to the original JSP or module can cause runtime errors.

All the JSP customization approaches are available to you. It's time to customize some JSPs!

73.3 Customizing JSPs with Dynamic Includes

The `liferay-util:dynamic-include` tag is an extension point for inserting content (e.g., JavaScript code, HTML, and more). To do this, create a module that has content you want to insert, register that content with the dynamic include tag, and deploy your module.

Note: If the JSP you want to customize has no `liferay-util:dynamic-include` tags (or tags whose classes inherit from `IncludeTag`), you must use a different customization approach, such as portlet filters.

We'll demonstrate how dynamic includes work using the Blogs entries. For reference, you can download the example module.

1. Find the `liferay-util:dynamic-include` tag where you want to insert content and note the tag's key.

   The Blogs app's `view_entry.jsp` has a dynamic include tag at the top and another at the very bottom.

   ```
   <%@ include file="/blogs/init.jsp" %>

   <liferay-util:dynamic-include key="com.liferay.blogs.web#/blogs/view_entry.jsp#pre" />

       ... JSP content is here

   <liferay-util:dynamic-include key="com.liferay.blogs.web#/blogs/view_entry.jsp#post" />
   ```

   Here are the Blogs view entry dynamic include keys:

   - `key="com.liferay.blogs.web#/blogs/view_entry.jsp#pre"`
   - `key="com.liferay.blogs.web#/blogs/view_entry.jsp#post"`

2. Create a module (e.g., `blade create my-dynamic-include`). The module will hold your dynamic include implementation.

3. Specify compile-only dependencies, like these Gradle dependencies, in your module build file:

   ```
   dependencies {
       compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
       compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
       compileOnly group: "com.liferay", name: "com.liferay.petra.string", version: "1.0.0"
       compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
       compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"
   }
   ```

4. Create an OSGi component class that implements the `DynamicInclude` interface.

   Here's an example dynamic include implementation for Blogs:

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osgi.service.component.annotations.Component;

import com.liferay.portal.kernel.servlet.taglib.DynamicInclude;

@Component(
    immediate = true,
    service = DynamicInclude.class
)
public class BlogsDynamicInclude implements DynamicInclude {

    @Override
    public void include(
            HttpServletRequest request, HttpServletResponse response,
            String key)
        throws IOException {

        PrintWriter printWriter = response.getWriter();

        printWriter.println(
            "<h2>Added by Blogs Dynamic Include!</h2><br />");
    }

    @Override
    public void register(DynamicIncludeRegistry dynamicIncludeRegistry) {
        dynamicIncludeRegistry.register(
            "com.liferay.blogs.web#/blogs/view_entry.jsp#pre");
    }

}
```

   Giving the class a `@Component` annotation that has the service attribute `service = DynamicInclude.class` makes the class a `DynamicInclude` service component.

```
@Component(
    immediate = true,
    service = DynamicInclude.class
)
```

   In the `include` method, add your content. The example `include` method writes a heading.

```
@Override
public void include(
        HttpServletRequest request, HttpServletResponse response,
        String key)
    throws IOException {

    PrintWriter printWriter = response.getWriter();

    printWriter.println(
    "<h2>Added by Blogs Dynamic Include!</h2><br />");
}
```

In the register method, specify the dynamic include tag you want to use. The example register method targets the dynamic include at the top of the Blogs view_entry.jsp.

```
@Override
public void register(DynamicIncludeRegistry dynamicIncludeRegistry) {
    dynamicIncludeRegistry.register(
        "com.liferay.blogs.web#/blogs/view_entry.jsp#pre");
}
```

Once you've deployed your module, the overridden JSP dynamically includes your content. Congratulations on injecting dynamic content into a JSP!

73.4 JSP Overrides Using Portlet Filters

Portlet filters let you intercept portlet requests before they're processed and portlet responses after they're processed but before they're sent back to the client. You can operate on the request and / or response to modify the JSP content. Unlike dynamic includes, portlet filters give you access to all of the content sent back to the client.

We'll demonstrate using a portlet filter to modify content in Liferay's Blogs portlet. For reference, you can download the example module.

Create a new module and make sure it specifies these compile-only dependencies, shown here in Gradle format:

```
dependencies {
    compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"
}
```

Create an OSGi component class that implements the javax.portlet.filter.RenderFilter interface.

Here's an example portlet filter implementation for Blogs:

```
import java.io.IOException;

import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import javax.portlet.filter.FilterChain;
import javax.portlet.filter.FilterConfig;
import javax.portlet.filter.PortletFilter;
import javax.portlet.filter.RenderFilter;
import javax.portlet.filter.RenderResponseWrapper;

import org.osgi.service.component.annotations.Component;

import com.liferay.portal.kernel.util.PortletKeys;

@Component(
    immediate = true,
    property = {
            "javax.portlet.name=" + PortletKeys.BLOGS
    },
    service = PortletFilter.class
)
public class BlogsRenderFilter implements RenderFilter {

    @Override
    public void init(FilterConfig config) throws PortletException {
    }

    @Override
    public void destroy() {

    }

    @Override
    public void doFilter(RenderRequest request, RenderResponse response, FilterChain chain)
            throws IOException, PortletException {

        RenderResponseWrapper renderResponseWrapper = new BufferedRenderResponseWrapper(response);

        chain.doFilter(request, renderResponseWrapper);

        String text = renderResponseWrapper.toString();

        if (text ≠ null) {
            String interestingText = "<input  class=\"field form-control\"";

            int index = text.lastIndexOf(interestingText);

            if (index ≥ 0) {
                String newText1 = text.substring(0, index);
                String newText2 = "\n<p>Added by Blogs Render Filter!</p>\n";
                String newText3 = text.substring(index);

                String newText = newText1 + newText2 + newText3;

                response.getWriter().write(newText);
            }
        }
    }

}
```

Make your class a `PortletFilter` service component by giving it the `@Component` annotation that has the service attribute `service = PortletFilter.class`. Target the portlet whose content you're overriding by assigning it a javax.portlet.name property that's the same as your portlet's key. Here's the example `@Component` annotation:

```
@Component(
   immediate = true,
   property = {
           "javax.portlet.name=" + PortletKeys.BLOGS
   },
   service = PortletFilter.class
)
```

Override the `doFilterMethod` to operate on the request or response to produce the content you want. The example appends a paragraph stating `Added by Blogs Render Filter!` to the portlet content:

```
@Override
public void doFilter(RenderRequest request, RenderResponse response, FilterChain chain)
        throws IOException, PortletException {

    RenderResponseWrapper renderResponseWrapper = new BufferedRenderResponseWrapper(response);

    chain.doFilter(request, renderResponseWrapper);

    String text = renderResponseWrapper.toString();

    if (text ≠ null) {
        String interestingText = "<input  class=\"field form-control\"";
        int index = text.lastIndexOf(interestingText);

        if (index ≥ 0) {
            String newText1 = text.substring(0, index);
            String newText2 = "\n<p>Added by Blogs Render Filter!</p>\n";
            String newText3 = text.substring(index);

            String newText = newText1 + newText2 + newText3;

            response.getWriter().write(newText);
        }
    }
}
```

The example uses a `RenderResponseWrapper` extension class called `BufferedRenderResponseWrapper`. `BufferedRenderResponseWrapper` is a helper class whose `toString` method returns the current response text and whose `getWriter` method lets you write data to the response before it's sent back to the client.

```
import java.io.CharArrayWriter;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;

import javax.portlet.RenderResponse;
import javax.portlet.filter.RenderResponseWrapper;

public class BufferedRenderResponseWrapper extends RenderResponseWrapper {

    public BufferedRenderResponseWrapper(RenderResponse response) {
        super(response);

        charWriter = new CharArrayWriter();
    }

    public OutputStream getOutputStream() throws IOException {
        if (getWriterCalled) {
            throw new IllegalStateException("getWriter already called");
        }

        getOutputStreamCalled = true;

        return super.getPortletOutputStream();
    }

    public PrintWriter getWriter() throws IOException {
        if (writer ≠ null) {
            return writer;
        }

        if (getOutputStreamCalled) {
            throw new IllegalStateException("getOutputStream already called");
        }

        getWriterCalled = true;

        writer = new PrintWriter(charWriter);

        return writer;
    }

    public String toString() {
        String s = null;

        if (writer ≠ null) {
            s = charWriter.toString();
        }

        return s;
    }

    protected CharArrayWriter charWriter;
    protected PrintWriter writer;
    protected boolean getOutputStreamCalled;
    protected boolean getWriterCalled;

}
```

Once you've deployed your module, the portlet's JSP shows your custom content.

Your portlet filter operates directly on portlet response content. Unlike dynamic includes, portlet filters allow you to work with all of a JSP's content.

73.5 JSP Overrides Using OSGi Fragments

You can completely override JSPs using OSGi fragments. This approach is powerful but can make things unstable when the host module is upgraded:

1. By overriding an entire JSP, you might not account for new content or new widgets that are essential to new host module versions.
2. Fragments are tied to a specific host module version. If the host module is upgraded, the fragment detaches from it. In this scenario, the original JSPs are still available and the module is functional (but lacks your JSP enhancements).

Liferay's API based approaches to overriding JSPs, on the other hand, provide more stability as they let you customize specific parts of the JSP that are safe to override. Also, the API based approaches don't limit your override to a specific host module version. In case you're maintaining existing JSP overrides that use OSGi fragments, however, this tutorial explains how they work.

An OSGi fragment that overrides a JSP requires these two things:

- Specifies a host module's symbolic name and version in the OSGi header `Fragment-Host` declaration.
- includes the original JSP with any modifications you need to make.

For more information about fragment modules, you can refer to section 3.14 of the OSGi Alliance's core specification document.

Declaring a Fragment Host

There are two players in this game: the fragment and the host. The fragment is a parasitic module that attaches itself to a host. That sounds harsh, so let's compare the fragment-host relationship to the relationship between a pilot fish and a huge, scary shark. It's symbiotic, really. Your fragment module benefits by not doing much work (like the pilot fish who benefits from the shark's hunting prowess). In return, the host module gets whatever benefits you've conjured up in your fragment's JSPs (for the shark, it gets free dental cleanings!). To the OSGi runtime, your fragment is part of the host module.

Your fragment must declare two things to the OSGi runtime regarding the host module:

1. The Bundle Symbolic Name of the host module.

   This is the module containing the original JSP.

2. The exact version of the host module to which the fragment belongs.

Both are declared using the OSGi header `Fragment-Host`.

```
Fragment-Host: com.liferay.login.web;bundle-version="[1.0.0,1.0.1)"
```

Supplying a specific host module version is important. If that version of the module isn't present, your fragment won't attach itself to a host, and that's a good thing. A new version of the host module might have changed its JSPs, so if your now-incompatible version of the JSP is applied to the host module, you'll break the functionality of the host. It's better to detach your fragment and leave it lonely in the OSGi runtime than it is to break the functionality of an entire application.

Provide the Overridden JSP

There are two possible naming conventions for targeting the host original JSP: `portal` or `original`. For example, if the original JSP is in the folder `/META-INF/resources/login.jsp`, then the fragment bundle should contain a JSP with the same path, using the following pattern:

```
<liferay-util:include
    page="/login.original.jsp" (or login.portal.jsp)
    servletContext="<%= application %>"
/>
```

After that, make your modifications. Just make sure you mimic the host module's folder structure when overriding its JAR. If you're overriding Liferay's login application's `login.jsp` for example, you'd put your own `login.jsp` in

```
my-jsp-fragment/src/main/resources/META-INF/resources/login.jsp
```

If you need to post-process the output, you can update the pattern to include Liferay DXP's buffering mechanism. Below is an example that overrides the original `create_account.jsp`:

```
<%@ include file="/init.jsp" %>

<liferay-util:buffer var="html">
    <liferay-util:include page="/create_account.portal.jsp"
    servletContext="<%= application %>"/>
</liferay-util:buffer>

<liferay-util:buffer var="openIdFieldHtml"><aui:input name="openId"
type="hidden" value="<%= ParamUtil.getString(request, "openId") %>" />
</liferay-util:buffer>

<liferay-util:buffer var="userNameFieldsHtml"><liferay-ui:user-name-fields />
</liferay-util:buffer>

<liferay-util:buffer var="errorMessageHtml">
    <liferay-ui:error
    exception="<%= com.liferay.portal.kernel.exception.NoSuchOrganizationException.class %>" message="no-such-registration-code" />
</liferay-util:buffer>

<liferay-util:buffer var="registrationCodeFieldHtml">
            <aui:input name="registrationCode" type="text" value="">
                    <aui:validator name="required" />
            </aui:input>
</liferay-util:buffer>

<%
    html = com.liferay.portal.kernel.util.StringUtil.replace(html,
      openIdFieldHtml, openIdFieldHtml + errorMessageHtml);
    html = com.liferay.portal.kernel.util.StringUtil.replace(html,
      userNameFieldsHtml, userNameFieldsHtml + registrationCodeFieldHtml);
%>

<%=html %>
```

Note: An OSGi fragment can access all of the fragment host's packages—it doesn't need to import them from another bundle. bnd adds external packages the fragment uses (even ones in the fragment host) to the fragment's `Import-Package: [package],...` OSGi manifest header. That's fine for packages exported to the OSGi runtime. The problem is, however, when bnd tries to import a host's internal package (a package the host doesn't export). The OSGi runtime can't activate the fragment because the internal package remains an `Unresolved requirement`—a fragment shouldn't import a fragment host's packages.

If your fragment uses an internal package from the fragment host, continue using it but explicitly exclude the package from your bundle's `Import-Package` OSGi manifest header. This `Import-Package` header, for example, excludes packages that match `com.liferay.portal.search.web.internal.*`.

```
Import-Package: !com.liferay.portal.search.web.internal.*,*
```

Now you can easily modify the JSPs of any application in Liferay.

To see a sample JSP-modifying fragment in action, look at the BLADE project named `module-jsp-override`.

Related Topics

Upgrading App JSP Hooks

73.6 JSP Overrides Using Custom JSP Bag

Liferay's API based approaches to overriding JSPs are the best way to override JSPs in apps and in the core. You can also use Custom JSP Bags to override core JSPs. But the approach is not as stable as the API based approaches. If your Custom JSP Bag's JSP is buggy (because of your code or because of a change in Liferay), you are most likely to find out at runtime, where functionality breaks and nasty log errors greet you. In the case that you're maintaining existing Custom JSP Bags, however, this tutorial explains how they work.

A Custom JSP Bag module must satisfy these criteria:

- Includes a class that implements the `CustomJspBag` interface.
- Registers the service in the OSGi runtime.
- Provides the JSP you're extending.

The module provides transportation for this code into Liferay's OSGi runtime. When configuring it to build a proper JAR, map the path of the JSPs in the JAR to their path in your module. In a `bnd.bnd` file you could specify

```
-includeresource: META-INF/jsps=src/META-INF/custom_jsps
```

If you're using the Maven Standard Directory Layout and placing your JSPs under `src/main/resources`, you can ignore the `-includeresource` directive.

Any core JSPs you're customizing should be put into this folder, and the rest of their path and their name must match exactly the path and name of the JSP that's nested underneath `portal-web/docroot/html`. For example, if you're overriding

```
portal-web/docroot/html/common/themes/bottom-ext.jsp
```

and you used the `includeresource` directive above, put the overridden JSP in this folder of your module:

```
my-module/src/META-INF/custom_jsps/html/common/themes/bottom-ext.jsp
```

Implement a Custom JSP Bag

Create a class that implements `CustomJspBag`. The overall goal is to make sure that Liferay (specifically the `CustomJspBagRegistryUtil` class) loads the JSPs from your module upon activation.

```
public class MyCustomJspBag implements CustomJspBag {
```

When the Component is activated, you need to add the URL path for all your custom core JSPs (by directory path) to a `List`.

```
    @Activate
    protected void activate(BundleContext bundleContext) {
        bundle = bundleContext.getBundle();

        _customJsps = new ArrayList<>();

        Enumeration<URL> entries = bundle.findEntries(
            getCustomJspDir(), "*.jsp", true);

        while (entries.hasMoreElements()) {
            URL url = entries.nextElement();

            _customJsps.add(url.getPath());
        }
    }

    ...

    private List<String> _customJsps;

    private Bundle bundle;
```

In the custom JSP bag you'll need to override the following methods:

- **`getCustomJspDir`:** Return the directory path in your module's JAR where the JSPs are placed (for example, `META-INF/jsps`).
- **`getCustomJsps`:** Return a List of the custom JSP URL paths.
- **`getURLContainer`:** Return a new `com.liferay.portal.kernel.url.URLContainer`. Instantiate the URL container and override its `getResources` and `getResource` methods. The `getResources` method is for looking up all of the paths to resources in the container by a given path. It should return a `HashSet` of `Strings` for the matching custom JSP paths. The `getResource` method returns one specific resource by its name (the path included).
- **`isCustomJspGlobal`:** Return `true`.

For an example of a full class that provides a working implementation of a custom JSP bag, refer to the core-jsp-override BLADE project.

Register the Custom JSP Bag

Register the custom JSP bag implementation from your module in the OSGi runtime with three properties:

- **`context.id`:** Specify your custom JSP bag class name. For example, `MyCustomJspBag`.
- **`contex.name`:** This should be a more human readable name, like `My Custom JSP Bag`.
- **`service.ranking:integer`:** This will determine the priority of your implementation. If you specify `100` here, and one of your coworkers develops a separate custom JSP bag implementation and gives theirs a ranking of `101`, you're out of luck. Theirs will take precedence. Logically then, you should use `102`.

Extend a JSP

If you want to add something to a core JSP, see if it has an empty `-ext.jsp` and override that instead of the whole JSP. It'll keep things simpler and more stable, since the full JSP might change significantly, breaking your customization in the process. By overriding the `-ext.jsp`, you're only relying on the original JSP including the `-ext.jsp`. For an example, open `portal-web/docroot/html/common/themes/bottom.jsp`, and scroll to the end. You'll see this:

```
<liferay-util:include page="/html/common/themes/bottom-ext.jsp" />
```

If you need to add something to `bottom.jsp`, override `bottom-ext.jsp`.

As of 7.0, the content from the following JSP files formerly in `html/common/themes` are inlined to improve performance. - `body_bottom-ext.jsp` - `body_top-ext.jsp` - `bottom-ext.jsp` - `bottom-test.jsp`

They're no longer explicit files in the code base. But you can still create them in your module to add functionality and content.

Remember, this type of customization should be seen as a last resort. There's a risk that your override will break due to the nature of this implementation, and core functionality in Liferay can go down with it. If the JSP you want to override is in another module, refer to the section on API based approaches to overriding JSPs.

Site Scoped JSP Customization

In Liferay Portal 6.2, you could use Application Adapters to scope your core JSP customizations to a specific site. Since the majority of JSPs were moved into modules for 7.0, the use case for this has shrunk considerably. If you need to scope a core JSP customization to a site, prepare an application adapter as you would have for Liferay Portal 6.2, and deploy it to 7.0. It will still work. However, note that this approach is deprecated in 7.0 and won't be supported at all in Liferay 8.0.

Related Topics

Upgrading Core JSP Hooks

73.7 Overriding Liferay DXP's Default YUI and AUI Modules

Liferay DXP contains several default YUI/AUI modules. You may need to override functionality provided by these module's scripts. It's possible to override JSPs using fragments, but you can't with JavaScript files. Instead, you create a custom AUI module containing three things:

- A copy of the original module's JavaScript file containing your modifications
- A `config.js` file that specifies the modified JavaScript file's path and the module it overrides
- A `bnd.bnd` file that tells the OSGi container to override the original

Follow these steps:

1. Create an OSGi module to override the original one. For example, you can create a module named `session-js-override-web` to override Liferay DXP's `session.js` file.

2. Create a `src/main/resources/META-INF/resources/js` folder in your module, copy the original JavaScript file into it, and rename it. For example, create a copy of the `session.js` module and rename it `session-override.js`. Make sure you also rename the module definition inside the `session-override.js`, e.g. `AUI().add('liferay-session-override', ....`

3. Apply your modifications and save the file.

4. Next, you'll write your module's configuration file (`config.js`) to apply your override. Add the `config.js` file to the module's `src/main/resources/META-INF/resources/js` folder. The example `config.js` file below specifies the `condition` that the YUI/AUI Loader should load the custom AUI module (`liferay-session-override`) `instead` (indicated with the `when` property) of the `trigger` module (`liferay-session`). You can follow this same pattern to create your module's `config.js` file:

```
;(function() {

    var base = MODULE_PATH + '/js/';

    AUI().applyConfig(
        {
            groups: {
                mymodulesoverride: { //mymodulesoverride
                    base: base,
                    combine: Liferay.AUI.getCombine(),
                    filter: Liferay.AUI.getFilterConfig(),
                    modules: {
                        'liferay-session-override': { //my-module-override
                            path: 'session-override.js', //my-module.js
                            condition: {
                                name: 'liferay-session-override', //my-module-override
                                trigger: 'liferay-session', //original module
                                when: 'instead'
                            }
                        }
                    },
                    root: base
                }
            }
        }
    );
})();
```

5. Finally, you must configure your `bnd.bnd` file. For the system to apply the changes, you must specify the `config.js`'s location with the `Liferay-JS-Config` BND header. The `liferay-session-override` module from the previous example has the configuration below in its `bnd.bnd` file:

```
Bundle-Name: session-js-override
Bundle-SymbolicName: session.js.override.web
Bundle-Version: 1.0.0
Liferay-JS-Config:/META-INF/resources/js/config.js
Web-ContextPath: /liferay-session-override-web
```

Now you know how to override Liferay DXP's default YUI/AUI modules!

Related Topics

Overriding JSPs
Configuring Modules for Liferay DXP's Module Loaders

73.8 Overriding Liferay Services (Service Wrappers)

Why might you need to customize Liferay services? Perhaps you've added a custom field to Liferay's `User` object and you want its value to be saved whenever the `addUser` or `updateUser` methods of Liferay's API are called. Or maybe you want to add some additional logging functionality to some of Liferay's APIs. Whatever your case may be, Liferay's service wrappers provide easy-to-use extension points for customizing Liferay's services.

To create a module that overrides one of Liferay's services, follow the Service Wrapper Template reference article to create a `servicewrapper` project type.

As an example, here's the `UserLocalServiceOverride` class that's generated in the Service Wrapper Template tutorial:

```
package com.liferay.docs.serviceoverride;

import com.liferay.portal.kernel.service.UserLocalServiceWrapper;
import com.liferay.portal.kernel.service.ServiceWrapper;
import org.osgi.service.component.annotations.Component;

@Component(
    immediate = true,
    property = {
    },
    service = ServiceWrapper.class
)
public class UserLocalServiceOverride extends UserLocalServiceWrapper {

    public UserLocalServiceOverride() {
        super(null);
    }

}
```

Notice that you must specify the fully qualified class name of the service wrapper class that you want to extend. The service argument was used in full in this import statement:

```
import com.liferay.portal.service.UserLocalServiceWrapper
```

This import statement, in turn, allowed the short form of the service wrapper class name to be used in the class declaration of your component class:

```
public class UserLocalServiceOverride extends UserLocalServiceWrapper
```

The bottom line is that when using `blade create` to create a service wrapper project, you must specify a fully qualified class name as the `service` argument. (This is also true when using `blade create` to create a service project.) For information about creating service projects, please see the Service Builder tutorial.

The generated `UserLocalServiceOverride` class does not actually customize any Liferay service. Before you can test that your service wrapper module actually works, you need to override at least one service method.

Open your `UserLocalServiceOverride` class and add the following methods:

```
@Override
public int authenticateByEmailAddress(long companyId, String emailAddress,
        String password, Map<String, String[]> headerMap,
        Map<String, String[]> parameterMap, Map<String, Object> resultsMap)
    throws PortalException {

    System.out.println(
        "Authenticating user by email address " + emailAddress);
    return super.authenticateByEmailAddress(companyId, emailAddress, password,
        headerMap, parameterMap, resultsMap);
}

@Override
public User getUser(long userId) throws PortalException {
    System.out.println("Getting user by id " + userId);
    return super.getUser(userId);
}
```

Each of these methods overrides a Liferay service method. These implementations merely add a few print statements that are executed before the original service implementations are invoked.

Lastly, you must add the following method to the bottom of your service wrapper so it can find the appropriate service it's overriding on deployment.

```
@Reference(unbind = "-")
private void serviceSetter(UserLocalService userLocalService) {
    setWrappedService(userLocalService);
}
```

Now you're ready to build your project. Navigate to your project's root folder and run `../../gradlew build`. The JAR file representing your portlet module is produced in your project's `build/libs` directory.

To deploy your project, run this command from your project's root directory:

```
blade deploy
```

Blade CLI detects your locally running Liferay instance and deploys the specified module to Liferay's module framework. After running the `blade deploy` command, you should see a message like this:

```
Installed or updated bundle 334
```

Use the Gogo shell to confirm that your module was installed: Run `blade sh lb` at the prompt. If your module was installed, you'll see an entry like this:

```
335|Active     |    1|com.liferay.docs.serviceoverride (1.0.0.201502122109)
```

Finally, log into your portal as an administrator. Navigate to the Users section of the Control Panel. Confirm that your customizations of Liferay's user service methods have taken effect by checking Liferay's log for the print statements that you added. Congratulations! You've created and deployed a 7.0 service wrapper module!

Related Topics

Upgrading Service Wrappers
Installing Blade CLI
Creating Modules with Blade CLI

73.9 Overriding Language Keys

Liferay DXP Core and portlet module `language*.properties` files implement site internationalization. They're fully customizable, too. This tutorial demonstrates this in the following topics:

- Overriding Global Language Keys
- Overriding a Module's Language Keys

Modifying Global Language Keys

Language files contain translations of your application's user interface messages. But you can also override the default language keys globally and in other applications (including your own). Here are the steps for overriding language keys:

1. Determine the language keys to override
2. Override the keys in a new language properties file
3. Create a Resource Bundle service component

Note: Many applications that were once part of Liferay Portal 6.2 are now modularized. Their language keys might have been moved out of Liferay's language properties files and into one of the application modules. The process for overriding a module's language keys is different from the process for overriding global language keys.

Determine the language keys to override

So how do you find global language keys? They're in the `Language[xx_XX].properties` files in the source code or your Liferay DXP bundle.

- From the source:

  `/portal-impl/src/content/Language[_xx_XX].properties`

- From a bundle:

  `portal-impl.jar`

All language properties files contain properties you can override, like the language settings properties:

```
##
## Language settings
##

...
lang.user.name.field.names=prefix,first-name,middle-name,last-name,suffix
lang.user.name.prefix.values=Dr,Mr,Ms,Mrs
lang.user.name.required.field.names=last-name
lang.user.name.suffix.values=II,III,IV,Jr,Phd,Sr
...
```

There are also many simple keys you can override to update default messages and labels.

```
##
## Category titles
##

category.admin=Admin
category.alfresco=Alfresco
category.christianity=Christianity
category.cms=Content Management
...
```

For example, Figure 1 shows a button that uses Liferay's `publish` default language key.

```
publish=Publish
```

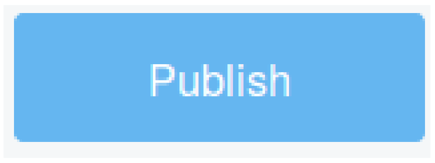


Figure 73.1: Messages displayed in Liferay's user interface can be customized.

Next, you'll learn how to override this key.

Override the keys in a new language properties file

Once you know the keys to override, create a language properties file for the locale you want (or the default `Language.properties` file) in your module's `src/main/resources/content` folder. In your file, define the keys your way. For example, you could override the `publish` key.

```
publish=Publish Override
```

To enable your change, you must create a resource bundle service component to reference your language file.

Create a Resource Bundle service component

In your module, create a class that extends `java.util.ResourceBundle` for the locale you're overriding. Here's an example resource bundle class for the `en_US` locale:

```
@Component(
    property = { "language.id=en_US" },
    service = ResourceBundle.class
)
public class MyEnUsResourceBundle extends ResourceBundle {

    @Override
    protected Object handleGetObject(String key) {
        return _resourceBundle.getObject(key);
    }

    @Override
    public Enumeration<String> getKeys() {
        return _resourceBundle.getKeys();
    }

    private final ResourceBundle _resourceBundle = ResourceBundle.getBundle(
        "content.Language_en_US", UTF8Control.INSTANCE);

}
```

The class's `_resourceBundle` field is assigned a `ResourceBundle`. The call to `ResourceBundle.getBundle` needs two parameters. The `content.Language_en_US` parameter is the language file's qualified name with respect to the module's `src/main/resources` folder. The second parameter is a `control` that sets the language syntax of the resource bundle. To use language syntax identical to Liferay's syntax, import Liferay's `com.liferay.portal.kernel.language.UTF8Control` class and set the second parameter to `UTF8Control.INSTANCE`.

The class's `@Component` annotation declares it an OSGi `ResourceBundle` service component. It's `language.id` property designates it for the `en_US` locale.

```
@Component(
    property = { "language.id=en_US" },
    service = ResourceBundle.class
)
```

The class overrides these methods:

- **`handleGetObject`:** Looks up the key in the module's resource bundle (which is based on the module's language properties file) and returns the key's value as an `Object`.

- **`getKeys`:** Returns an `Enumeration` of the resource bundle's keys.

Your resource bundle service component redirects the default language keys to your module's language key overrides.

Note: Global language key overrides for multiple locales require a separate module for each locale. Each module's ResourceBundle extension class (like the MyEnUsResourceBundle class above) must specify its locale in the language.id component property definition and in the language file qualified name parameter. For example, here is what they look like for the Spanish locale.

Component definition:

```
@Component(
    property = { "language.id=es_ES" },
    service = ResourceBundle.class
)
```

Resource bundle assignment:

```
private final ResourceBundle _resourceBundle = ResourceBundle.getBundle(
    "content.Language_es_ES", UTF8Control.INSTANCE);
```

Important: If your module uses language keys from another module and overrides any of that other module's keys, make sure to use OSGi headers to specify the capabilities your module requires and provides. This lets you prioritize resource bundles from the modules.

To see your Liferay language key overrides in action, deploy your module and visit the portlets and pages that use the keys.

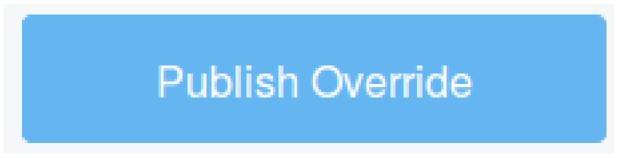


Figure 73.2: This button uses the overridden publish key.

That's all there is to overriding global language keys.

Overriding a Module's Language Keys

What do you do if the language keys you want to modify are in one of Liferay's applications or another module whose source code you don't control? Since module language keys are in the respective module, the process for overriding a module's language keys is different from the process of overriding global language keys.

Here is the process:

1. Find the module and its metadata and language keys
2. Write your custom language key values
3. Prioritize your module's resource bundle

Find the module and its metadata and language keys

In Gogo shell, list the bundles and grep for keyword(s) that match the portlet's display name. Language keys are in the portlet's web module (bundle). When you find the bundle, note its ID number.

To find the Blogs portlet, for example, your Gogo commands and output might look like this:

```
g! lb | grep Blogs
  152|Active     |    1|Liferay Blogs Service (1.0.2)
  184|Active     |    1|Liferay Blogs Editor Config (2.0.1)
  202|Active     |    1|Liferay Blogs Layout Prototype (2.0.2)
  288|Active     |    1|Liferay Blogs Recent Bloggers Web (1.0.2)
  297|Active     |    1|Liferay Blogs Item Selector Web (1.0.2)
  374|Active     |    1|Liferay Blogs Item Selector API (2.0.1)
  448|Active     |    1|Liferay Blogs API (3.0.1)
  465|Active     |    1|Liferay Blogs Web (1.0.6)
true
```

List the bundle's headers by passing its ID to the headers command.

```
g! headers 465

Liferay Blogs Web (465)
-----------------------
Manifest-Version = 1.0
Bnd-LastModified = 1459866186018
Bundle-ManifestVersion = 2
Bundle-Name = Liferay Blogs Web
Bundle-SymbolicName = com.liferay.blogs.web
Bundle-Version: 1.0.6
...
Web-ContextPath = /blogs-web
g!
```

Note the `Bundle-SymbolicName`, `Bundle-Version`, and `Web-ContextPath`. The `Web-ContextPath` value, following the `/`, is the servlet context name.

Important: Record the servlet context name, bundle symbolic name and version, as you'll use them to create the resource bundle loader later in the process.

For example, here are those values for Liferay Blogs Web module:

- Bundle symbolic name: `com.liferay.blogs.web`
- Bundle version: `1.0.6`
- Servlet context name: `blogs-web`

Next find the module's JAR file so you can examine its language keys. Liferay follows this module JAR file naming convention:

```
[bundle symbolic name]-[version].jar
```

For example, the Blogs Web version 1.0.6 module is in `com.liferay.blogs.web-1.0.6.jar`.

Here's where to find the module JAR:

- Liferay's Nexus repository
- `[Liferay Home]/osgi/modules`
- Embedded in an application's or application suite's LPKG file in `[Liferay Home]/osgi/marketplace`.

The language property files are in the module's `src/main/resources/content` folder. Identify the language keys you want to override in the `Language[_xx].properties` files.

Checkpoint: Make sure you have the required information for overriding the module's language keys:

- Language keys
- Bundle symbolic name
- Servlet context name

Next you'll write new values for the language keys.

Write custom language key values

Create a new module to hold a resource bundle loader and your custom language keys.

In your module's `src/main/resources/content` folder, create language properties files for each locale whose keys you want to override. In each language properties file, specify your language key overrides.

Next you'll prioritize your module's language keys as a resource bundle for the target module.

Prioritize Your Module's Resource Bundle

Now that your language keys are in place, use OSGi manifest headers to specify the language keys are for the target module. To compliment the target module's resource bundle, you'll aggregate your resource bundle with the target module's resource bundle. You'll list your module first to prioritize its resource bundle over the target module resource bundle. Here's an example of module `com.liferay.docs.l10n.myapp.lang` prioritizing its resource bundle over target module `com.liferay.blogs.web`'s resource bundle:

```
Provide-Capability:\
liferay.resource.bundle;resource.bundle.base.name="content.Language",\
liferay.resource.bundle;resource.bundle.aggregate:String="(bundle.symbolic.name=com.liferay.docs.l10n.myapp.lang),(bundle.symbolic.name=com.liferay.blogs.we
servlet.context.name=blogs-web
```

Let's examine the example `Provide-Capability` header.

1. `liferay.resource.bundle;resource.bundle.base.name="content.Language"` declares that the module provides a resource bundle whose base name is `content.language`.
2. The `liferay.resource.bundle;resource.bundle.aggregate:String=...` directive specifies the list of bundles whose resource bundles are aggregated, the target bundle, the target bundle's resource bundle name, and this service's ranking:

    - `"(bundle.symbolic.name=com.liferay.docs.l10n.myapp.lang),(bundle.symbolic.name=com.liferay.blogs.web)"`: The service aggregates resource bundles from bundles `com.liferay.docs.l10n.myapp.lang` and `com.liferay.blogs.web`. Aggregate as many bundles as desired. Listed bundles are prioritized in descending order.
    - `bundle.symbolic.name=com.liferay.blogs.web;resource.bundle.base.name="content.Language"`: Override the `com.liferay.blogs.web` bundle's resource bundle named `content.Language`.
    - `service.ranking:Long="2"`: The resource bundle's service ranking is `2`. The OSGi framework applies this service if it outranks all other resource bundle services that target `com.liferay.blogs.web`'s `content.Language` resource bundle.
    - `servlet.context.name=blogs-web`: The target resource bundle is in servlet context `blogs-web`.

Deploy your module to see the language keys you've overridden.

Tip: If your override isn't showing, use Gogo Shell to check for competing resource bundle services. It may be that another service outranks yours. To check for competing resource bundle services whose aggregates include `com.liferay.blogs.web`'s resource bundle, for example, execute this Gogo Shell command:

```
 services "(bundle.symbolic.name=com.liferay.login.web)"
```

Search the results for resource bundle aggregate services whose ranking is higher.

Now you can modify the language keys of modules in Liferay's OSGi runtime. Remember, language keys you want to override might actually be in Liferay's core. You can override global language keys too.

Related Topics

Resource Bundle Override Sample Project
Upgrading Core Language Key Hooks
Internationalization

73.10 Overriding Portal Properties using a Hook

A portal properties hook plugin lets you override a subset of portal properties dynamically. These properties define event actions, model listeners, validators, generators, and content sanitizers. The `liferay-hook-7.0.dtd` file lists this subset of properties.

Note: To customize a property that's not in the `liferay-hook-7.0.dtd` file, you must use an Ext plugin.

Some portal properties accept *multiple* values. For example, the `login.event.pre` property defines action classes to invoke before login. Deploying multiple hooks for properties like this appends the values to the property's current value. For example, multiple hooks that add login event actions append their action classes to the portal instance's `login.event.pre` property. The portal property reference documentation shows whether a property accepts multiple values by stating it or showing value lists assigned to a default or example property setting.

Some portal properties accept a *single* value only. For example, the `terms.of.use.required` property is either `true` or `false`. Override a single value property from one hook only–there's no telling which value is assigned if multiple hooks override it.

Here's how to override a portal property using a Hook:

1. Create a Hook plugin using Liferay @ide@ or Maven.

2. In the plugin's `WEB-INF/src` folder, create a `portal.properties` file and override properties with the values you want.

3. In the plugin's `WEB-INF/liferay-hook.xml` file, add the following `portal-properties` element as a child of the `hook` element. Refer to the `liferay-hook-7.0.dtd` file for details.

   ```
   <portal-properties>portal.properties</portal-properties>
   ```

4. Deploy the plugin.

You've modified the portal property. The *Server Administration* page's *Properties* screen in the Control Panel shows your new property setting.

Related Topics

Portal Properties Reference
Server Administration
Liferay @ide@
Generating New Projects using Maven Archetypes.

73.11 Overriding MVC Commands

MVC Commands are used to break up the controller layer of a Liferay MVC application into smaller, more digestible code chunks.

Sometimes you'll want to override an MVC command, whether it's in a Liferay application or another Liferay MVC application whose source code you don't own. Since MVC commands are components registered in the OSGi runtime, you can simply publish your own component, and give it a higher service ranking. Your MVC command will then be invoked instead of the original one.

The logical way of breaking up the controller layer is to do it by portlet phase. The three MVC command classes you can override are

- **`MVCActionCommand`:** An interface that allows the portlet to process a particular action request.
- **`MVCRenderCommand`:** An interface that handles the render phase of the portlet.
- **`MVCResourceCommand`:** An interface that allows the portlet to serve a resource.

Find more information about implementing each of these MVC command classes in the tutorials on Liferay MVC Portlets. Here we're going to focus on overriding the logic contained in existing MVC commands.

Note: While it's possible to copy the logic from an existing MVC command into your override class, then customize it to your liking, it's strongly recommended to decouple the original logic from your override logic. Keeping the override logic separate form the original logic will keep the code clean, maintainable, and easy to understand.

To do this, use the `@Reference` method to fetch a reference to the original MVC command component. If there are no additional customizations on the same command, this reference will be the original MVC command.

```
@Reference(
    target = "(component.name=com.liferay.blogs.web.internal.portlet.action.EditEntryMVCRenderCommand)")
protected MVCRenderCommand mvcRenderCommand;
```

Set the `component.name` target to the MVC command class name. If you use this approach, your extension will continue to work with new versions of the original portlet, because no coupling exists between the original portlet logic and your customization. The command implementation class can change. Make sure to keep your reference updated to the name of the current implementation class.

Note: In 7.0 GA1, there's a bug that occurs when modules with override MVC commands are removed from the OSGi runtime. Instead of looking for an MVC command with a lower service ranking (the original MVC command in most cases) to replace the removed one, the reference to the command is removed entirely. This bug is documented and fixed here

---

Start by learning to override `MVCRenderCommand`. The process will be similar for the other MVC commands.

Overriding MVCRenderCommand

You can override `MVCRenderCommand` for any portlet that uses Liferay's MVC framework and publishes an `MVCRenderCommand` component.

For example, Liferay's Blogs application has a class called `EditEntryMVCRenderCommand`, with this component:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS,
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_AGGREGATOR,
        "mvc.command.name=/blogs/edit_entry"
    },
    service = MVCRenderCommand.class
)
```

This MVC render command can be invoked from any of the portlets specified by the `javax.portlet.name` parameter, by calling a render URL that names the MVC command.

```
<portlet:renderURL var="addEntryURL">
    <portlet:param name="mvcRenderCommandName" value="/blogs/edit_entry" />
    <portlet:param name="redirect" value="<%= viewEntriesURL %>" />
</portlet:renderURL>
```

What if you want to override the command, but not for all of the portlets listed in the original component? In your override component, just list the `javax.portlet.name` of the portlets where you want the override to take effect. For example, if you want to override the `/blogs/edit_entry` MVC render command just for the Blogs Admin portlet (the Blogs Application accessed in the site administration section of Liferay), your component could look like this:

```
@Component(
  immediate = true,
  property = {
     "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
     "mvc.command.name=/blogs/edit_entry",
     "service.ranking:Integer=100"
  },
  service = MVCRenderCommand.class
)
```

Note the last property listed, `service.ranking`. It's used to tell the OSGi runtime which service to use, in cases where there are multiple components registering the same service, with the same properties. The higher the integer you specify here, the more weight your component carries. In this case, the override component will be used instead of the original one, since the default value for this property is `0`.

After that, it's up to you to do whatever you'd like. You can add logic to the existing `render` method or redirect to an entirely new JSP.

Adding Logic to an Existing MVC Render Command

Don't copy the existing logic from the MVC render command into your override command class. This unnecessary duplication of code that makes maintenance more difficult. If you want to do something new (like set a request attribute) and then execute the logic in the original MVC render command, obtain a reference to the original command and call its `render` method like this:

```
@Override
public String render(RenderRequest renderRequest,
                 RenderResponse renderResponse) throws PortletException {

    //Do something here

    return mvcRenderCommand.render(renderRequest, renderResponse);
}

@Reference(target =
      "(component.name=com.liferay.blogs.web.internal.portlet.action.EditEntryMVCRenderCommand)")
  protected MVCRenderCommand mvcRenderCommand;
}
```

Sometimes, you might need to redirect the request to an entirely new JSP that you'll place in your command override module.

Redirecting to a New JSP

If you want to render an entirely new JSP, the process is different.

The render method of MVCRenderCommand returns the path to a JSP as a String. The JSP must live in the original module, so you cannot simply specify a path to a custom JSP in your override module. You need to make the method skip dispatching to the original JSP altogether, by using the MVC_PATH_VALUE_SKIP_DISPATCH constant from the MVCRenderConstants class. Then you need to initiate your own dispatching process, directing the request to your JSP path. Here's how that might look in practice:

```
public class CustomEditEntryMVCRenderCommand implements MVCRenderCommand {

    @Override
    public String render
        (RenderRequest renderRequest, RenderResponse renderResponse) throws
        PortletException {

        System.out.println("Rendering custom_edit_entry.jsp");

        RequestDispatcher requestDispatcher =
            servletContext.getRequestDispatcher("/custom_edit_entry.jsp");

        try {
            HttpServletRequest httpServletRequest =
                PortalUtil.getHttpServletRequest(renderRequest);
            HttpServletResponse httpServletResponse =
                PortalUtil.getHttpServletResponse(renderResponse);

            requestDispatcher.include
                (httpServletRequest, httpServletResponse);
        } catch (Exception e) {
            throw new PortletException
                ("Unable to include custom_edit_entry.jsp", e);
        }

        return MVCRenderConstants.MVC_PATH_VALUE_SKIP_DISPATCH;
    }

    @Reference(target = "(osgi.web.symbolicname=com.custom.code.web)")
    protected ServletContext servletContext;

}
```

In this approach, there's no reference to the original MVC render command because the original logic isn't reused. Instead, there's a reference to the servlet context of your module, which is needed to use the request dispatcher.

A servlet context is automatically created for portlets. It can be created for other modules by including the following line in your bnd.bnd file:

```
Web-ContextPath: /custom-code-web
```

Once we have the servlet context we just need to dispatch to the specific JSP in our own module.

Overriding MVCActionCommand

You can override MVC action commands using a similar process to the one presented above for MVC render commands. Again, you'll register a new OSGi component with the same properties, but with a higher service ranking. This time the service you're publishing is `MVCActionCommand.class`.

For MVC action command overrides, extend the `BaseMVCActionCommand` class, and the only method you'll need to override is `doProcessAction`, which must return `void`.

As with MVC render commands, you can add your logic to the original behavior of the action method by getting a reference to the original service, and calling it after your own logic. Here's an example of an `MVCActionCommand` override that checks whether the `delete` action is invoked on a blog entry, and prints a message to the log, before continuing with the original processing:

```
@Component(
    property = {
        "javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN,
        "mvc.command.name=/blogs/edit_entry",
        "service.ranking:Integer=100"
    },
    service = MVCActionCommand.class)
public class CustomBlogsMVCActionCommand extends BaseMVCActionCommand {

    @Override
    protected void doProcessAction
        (ActionRequest actionRequest, ActionResponse actionResponse)
        throws Exception {

        String cmd = ParamUtil.getString(actionRequest, Constants.CMD);

        if (cmd.equals(Constants.DELETE)) {
            System.out.println("Deleting a Blog Entry");
        }

        mvcActionCommand.processAction(actionRequest, actionResponse);
    }

    @Reference(
        target = "(component.name=com.liferay.blogs.web.internal.portlet.action.EditEntryMVCActionCommand)")

    protected MVCActionCommand mvcActionCommand;

}
```

It's straightforward to override MVC action commands while keeping your code decoupled from the original action methods. You can also override MVC resource commands.

Overriding MVCResourceCommand

There are fewer uses for overriding MVC resource commands, but it can also be done.

The process is similar to the one described for `MVCRenderCommand` and `MVCActionCommand`. There's a couple things to keep in mind:

- The service to specify in your component is `MVCResourceCommand.class`
- As with overriding `MVCRenderCommand`, there's no base implementation class to extend. You'll implement the interface yourself.
- Keep your code decoupled from the original code by adding your logic to the original `MVCResourceCommand`'s logic by getting a reference to the original and returning a call to its `serveResource` method:

```
        return mvcResourceCommand.serveResource(resourceRequest, resourceResponse);
```

The following example overrides the behavior of `com.liferay.login.web.portlet.action.CaptchaMVCResourceCommand`, from the `login-web` module of the Login portlet. It simply prints a line in the console then executes the original logic: returning the Captcha image for the account creation screen.

```
@Component(
    property = {
        "javax.portlet.name=" + LoginPortletKeys.LOGIN,
        "mvc.command.name=/login/captcha" },
    service = MVCResourceCommand.class)
public class CustomCaptchaMVCResourceCommand implements MVCResourceCommand {

    @Override
    public boolean serveResource
        (ResourceRequest resourceRequest, ResourceResponse resourceResponse) {

        System.out.println("Serving login captcha image");

        return mvcResourceCommand.serveResource(resourceRequest, resourceResponse);
    }

    @Reference(target = "(component.name=com.liferay.login.web.internal.portlet.action.CaptchaMVCResourceCommand)")
    protected MVCResourceCommand mvcResourceCommand;

}
```

And that, as they say, is that. Even if you don't own the source code of an application, you can override its MVC commands just by knowing the component class name.

73.12 Overriding lpkg files

Applications are delivered through Liferay Marketplace as *lpkg* files. This is a simple compressed file format that contains .jar files to be deployed to Liferay DXP. If you want to examine an application from Marketplace, all you have to do is unzip it to reveal the .jar files it contains.

After examining them, you may want to customize one of these applications. Make your customization, but don't deploy it the way you'd normally deploy an application. Instead, Liferay DXP provides a way to update application modules without modifying the original .lpkg file they're packaged in, by overriding the .lpkg file. This only takes a few steps:

1. Shut down Liferay DXP.
2. Create a folder called `override` in the Liferay DXP instance's `osgi/marketplace` folder.
3. Name your updated .jar the same as the .jar in the original .lpkg, minus the version information. For example, if you're overriding the `com.liferay.amazon.rankings.web-1.0.5.jar` from the `Liferay CE Amazon Rankings.lpkg`, you'd name your .jar `com.liferay.amazon.rankings.web.jar`.
4. Copy this .jar into the `override` folder you created in step one.

This works for applications from Marketplace, but there's also the static .lpkg that contains core Liferay technology and third-party utilities (such as the servlet API, Apache utilities, etc.). If you find you need to customize or patch any of these .jar files, deploying these customizations is a similar process:

1. Make your customization and package your .jar file.

2. Name your updated .jar the same as the original .jar, minus the version information. For example, a customized `com.liferay.portal.profile-1.0.4.jar` should be `com.liferay.portal.profile.jar`.

3. Place this .jar in the `osgi/static` folder.

Now start @product. Note that any time you add and remove .jars this way, Liferay DXP must be shut down and then restarted to make the changes take effect.

If you must roll back your customizations, delete the overriding .jar files: Liferay DXP uses the original .jar on its next startup.

73.13 Creating Model Listeners

Model Listeners implement the `ModelListener` interface. They are used to listen for persistence events on models and do something in response (either before or after the event).

Model listeners were designed to perform lightweight actions in response to a `create`, `remove`, or `update` attempt on an entity's database table or a mapping table (for example, `users_roles`). Here are some supported use cases:

- Audit Listener: In a separate database, record information about updates to an entity's database table.
- Cache Clearing Listener: Clear caches that you've added to improve the performance of custom code.
- Validation Listener: Perform additional validation on a model's attribute values before they are persisted to the database.
- Entity Update Listener: Do some additional processing when an entity table is updated. For example, notify users when changes are made to their account.

There are also use cases that are not recommended, since they're likely to break unpredictably and give you headaches:

- Setting a model's attributes in an `onBeforeUpdate` call. If some other database table has already been updated with the values before your model listener is invoked, your database will get out of sync. To change how an entity's attributes are set, consider using a service wrapper instead.
- Wrapping a model. Model listeners are not called when fetching records from the database.
- Creating worker threads to do parallel processing and querying data you updated via your listener. Model listeners are called *before* the database transaction is complete (even the `onAfter...` methods), so the queries could be executed before the database transaction is completed.

If there is no existing listener on the model, your model listener is the only one that runs. However, there can be multiple listeners on a single model, and the order in which the listeners run cannot be controlled.

You can create a model listener in a module by doing two simple things:

- Implement `ModelListener`
- Register the service in Liferay's OSGi runtime

Creating a Model Listener Class

Create a `-ModelListener` class that extends `BaseModelListener`.

```
package ...;

import ...;

public class CustomEntityListener extends BaseModelListener<CustomEntity> {

    /* Override one or more methods from the ModelListener
        interface.
    */

}
```

In the body of the class override any methods from the `ModelListener` interface. The available methods are listed and described at the end of this article.

In your model listener class, the parameterized type (for example, `CustomEntity` in the snippet above) is used to tell the listener's `ServiceTrackerCustomizer` which model class the listener should be registered against.

Register the Model Listener Service

Register the service with Liferay's OSGi runtime. If using Declarative Services, set `service= ModelListener.class` and `immediate=true` in the Component.

```
@Component(
    immediate = true,
    service = ModelListener.class
)
```

That's all there is to preparing a model listener. Now learn what model events you can respond to.

Listening For Persistence Events

The `ModelListener` interface provides lots of opportunity to listen for model events:

- **`onAfterAddAssociation`:** If there's an association between two models (if they have a mapping table), use this method to do something after an association record is added.
- **`onAfterCreate`:** Use this method to do something after the persistence layer's `create` method is called.
- **`onAfterRemove`:** Use this method to do something after the persistence layer's `remove` method is called.
- **`onAfterRemoveAssociation`:** If there's an association between two models (if they have a mapping table), do something after an association record is removed.
- **`onAfterUpdate`:** Use this method to do something after the persistence layer's `update` method is called.
- **`onBeforeAddAssociation`:** If there's an association between two models (if they have a mapping table), do something before an addition to the mapping table.
- **`onBeforeCreate`:** Use this method to do something before the persistence layer's `create` method is called.
- **`onBeforeRemove`:** Use this method to do something before the persistence layer's `remove` method is called.
- **`onBeforeRemoveAssociation`:** If there's an association between two models (if they have a mapping table), do something before a removal from the mapping table.
- **`onBeforeUpdate`:** Use this method to do something before the persistence layer's `update` method is called.

Look in Liferay's `BasePersistenceImpl`, particularly the `remove` and `update` methods, and you'll see how model listeners are accounted for before (for the `onBefore...` case) and after (for the `onAfter...` case) the model persistence event.

Now that you know how to create model listeners, keep in mind that they're useful as standalone projects or inside of your application. If your application needs to do something (like add a custom entity) every time a User is added in Liferay, you can include the model listener inside your application.

Related Topics

Upgrading Model Listener Hooks

Chapter 74

Application Display Templates

The application display template (ADT) framework allows portal administrators to override the default display templates, removing limitations to the way your site's content is displayed. With ADTs, you can define custom display templates used to render asset-centric applications.

Usually, when you need to modify the UI of a Liferay portlet, you can do so using a hook (e.g., HTML-related change) or a theme (e.g., CSS-related change). It'd be nice, however, if you could apply particular display changes to specific portlet instances without having to redeploy any plugins. Ideally, you should be able to provide authorized portal users the ability to apply custom display interfaces to portlets.

Be of good cheer! That's precisely what Application Display Templates (ADTs) provide–the ability to add custom display templates to your portlets from the portal. This isn't actually a new concept in Liferay; some applications already had templating capabilities (e.g., *Web Content* and *Dynamic Data Lists*), in which you can already add as many display options (or templates) as you want. Now you can add them to your custom portlets, too.

Some portlets that already support Application Display Templates in 7.0 are *Asset Catgories Navigation*, *Asset Publisher*, *Asset Tags Navigation*, *Blogs*, *Media Gallery*, *RSS*, *Breadcrumb*, *Language*, *Navigation Menu*, *SiteMap*, and *Wiki*.

74.1 Implementing Application Display Templates

Application Display Templates (ADTs) provide–the ability to add custom display templates to your portlets from the portal. The figure below shows what the Display Template option looks like in a portlet Configuration menu.

In this tutorial, you'll learn how to use the Application Display Templates API to add an ADT to a portlet.

Using the Application Display Templates API

To leverage the ADT API, there are several steps you need to follow. These steps involve registering your portlet to use ADTs, defining permissions, and exposing the ADT functionality to users. You'll walk through these steps now:

1. Create and register a custom `*PortletDisplayTemplateHandler` component. Liferay provides the BasePortletDisplayTemplateHandler as a base implementation for you to extend. You can check the TemplateHandler interface Javadoc to learn about each template handler method.

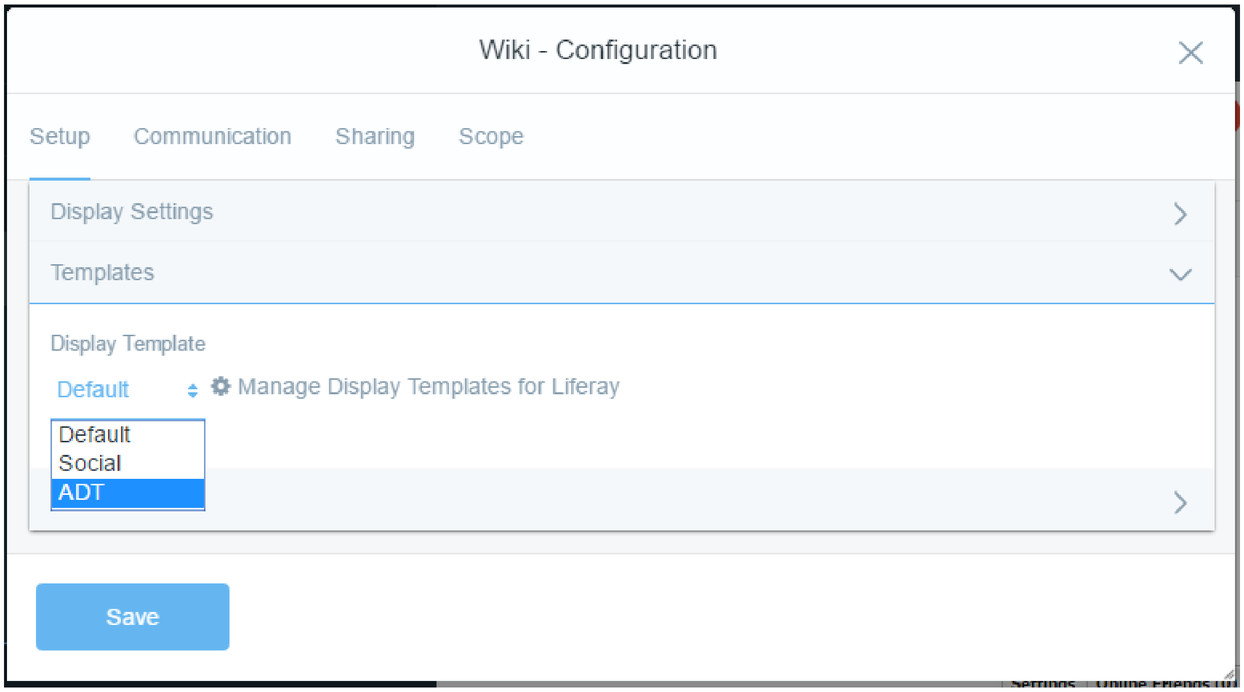


Figure 74.1: By using a custom display template, your portlet's display can be customized.

The Component annotation ties your handler to a specific portlet setting the property `javax.portlet.name` as the portlet name of your portlet. The same property should be found in your portlet class. For example:

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name="+ AssetCategoriesNavigationPortletKeys.ASSET_CATEGORIES_NAVIGATION
    },
    service = TemplateHandler.class
)
```

Each of the methods in this class have a significant role in defining and implementing ADTs for your custom portlet. View the list below for a detailed explanation for each method defined specifically for ADTs:

- **getClassName():** Defines the type of entry your portlet is rendering.
- **getName():** Declares the name of your ADT type (typically, the name of the portlet).
- **getResourceName():** Specifies which resource is using the ADT (e.g., a portlet) for permission checking. This method must return the portlet's Fully Qualified Portlet ID (FQPI).
- **getTemplateVariableGroups():** Defines the variables exposed in the template editor.

As an example `*PortletDisplayTemplateHandler` implementation, you can look at WikiPortletDisplayTemplateHandler.java.

2. Since the ability to add ADTs is new to your portlet, you must configure permissions so that administrative users can grant permissions to the roles that will be allowed to create and manage display templates.

Add the action key `ADD_PORTLET_DISPLAY_TEMPLATE` to your portlet's `/src/main/resources/resource-actions/default.xml` file:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action Mapping 7.0.0//EN" "http://www.liferay.com/dtd/liferay-
resource-action-mapping_7_0_0.dtd">
<resource-action-mapping>

    ...

    <portlet-resource>
        <portlet-name>yourportlet</portlet-name>
        <permissions>
            <supports>
                <action-key>ADD_PORTLET_DISPLAY_TEMPLATE</action-key>
                <action-key>ADD_TO_PAGE</action-key>
                <action-key>CONFIGURATION</action-key>
                <action-key>VIEW</action-key>
            </supports>

            ...

        </permissions>
    </portlet-resource>

    ...

</resource-action-mapping>
```

3. Next, you need to make sure that Liferay can find the updated `default.xml` with the new resource action when you deploy the module. Create a file named `portlet.prtoperies` in the `/resources` folder and add the following contents providing the path to your `default.xml`:

```
include-and-override=portlet-ext.properties
resource.actions.configs=resource-actions/default.xml
```

4. Now that your portlet officially supports ADTs, you'll want to expose the ADT option to your users. Just include the `<liferay-ui:ddm-template-selector>` tag in the JSP file you're using to control your portlet's configuration.

   For example, it may be helpful for you to insert an `<aui:fieldset>` in your configuration JSP file, like the following:

```
<aui:fieldset>
    <div class="display-template">
        <liferay-ddm:template-selector
            classNameId="<%= YourEntity.class.getName() %>"
            displayStyle="<%= displayStyle %>"
            displayStyleGroupId="<%= displayStyleGroupId %>"
            refreshURL="<%= PortalUtil.getCurrentURL(request) %>"
            showEmptyOption="<%= true %>"
        />
    </div>
</aui:fieldset>
```

   In this JSP, the `<liferay-ddm:template-selector>` tag specifies the Display Template drop-down menu to be used in the portlet's Configuration menu. The variables `displayStyle` and `displayStyleGroupId` are preferences that your portlet stores when you use this taglib and your portlet uses the BaseJSPSettingsConfigurationAction or DefaultConfigurationAction. Otherwise, you would need to obtain the value of those parameters and store them manually in your configuration class.

As an example JSP, see the Wiki application's configuration.jsp.

5. You must now extend your view code to render your portlet with the selected ADT. This allows you to decide which part of your view will be rendered by the ADT and what will be available in the template context.

   First, initialize the Java variables needed for the ADT:

```
<%
String displayStyle = GetterUtil.getString(portletPreferences.getValue("displayStyle", StringPool.BLANK));
long displayStyleGroupId = GetterUtil.getLong(portletPreferences.getValue("displayStyleGroupId", null), scopeGroupId);
%>
```

   Next, you can test if the ADT is configured, grab the entities to be rendered, and render them using the ADT. The tag `<liferay-ddm:template-renderer>` aids with this process. It will automatically use the selected template or render its body if no template is selected.

   Here's some example code that demonstrates implementing this:

```
<liferay-ddm:template-renderer
    className="<%= YourEntity.class.getName() %>"
    contextObjects="<%= contextObjects %>"
    displayStyle="<%= displayStyle %>"
    displayStyleGroupId="<%= displayStyleGroupId %>"
    entries="<%= yourEntities %>"
>

    <%-- The code that will be rendered by default when there is no
    template available should be inserted here. --%>

</liferay-ddm:template-renderer>
```

   In this step, you initialized variables dealing with the display settings (`displayStyle` and `displayStyleGroupId`) and passed them to the tag along with other parameterers listed below:

   - `className`: your entity's class name.
   - `contextObjects`: accepts a `Map<String, Object>` with any object you want to the template context.
   - `entries`: accepts a list of your entities (e.g., `List<YourEntity>`).

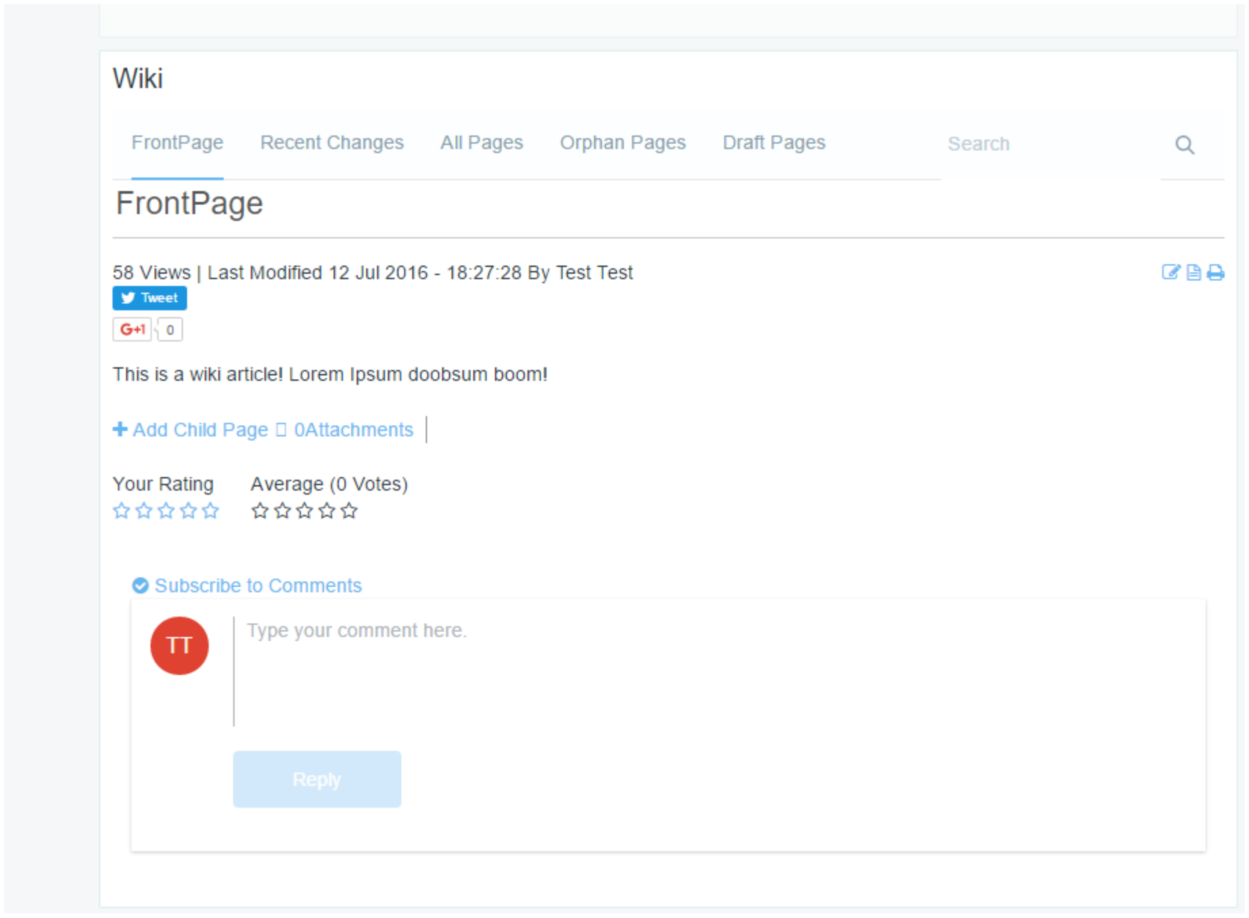
   For an example that demonstrates implementing this, see configuration.jsp.

Now that your portlet supports ADTs, you can create your own scripts to change the display of your portlet. You can experiment by adding your own custom ADT.

1. Navigate to Site Admin* → *Configuration* → *Application Display Templates*. Then select *Add* (+) → *Your Template*. Give your ADT a name and insert FreeMarker (like the following code) or Velocity code into the template editor, and click *Save*:

```
<#if entries?has_content>
    Quick List:
    <ul>
    <#list entries as curEntry>
        <li>${curEntry.name} - ${curEntry.streetAddress}, ${curEntry.city}, ${curEntry.stateOrProvince}</li>
    </#list>
    </ul>
</#if>
```

2. Go back to your portlet and select *Options* (⋮) → *Configuration* and click the *Display Template* drop-down. Select the ADT you created, and click *Save*.



FrontPage

58 Views | Last Modified 12 Jul 2016 - 18:27:28 By Test Test

This is a wiki article! Lorem Ipsum doobsum boom!

Figure 74.2: The example Social template for the Wiki application provides extended social functionalities.

Once your script is uploaded into the portal and saved, users with the specified roles can select the template when they're configuring the display settings of your portlet on a page. You can visit the Styling Apps with Application Display Templates section for more details on using ADTs.

Next, we'll provide some recommendations for using ADTs in Liferay Portal.

Recommendations for Using ADTs

You've harnessed a lot of power by learning to leverage the ADT API. Be careful, for with great power, comes great responsibility! To that end, you'll learn about some practices you can use to optimize your portlet's performance and security.

First let's talk about security. You may want to hide some classes or packages from the template context, to limit the operations that ADTs can perform on your portal. Liferay provides some portal system settings, which can be accessed by navigating to *Control Panel* → *Configuration* → *System Settings* → *Foundation* → *FreeMarker/Velocity Engine*, to define the restricted classes, packages, and variables. In particular, you may want to add `serviceLocator` to the list of default values assigned to the FreeMarker and Velocity Engine Restricted variables.

Application Display Templates introduce additional processing tasks when your portlet is rendered. To minimize negative effects on performance, make your templates as minimal as possible by focusing on the presentation, while using the existing API for complex operations. The best way to make Application Display Templates efficient is to know your template context well, and understand what you can use from it. Fortunately, you don't need to memorize the context information, thanks to Liferay's advanced template editor!

To navigate to the template editor for ADTs, go to the Site Admin menu and select *Configuration* → *Application Display Templates* and then click *Add* and select the specific portlet on which you decide to create an ADT.

The template editor provides fields, general variables, and util variables customized for the portlet you chose. These variable references can be found on the left-side panel of the template editor. You can use them by simply placing your cursor where you'd like the variable placed, and clicking the desired variable to place it there. You can learn more about the template editor in the Styling Apps with Application Display Templates section.

Finally, don't forget to run performance tests and tune the template cache options by modifying the *Resource modification check* field in *System Settings* → *Foundation* → *FreeMarker/Velocity Engine*.

The cool thing about ADTs is the power they provide to your Liferay portlets, providing infinite ways of editing your portlet to provide new interfaces for your portal users. You stepped through how to configure ADTs for a custom portlet, tried out a sample template, and ran through important recommendations for using ADTs, which included security and performance.

Related Topics

Styling Apps with Application Display Templates

Liferay JavaScript APIs

Internationalization

Chapter 75

Mobile

Liferay provides two ways to create native Android and iOS apps that work with your Liferay instances: Liferay Screens and the Liferay Mobile SDK. Liferay Screens does this via ready-to-use components called *Screenlets*. Since Screenlets already contain the code required to call your Liferay instance–and a complete UI–all you need to do is insert and configure them in your Android or iOS app. Screens provides Screenlets for common tasks such as logging in, viewing web content, adding DDL records, and more. You can also customize each Screenlet to fit your specific needs, or write your own Screenlet. Behind the scenes, Screenlets use the Liferay Mobile SDK to call Liferay's remote services.

The Liferay Mobile SDK is a lower-level tool that lets you manually invoke Liferay's remote services. You'll need to use the Mobile SDK to write your own Screenlets, or call Liferay's remote services independent of Screens. In most cases, you'll find that using Screens saves you time and effort. For example, although you can use the Mobile SDK to implement login in your app, Screens already provides this via Login Screenlet. There are certain cases, however, where using the Mobile SDK makes sense. For example, if you need to call one or more Liferay remote services but your app's UI doesn't need to reflect this, then it doesn't make sense to use Screenlets for this purpose. Each Screenlet must contain a UI.

Regardless of your specific needs, Liferay has you covered with Liferay Screens and the Liferay Mobile SDK. This section of tutorials contains the following sections that show you how to use both:

- Android Apps with Liferay Screens
- iOS Apps with Liferay Screens
- Using Xamarin with Liferay Screens
- Liferay Mobile SDK

Venture forth to become a mobile guru!

Chapter 76

Android Apps with Liferay Screens

Liferay Screens speeds up and simplifies developing native mobile apps that use Liferay. Its power lies in its *Screenlets*. A Screenlet is a visual component that you insert into your native app to leverage Liferay Portal's content and services. On Android, Screenlets are available to log in to your portal, create accounts, submit forms, display content, and more. You can use any number of Screenlets in your app; they're independent, so you can use them in modular fashion. Screenlets on Android also deliver UI flexibility with pluggable *Views* that implement elegant user interfaces. Liferay's reference documentation for Android Screenlets describes each Screenlet's features and Views.

You might be thinking, "These Screenlets sound like the greatest thing since taco Tuesdays, but what if they don't fit in with my app's UI? What if they don't behave exactly how I want them to? What if there's no Screenlet for what I want to do?" Fret not! You can customize Screenlets to fit your needs by changing or extending their UI and behavior. You can even write your own Screenlets! What's more, Screens seamlessly integrates with your existing Android projects.

Screenlets leverage the Liferay Mobile SDK to make server calls. The Mobile SDK is a low-level layer on top of the Liferay JSON API. To write your own Screenlets, you must familiarize yourself with Liferay's remote services. If no existing Screenlet meets your needs, consider customizing an existing Screenlet, creating a Screenlet, or directly using the Mobile SDK. Creating a Screenlet involves writing Mobile SDK calls and constructing the Screenlet; if you don't plan to reuse or distribute the implementation then you may want to forgo writing a Screenlet and, instead, work with the Mobile SDK. A benefit of integrating an existing Screenlet into your app, however, is that the Mobile SDK's details are abstracted from you.

These tutorials show you how to use, customize, create, and distribute Screenlets for Android. They show you how to create Views too. There's even a tutorial that explains the nitty-gritty details of the Liferay Screens architecture. No matter how deep you want to go, you'll use Screenlets in no time. Start by preparing your Android project to use Liferay Screens.

76.1 Preparing Android Projects for Liferay Screens

To use Liferay Screens, you must install it in your Android project and then configure it to communicate with your Liferay DXP instance. Note that Screens is released as an AAR file hosted in jCenter.

There are three different ways to install Screens. This tutorial shows you each:

1. With Gradle: Gradle is the build system Android Studio uses to build Android projects. We therefore recommend that you use it to install Screens.

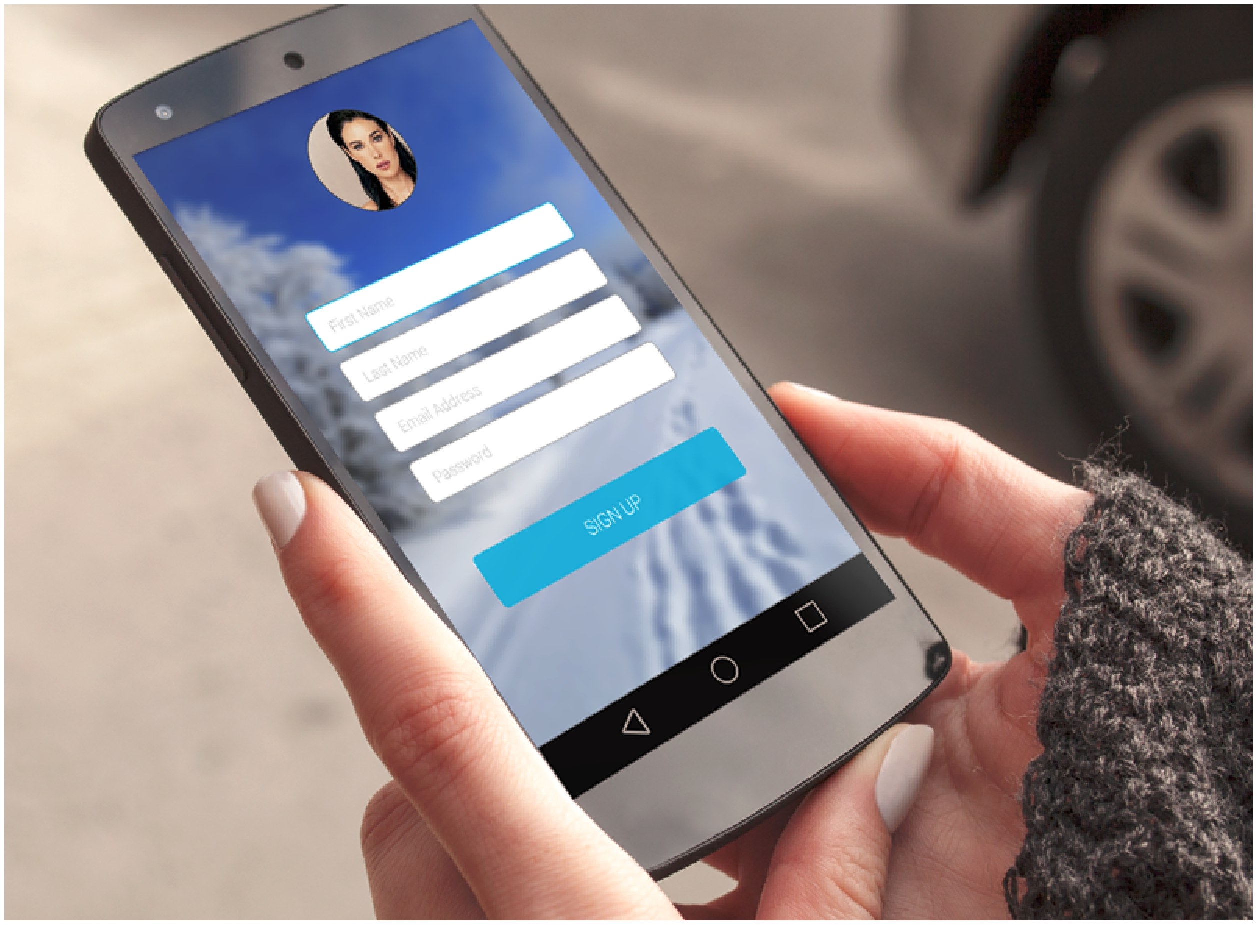


Figure 76.1: Here's an app that uses a Liferay Screens Sign Up Screenlet.

2. With Maven
3. Manually

Note: After installation, you must configure Liferay Screens to communicate with your Liferay DXP instance. The last section in this tutorial shows you how to do this.

Requirements

Liferay Screens for Android includes the Component Library (the Screenlets) and a sample project. It requires the following software:

- Android Studio 3.0 or above.
- Android SDK 4.1 (API Level 16) or above.
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, or Liferay DXP 7.0
- Liferay Screens Compatibility Plugin (CE or DXP/EE, depending on your portal edition). This app is preinstalled in Liferay CE Portal 7.0/7.1 CE and Liferay DXP 7.0.
- Liferay Screens source code.

Liferay Screens for Android uses EventBus internally.

Securing JSON Web Services

Each Screenlet in Liferay Screens calls one or more of Liferay DXP's JSON web services, which are enabled by default. The Screenlet reference documentation lists the web services that each Screenlet calls. To use a Screenlet, its web services must be enabled in the portal. It's possible, however, to disable the web services needed by Screenlets you're not using. For instructions on this, see the tutorial Configuring JSON Web Services. You can also use Service Access Policies for more fine-grained control over accessible services.

Using Gradle to Install Liferay Screens

To use Gradle to install Liferay Screens in your Android Studio project, you must edit your app's `build.gradle` file. Note that your project has two `build.gradle` files: one for the project and another for the app module. You can find them under Gradle Scripts in your Android Studio project. This screenshot highlights the app module's `build.gradle` file:

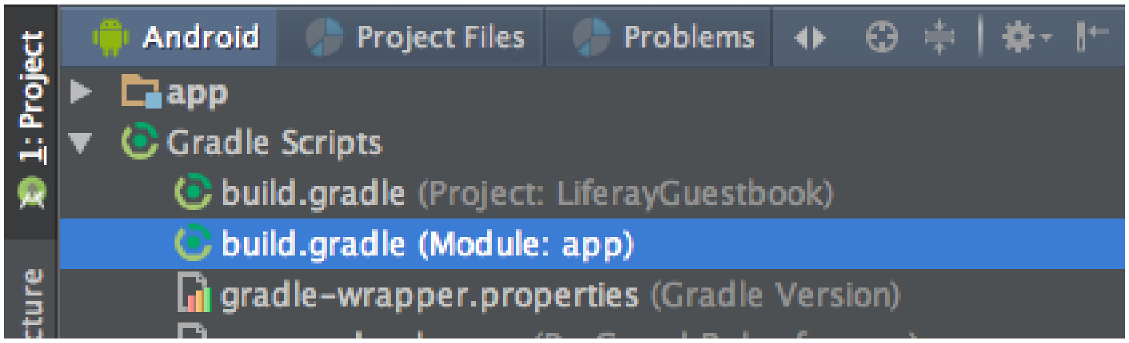

1: Project
Android
Project Files
Problems
app
Gradle Scripts
build.gradle (Project: LiferayGuestbook)
build.gradle (Module: app)
gradle-wrapper.properties (Gradle Version)

Figure 76.2: The app module's `build.gradle` file.

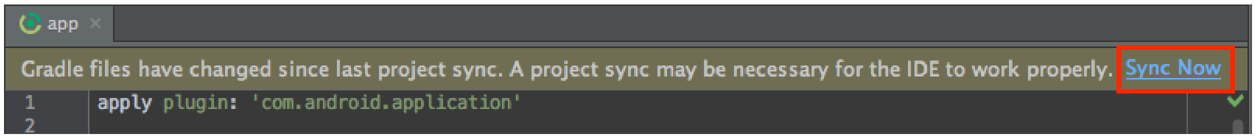
In the app module's `build.gradle` file, add the following line of code inside the `dependencies` element:

```
implementation 'com.liferay.mobile:liferay-screens:+'
```

Note that the `+` symbol tells Gradle to install the newest version of Screens. If your app relies on a specific version of Screens, you can replace the `+` symbol with that version.

If you're not sure where to add the above lines, see the below screenshot.

Once you edit `build.gradle`, a message appears at the top of the file that asks you to *sync* your app with its Gradle files. Syncing the Gradle files incorporates the changes you make to them. Syncing also downloads and installs any new dependencies, like the Liferay Screens dependency that you just added. Sync the Gradle files now by clicking the *Sync Now* link in the message. The following screenshot shows the top of an edited `build.gradle` file with the Sync Now link highlighted by a red box:



Gradle files have changed since last project sync. A project sync may be necessary for the IDE to work properly.

```
apply plugin: 'com.android.application'
```

Figure 76.3: After editing the app module's `build.gradle` file, click *Sync Now* to incorporate the changes in your app.

In the case of conflict with the `appcompat-v7` or other support libraries (`com.android.support:appcompat-v7`, `com.android.support:support-v4`), you have several options:

- Explicitly add the versions of the conflicting libraries you want to use. For example:

  ```
  implementation 'com.android.support:design:27.0.2'
  implementation 'com.android.support:support-media-compat:27.0.2'
  implementation 'com.android.support:exifinterface:27.0.2'
  ```

- Remove the `com.android.support:appcompat-v7` dependency from your project and use the one embedded in Liferay Screens.
- Exclude the problematic library from Liferay Screens. For example:

  ```
  implementation ('com.liferay.mobile:liferay-screens:+') {
      exclude group: 'com.android.support:', module: 'design'
  }
  ```

- Ignore the inspection, adding a comment like this:

  ```
  //noinspection GradleCompatible
  ```

- Ignore the warning–Liferay Screens will work without problems.

Although we strongly recommend that you use Gradle to install Screens, the following section shows you how to install Screens with Maven.

Using Maven to Install Liferay Screens

Note that we strongly recommend that you use Gradle to install Screens. It's possible though to use Maven to install Screens. Follow these steps to configure Liferay Screens in a Maven project:

1. Add the following dependency to your `pom.xml`:

   ```
   <dependency>
       <groupId>com.liferay.mobile</groupId>
       <artifactId>liferay-screens</artifactId>
       <version>LATEST</version>
   </dependency>
   ```

2. Force a Maven update to download all the dependencies.

If Maven doesn't automatically locate the artifact, you must add jCenter as a new repository in your maven settings (e.g., `.m2/settings.xml` file):

```
<profiles>
    <profile>
        <repositories>
            <repository>
                <id>bintray-liferay-liferay-mobile</id>
                <name>bintray</name>
                <url>http://dl.bintray.com/liferay/liferay-mobile</url>
            </repository>
        </repositories>
        <pluginRepositories>
```

```
            <pluginRepository>
                <id>bintray-liferay-liferay-mobile</id>
                <name>bintray-plugins</name>
                <url>http://dl.bintray.com/liferay/liferay-mobile</url>
            </pluginRepository>
        </pluginRepositories>
        <id>bintray</id>
    </profile>
</profiles>
<activeProfiles>
    <activeProfile>bintray</activeProfile>
</activeProfiles>
```

Nice work!

Manual Configuration in Gradle

Although we strongly recommend that you use Gradle to install Screens automatically, it's possible to use Gradle to install Screens manually. Follow these steps to use Gradle to install Screens and its dependencies manually in your Android project:

1. Download the latest version of Liferay Screens for Android.

2. Copy the contents of `Android/library` into a folder outside your project.

3. In your project, configure a `settings.gradle` file with the paths to the library folders:

   ```
   include ':core'
   project(':core').projectDir = new File(settingsDir, '../../library/core')
   project(':core').name = 'liferay-screens'
   ```

4. Include the required dependencies in your `build.gradle` file:

   ```
   implementation project(':liferay-screens')
   ```

You can also configure the `.aar` binary files (in `Android/dist`) as local `.aar` file dependencies. You can download all necessary files from jCenter.

To check your configuration, you can compile and execute a blank activity and import a Liferay Screens class (like Login Screenlet).

Next, you'll set up communication with Liferay DXP.

Configuring Communication with Liferay DXP

Before using Liferay Screens, you must configure it to communicate with your Liferay DXP instance. To do this, you must provide Screens the following information:

- Your Liferay DXP instance's ID

- The ID of the site your app needs to communicate with

- Your Liferay DXP instance's version

- Any other information required by specific Screenlets

Fortunately, this is straightforward. In your Android project's res/values folder, create a new file called `server_context.xml`. Add the following code to the new file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <!-- Change these values for your Liferay DXP installation -->

    <string name="liferay_server">http://10.0.2.2:8080</string>

    <string name="liferay_company_id">10155</string>
    <string name="liferay_group_id">10182</string>

    <integer name="liferay_portal_version">70</integer>

</resources>
```

As the above comment indicates, make sure to change these values to match your Liferay DXP instance. The server address `http://10.0.2.2:8080` is suitable for testing with Android Studio's emulator, because it corresponds to `localhost:8080` through the emulator. If you're using the Genymotion emulator, you should, however, use address `192.168.56.1` instead of `localhost`.

The `liferay_company_id` value is your Liferay DXP instance's ID. You can find it in your Liferay DXP instance at *Control Panel* → *Configuration* → *Virtual Instances*. The instance's ID is in the *Instance ID* column. Copy and paste this value into the `liferay_company_id` value in `server_context.xml`.

The `liferay_group_id` value is the ID of the site your app needs to communicate with. To find this value, first go to the site in your Liferay DXP instance that you want your app to communicate with. In the *Site Administration* menu, select *Configuration* → *Site Settings*. The site ID is listed at the top of the *General* tab. Copy and paste this value into the `liferay_group_id` value in `server_context.xml`.

The `liferay_portal_version` value `70` tells Screens that it's communicating with a Liferay CE Portal 7.0 or Liferay DXP 7.0 instance. Here are the supported `liferay_portal_version` values and the portal versions they correspond to:

- `71`: Liferay CE Portal 7.1 or Liferay DXP 7.1
- `70`: Liferay CE Portal 7.0 or Liferay DXP 7.0
- `62`: Liferay Portal 6.2 CE/EE

You can also configure Screenlet properties in your `server_context.xml` file. The example properties listed below, `liferay_recordset_id` and `liferay_recordset_fields`, enable DDL Form Screenlet and DDL List Screenlet to interact with a Liferay DXP instance's DDLs. You can see an additional example `server_context.xml` file here.

```
<!-- Change these values for your Liferay DXP installation -->

<string name="liferay_recordset_id">20935</string>
<string name="liferay_recordset_fields">Title</string>
```

Super! Your Android project's ready for Liferay Screens.

Example Apps

As you use Screens to develop your apps, you may want to refer to some example apps that also use it. There are two demo applications available:

- test-app: A showcase app containing all the currently available Screenlets.

- Westeros Bank: An example app that uses Screenlets to manage technical issues for the *Westeros Bank*. It's also available in Google Play.

Great! Now you're ready to put Screens to use. The following tutorials show you how to do this.

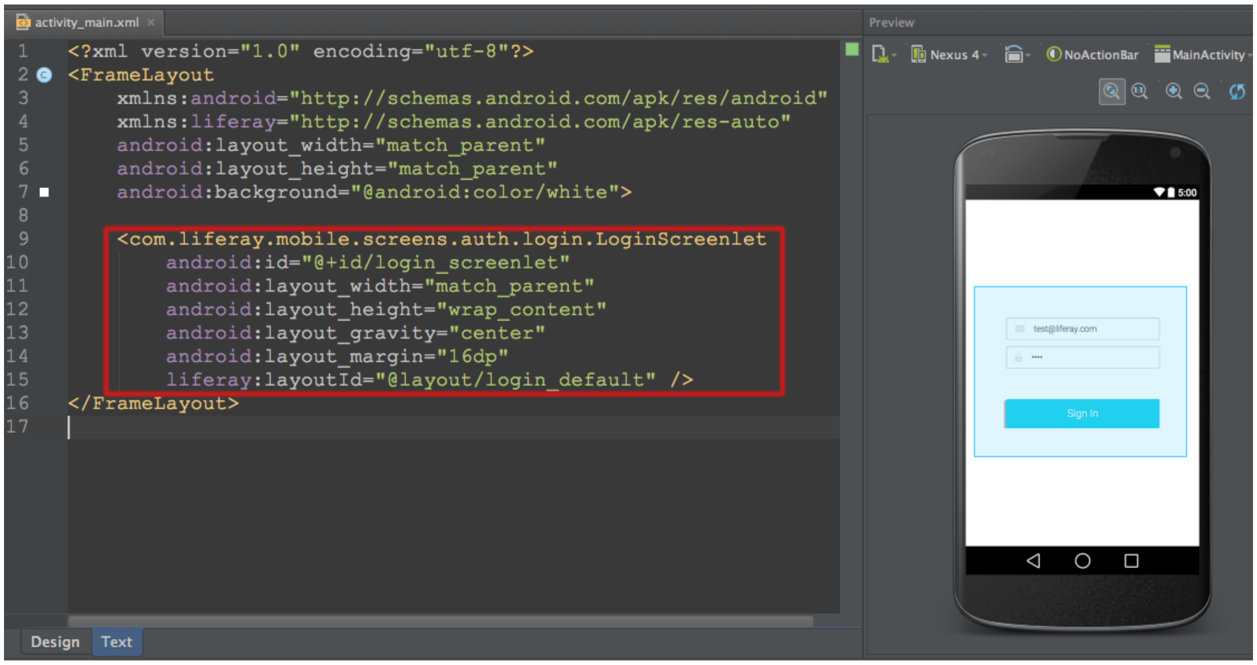
Related Topics

Using Screenlets in Android Apps
Using Views in Android Screenlets
Preparing iOS Projects for Liferay Screens

76.2 Using Screenlets in Android Apps

You can start using Screenlets once you've prepared your project to use Liferay Screens. There are plenty of Liferay Screenlets available and they're described in the Screenlet reference documentation. It is very straightforward to use Screenlets. This tutorial shows you how to insert Screenlets into your android app and configure them. You'll be a Screenlet master in no time!

First, in Android Studio's visual layout editor or your favorite editor, open your app's layout XML file and insert the Screenlet in your activity or fragment layout. The following screenshot, for example, shows the Login Screenlet inserted in an activity's `FrameLayout`.

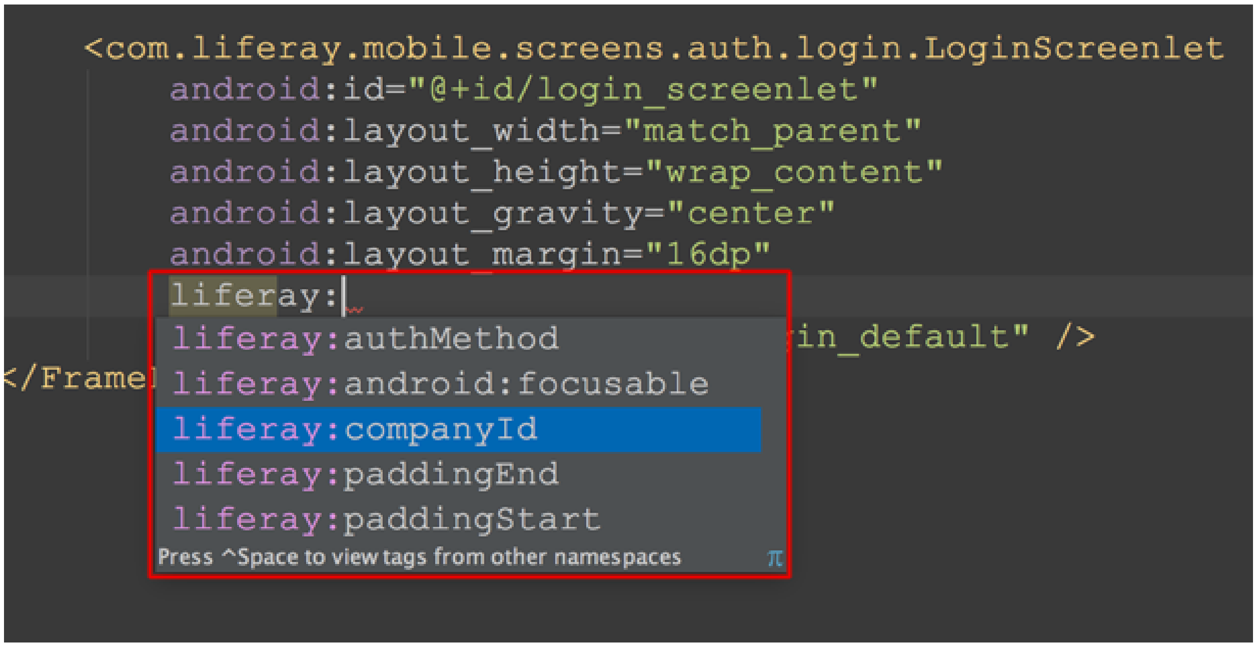


```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:liferay="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/white">

    <com.liferay.mobile.screens.auth.login.LoginScreenlet
        android:id="@+id/login_screenlet"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:layout_margin="16dp"
        liferay:layoutId="@layout/login_default" />
</FrameLayout>
```

Figure 76.4: Here's the Login Screenlet in an activity's layout in Android Studio.

Next, set the Screenlet's attributes. If it's a Liferay Screenlet, refer to the Screenlet reference documentation to learn the Screenlet's required and supported attributes. This screenshot shows the attributes of the Login Screenlet being set:

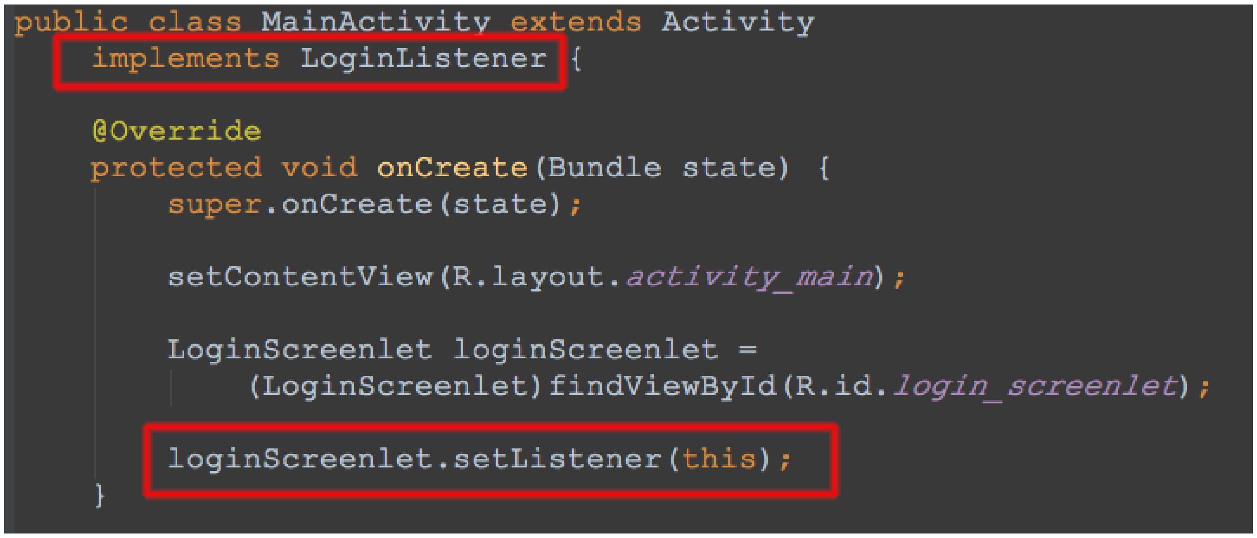
To configure your app to listen for events the Screenlet triggers, implement the Screenlet's listener interface in your activity or fragment class. Refer to the Screenlet's documentation to learn its listener



```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:layout_margin="16dp"
    liferay:
    liferay:authMethod                      in_default" />
</Frame liferay:android:focusable
    liferay:companyId
    liferay:paddingEnd
    liferay:paddingStart
    Press ^Space to view tags from other namespaces
```

Figure 76.5: You can set a Screenlet's attributes via the app's layout XML file.

interface. Then register your activity or fragment as the Screenlet's listener. The activity class, for example, in the screenshot below, declares that it implements the Login Screenlet's `LoginListener` interface, and it registers itself to listen for the Screenlet's events.



```
public class MainActivity extends Activity
    implements LoginListener {

    @Override
    protected void onCreate(Bundle state) {
        super.onCreate(state);

        setContentView(R.layout.activity_main);

        LoginScreenlet loginScreenlet =
            (LoginScreenlet)findViewById(R.id.login_screenlet);

        loginScreenlet.setListener(this);
    }
```

Figure 76.6: Implement the Screenlet's listener in your activity or fragment class.

Make sure to implement all methods required by the Screenlet's listener interface. For Liferay's Screenlets, the listener methods are listed in each Screenlet's reference documentation. That's all there is to it! Awesome! Now you know how to use Screenlets in your Android apps.

Related Topics

Preparing Android Projects for Liferay Screens
Using Views in Android Screenlets
Creating Android Screenlets
Using Screenlets in iOS apps

76.3 Using Views in Android Screenlets

You can use a Liferay Screens *View* to set a Screenlet's look and feel independent of the Screenlet's core functionality. Liferay's Screenlets come with several Views, and more are being developed by Liferay and the community. The Screenlet reference documentation lists the Views available for each Screenlet included with Screens. This tutorial shows you how to use Views in Android Screenlets. It's straightforward; you'll master using Views in no time!

Views and View Sets

Before using Views, you should know what components make them up. Note that what follows is a simple description, sufficient for learning how to use different Views. For a detailed description of the View layer in Liferay Screens, see the tutorial Architecture of Liferay Screens for Android.

A View consists of the following items:

Screenlet class: A Java class that coordinates and implements the Screenlet's functionality. The Screenlet class contains attributes for configuring the Screenlet's behavior, a reference to the Screenlet's View class, methods for invoking server operations, and more.

View class: A Java class that implements a View's behavior. This class usually listens for the UI components' events.

Layout: An XML file that defines a View's UI components. The View class is usually this file's root element. To use a View, you must specify its layout in the Screenlet XML (you'll see an example of this shortly).

Note that because it contains a Screenlet class and a specific set of UI components, a View can only be used with one particular Screenlet. For example, the Default View for Login Screenlet can only be used with Login Screenlet. Multiple Views for several Screenlets can be combined into a *View Set*. A View Set typically implements a similar look and feel for several Screenlets. This lets an app use a View Set to present a cohesive look and feel. For example, the Bank of Westeros sample app uses the Westeros View Set's Views with several Screenlets to present the red and white motif you can see here on Google Play. Liferay Screens for Android comes with the Default View Set, but Liferay makes additional View Sets, like Material and Westeros, available in jCenter. Anyone can create View Sets and publish them in public repositories like Maven Central and jCenter.

To install View Sets besides Default, add them as dependencies in your project. The `build.gradle` file code snippet below specifies the Material and Westeros View Sets as dependencies:

```
dependencies {
    ...
    implementation 'com.liferay.mobile:liferay-material-viewset:+'
    implementation 'com.liferay.mobile:liferay-westeros-viewset:+'
    ...
}
```

Here are the View Sets that Liferay currently provides for Screens:

Default: Comes standard with a Screenlet. It's used by a Screenlet if no layout ID is specified or if no View is found with the layout ID. The Default Views can be used as parent Views for your custom Views. Refer to the architecture tutorial for more details.

Material: Demonstrates Views built from scratch. It follows Google's Material Design guidelines. Refer to the View creation tutorial for instructions on creating your own Views.

Westeros: Customizes the behavior and appearance of the Westeros Bank demo app.

Now that you know about Views and View Sets, you're ready to put them to use!

Using Views

To use a View in a Screenlet, specify the View's layout as the `liferay:layoutId` attribute's value when inserting the Screenlet XML in an activity or fragment layout. For example, to use Login Screenlet with its Material View, insert the Screenlet's XML with `liferay:layoutId` set to `@layout/login_material`:

```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    liferay:layoutId="@layout/login_material"
    />
```

The following links list the View layouts available in each View Set:

- Default
- Material
- Westeros

If the View you want to use is part of a View Set, your app or activity's theme must also inherit the theme that defines that View Set's styles. For example, the following code in an app's `res/values/styles.xml` tells `AppTheme.NoActionBar` to use the Material View Set as its parent theme:

```
<resources>

    <style name="AppTheme.NoActionBar" parent="material_theme">
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>

        <item name="windowActionBar">false</item>
        <item name="windowNoTitle">true</item>
    </style>
    ...
</resources>
```

To use the Default or Westeros View Set, inherit `default_theme` or `westeros_theme`, respectively.

That's it! Great! Now you know how to use Views to spruce up your Android Screenlets. This opens up a world of possibilities, like writing your own Views.

Related Topics

Preparing Android Projects for Liferay Screens

Using Screenlets in Android Apps

Creating Android Views

Architecture of Liferay Screens for Android

Using Themes in iOS Screenlets

76.4 Using Offline Mode in Android

Offline mode in Liferay Screens lets your apps function when connectivity is unavailable or intermittent. Even though the steady march of technology makes connections more stable and prevalent, there are still plenty of places the Internet has trouble reaching. Areas with complex terrain, including cities with large buildings, often lack stable connections. Remote areas typically don't have connections at all. Using Screens's offline mode in your apps gives your users flexibility in these situations.

This tutorial shows you how to use offline mode in Screenlets. For an explanation of how offline mode works, see the tutorial Architecture of Offline Mode in Liferay Screens. Offline mode's architecture is the same on iOS and Android, although its use on these platforms differs.

Configuring Screenlets for Offline Mode

If you want to enable offline mode in any of your screenlets, you must configure the `offlinePolicy` attribute when inserting the Screenlet's XML in a layout. This attribute can take four possible values:

- `REMOTE_ONLY`
- `CACHE_ONLY`
- `REMOTE_FIRST`
- `CACHE_FIRST`

For a description of these values, see the section Using Policies with Offline Mode in the offline mode architecture tutorial. Note that each Screenlet behaves a bit differently with offline mode. For specific details, see the Screenlet reference documentation.

Handling Synchronization

Under some scenarios, values stored in the local cache need to be synchronized with the portal. To do this, you need to use the `CacheSyncService` class. This class sends information from the local cache to the portal. To register `CacheSyncService` with your app, you must add the following code to your `AndroidManifest.xml` file:

```
<receiver android:name=".CacheReceiver">
    <intent-filter>
        <action android:name="com.liferay.mobile.screens.auth.login.success"/>
        <action android:name="com.liferay.mobile.screens.cache.resync"/>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
    </intent-filter>
</receiver>

<service
    android:name=".CacheSyncService"
    android:exported="false"/>
```

This code registers the `CacheReceiver` and `CacheSyncService` components. The `CacheReceiver` is invoked in the following scenarios:

- When a connectivity change occurs (for example, when the network connection is restored).
- When Login Screenlet successfully completes the login.
- When a specific resync intent is broadcasted. In this case, use `context.sendBroadcast(new Intent("com.liferay.mobile.screens.cache.resync"));`.

The `CacheSyncService` performs the synchronization process when invoked from the above receiver. This is currently an unassisted process. Future versions will include some kind of control mechanism.

Related Topics

Architecture of Offline Mode in Liferay Screens
Using Screenlets in Android Apps
Using Offline Mode in iOS
Using Screenlets in iOS Apps

76.5 Architecture of Liferay Screens for Android

Liferay Screens applies architectural ideas from Model View Presenter, Model View ViewModel, and VIPER. Screens isn't considered a canonical implementation of these architectures, because it isn't an app, but it borrows from them to separate presentation layers from business-logic. This tutorial explains Screen's high-level architecture, its components' low-level architecture, and the Android Screenlet lifecycle. Now go ahead and get started examining Screens's building blocks!

High-Level Architecture

Liferay Screens for Android is composed of a Core, a Screenlet layer, a View layer, Interactors, and Server Connectors. Interactors are technically part of the core, but are worth covering separately. They facilitate interaction with both local and remote data sources, as well as communication between the Screenlet layer and the Liferay Mobile SDK.

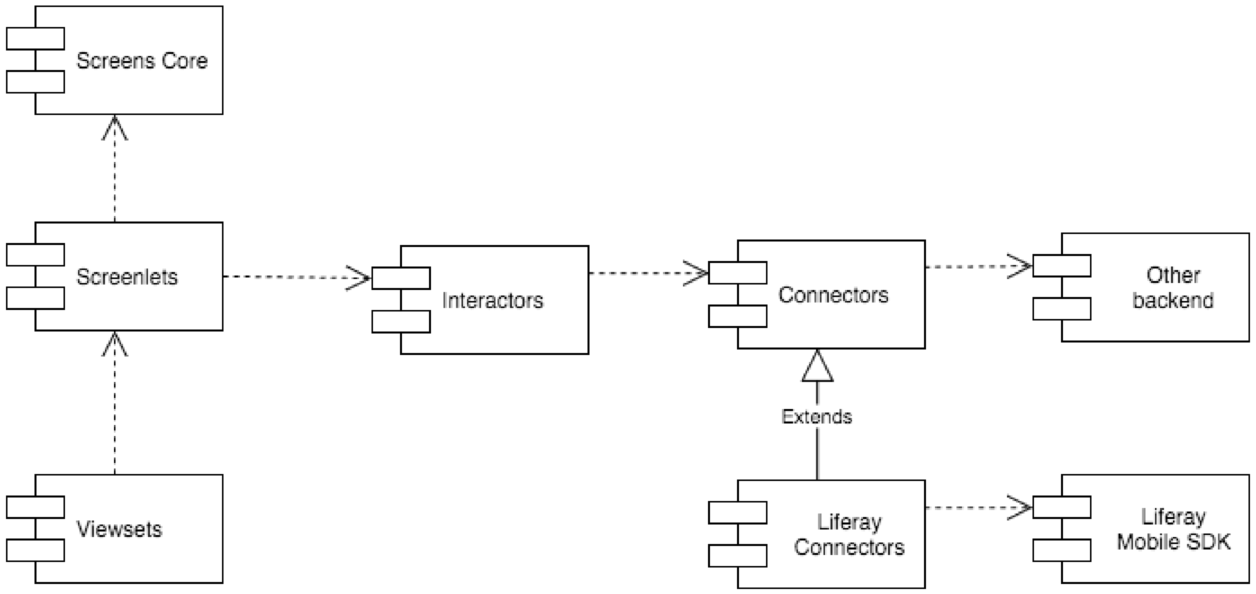


Figure 76.7: Here are the high-level components of Liferay Screens for Android. The dashed arrow connectors represent a "uses" relationship, in which a component uses the component its pointing to.

Each component is described below.

Core: includes all the base classes for developing other Screens components. It's a micro-framework that lets developers write their own Screenlets, Views, and Interactors.

Screenlets: Java view classes for inserting into any activity or fragment view hierarchy. They render a selected layout in the runtime and in Android Studio's visual editor and react to UI events, sending any necessary server requests. You can set a Screenlet's properties from its layout XML file and Java classes. The Screenlets bundled with Liferay Screens are known collectively as the Screenlet Library.

Server Connectors: a collection of classes that interact with different Liferay DXP versions. These classes abstract away the complexity of communicating with different versions. This allows the developer to call API methods and the correct Interactor without worrying about the specific Liferay DXP version.

Interactors: implement specific use cases for communicating with servers. They can use local and remote data sources. Most Interactors use the Liferay Mobile SDK to exchange data with a Liferay instance. If a user action or use case needs to execute more than one query on a local or remote store, the sequence is done in the corresponding Interactor. If a Screenlet supports more than one user action or use case, an Interactor must be created for each. Interactors are typically bound to one specific Liferay version, and instantiated by a Server Connector. Interactors run in a background thread and can therefore perform intensive operations without blocking the UI thread.

Views: a set of layouts and accompanying custom view classes that present Screenlets to the user.

Next, the core layer is described in detail.

Core Layer

The core layer is the micro-framework that lets developers write Screenlets in a structured and isolated way. All Screenlets share a common structure based on the core classes, but each Screenlet can have a unique purpose and communication API.

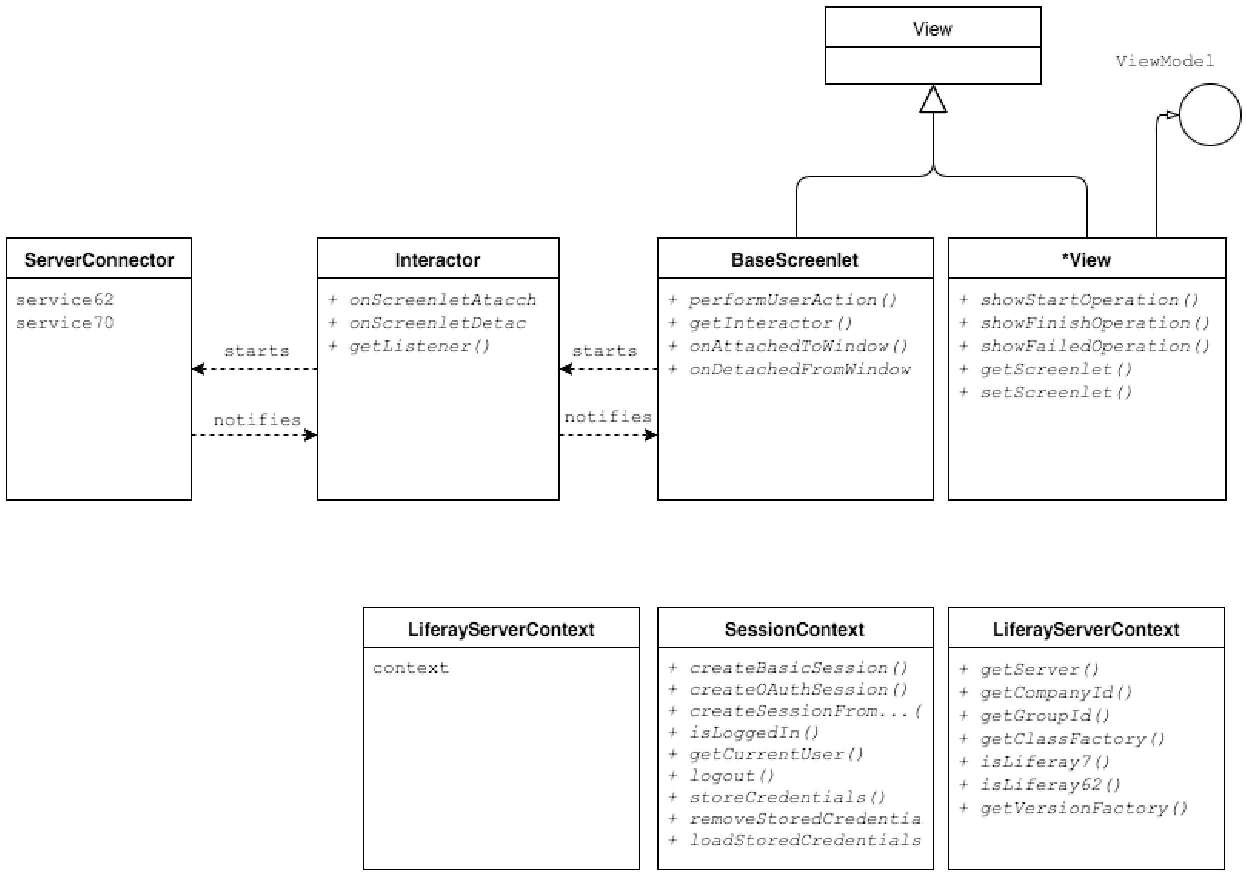


Figure 76.8: Here's the core layer of Liferay Screens for Android.

Here are the core's main components:

Interactor: the base class for all Liferay Portal interactions and use cases that a Screenlet supports. Interactors call services through the Liferay Mobile SDK and receive responses asynchronously through the EventBus, eventually changing a View's state. Their actions can vary in complexity, from performing simple algorithms to requesting data asynchronously from a server or database. A Screenlet can have multiple Interactors, each dedicated to supporting a specific operation.

BaseScreenlet: the base class for all Screenlet classes. It receives user events from a Screenlet's View, instantiates and calls the Interactors, and then updates the View with operation results. Classes that extend it can override its template methods:

- *createScreenletView:* typically inflates the Screenlet's View and gets the attribute values from the XML definition.
- *createInteractor:* instantiates an Interactor for the specified action. If a Screenlet only supports one Interactor type then that type of Interactor is always instantiated.
- *onUserAction:* runs the Interactor associated with the specified action.

Screenlet View: implements the Screenlet's UI. It's instantiated by the Screenlet's `createScreenletView` method. It renders a specific UI using standard layout files and updates the UI with data changes. When developing your own Views that extend a parent View, you can read the parent Screenlet's properties or call its methods from this class.

EventBus: notifies the Interactor when asynchronous operations complete. It decouples the `AsyncTask` class instance from the activity life cycle, to avoid problems typically associated with `AsyncTask` instances.

Liferay Mobile SDK: calls a Liferay instance's remote services in a type-safe and transparent way.

SessionContext: a singleton class that holds the logged in user's session. Apps can use an implicit login, invisible to the user, or a login that relies on explicit user input to create the session. User logins can be implemented with the Login Screenlet. This is explained in detail here.

LiferayServerContext: a singleton object that holds server configuration parameters. It's loaded from the `server_context.xml` file, or from any other XML file that overrides the keys defined in the `server_context.xml`.

server_context.xml: specifies the default server, `companyId` (Liferay instance ID) and `groupId` (site ID). You can also configure other Screens parameters in this file, such as the current Liferay version (with the attribute `liferay_portal_version`) or an alternative `ServiceVersionFactory` to access custom backends.

LiferayScreensContext: a singleton object that holds a reference to the application context. It's used internally where necessary.

ServiceVersionFactory: an interface that defines all the server operations supported in Liferay Screens. This is created and accessed through a `ServiceProvider` that creates the Server Connectors needed to interact with a specific Liferay version. The `ServiceVersionFactory` is an implementation of an Abstract Factory pattern.

Now that you know what makes up the core layer, you're ready to learn the Screenlet layer's details.

Screenlet Layer

The Screenlet layer contains the Screenlets available in Liferay Screens for Android. The following diagram uses Screenlet classes prefixed with *MyScreenlet* to show the Screenlet layer's relationship with the core, View, and Interactor components.

Screenlets are comprised of several Java classes and an XML descriptor file:

MyScreenletViewModel: an interface that defines the attributes shown in the UI. It typically accounts for all the input and output values presented to the user. For instance, `LoginViewModel` includes attributes like the user name and password. The Screenlet can read the attribute values, invoke Interactor operations, and change these values based on operation results.

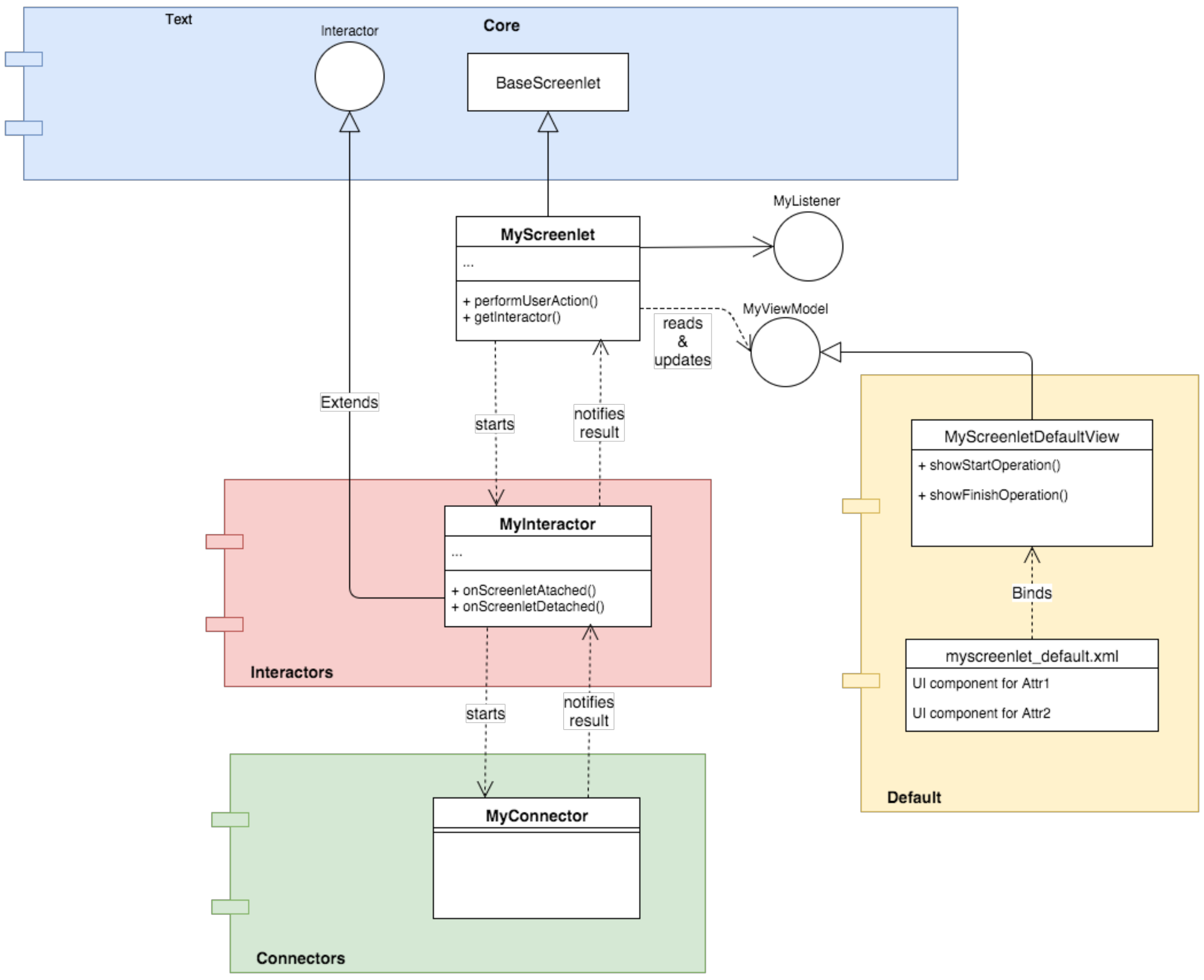


Figure 76.9: This diagram illustrates the Android Screenlet layer's relationship to other Screens components.

MyScreenlet: a class that represents the Screenlet component the app developer interacts with. It includes the following things:

- Attribute fields for configuring the Screenlet's behavior. They are read in the Screenlet's `createScreenletView` method and their default values can optionally be set there too.
- A reference to the Screenlet's View, specified by the `liferay:layoutId` attribute's value. Note: a View must implement the Screenlet's `ViewModel` interface.
- Any number of methods for invoking Interactor operations. You can optionally make them public for app developers to call. They can also handle UI events received in the view class through a regular listener (such as Android's `OnClickListener`) or events forwarded to the Screenlet via the `performUserAction` method.
- An optional (but recommended) listener object for the Screenlet to call on a particular event.

MyScreenletInteractor: implements an end-to-end use case that communicates with a server or consumes a Liferay service. It might perform several intermediate steps. For example, it might send a request to a server, compute a local value based on the response, and then send this value to a different server. On completing an interaction, the Interactor must notify its listeners, one of which is typically the Screenlet class instance. The number of Interactors a Screenlet requires depends on the number of server use cases it supports. For example, the Login Screenlet class only supports one use case (log in the user), so it has only one Interactor. The DDL Forms Screenlet class, however, supports several use cases (load the form, load a record, submit the form, etc.), so it uses a different Interactor class for each use case.

MyScreenletConnector62 and **MyScreenletConnector70**: the classes that create the Interactors required to communicate with a specific Liferay version. The `ServiceProvider` creates a singleton `ServiceVersionFactory` that returns the right Connector.

MyScreenletDefaultView: a class that renders the Screenlet's UI with the default layout. The class in Figure 3, for example, belongs to the Default View set. The View object and the layout file communicate using standard mechanisms, like a `findViewById` method or a listener object. User actions are received by a specified listener (for example, `OnClickListener`) and then passed to the Screenlet object via the `performUserAction` method.

myscreenlet_default.xml: an XML file that specifies how to render the Screenlet's View. Here's a skeleton of a Screenlet's layout XML file:

```
<?xml version="1.0" encoding="utf-8"?>
<com.your.package.MyScreenletView
    xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- Put your regular components here: EditText, Button, etc. -->

</com.your.package.MyScreenletView>
```

Refer to the tutorial Creating Android Screenlets for more Screenlet details. Next, the View layer's details are described.

View Layer

The View layer lets developers set a Screenlet's look and feel. Each Screenlet's `liferay:layoutId` attribute specifies its View. A View consists of a Screenlet class, view class, and layout XML file. The layout XML file specifies the UI components, while the Screenlet class and view class control the View's behavior. By inheriting one or more of these View layer components from another View, the different View *types* allow varying levels of control over a Screenlet's UI design and behavior.

There are several different View types:

Themed: presents the same structure as the current View, but alters the theme colors and tints of the View's resources. All existing Views can be themed with different styles. The View's colors reflect the current value of the Android color palette. If you want to use one View Set with another View Set's colors, you can use those colors in your app's theme (e.g. `colorPrimary_default`, `colorPrimary_material`, `colorPrimary_westeros`).

Child: presents the same behavior and UI components as its parent, but can change the UI components' appearance and position. A Child View specifies visual changes in its own layout XML file; it inherits the parent's view class and Screenlet class. It can't add or remove any UI components. The parent must be a Full View. Creating a Child View is ideal when you only need to make visual changes to an existing View. For example, you might create a Child View for Login Screenlet's Default View to set new positions and sizes for the standard text boxes.

Extended: inherits the parent View's behavior and appearance, but lets you change and add to both. You can do so by creating a custom view class and a new layout XML file. An Extended View inherits all the parent View's other classes, including its Screenlet, listeners, and Interactors; if you need to customize any of them, you must create a Full View to do so. An Extended View's parent must be a Full View. Creating an Extended View is ideal for adding, removing, or changing an existing View's UI components. For example, you can extend the Login Screenlet's Default View to present different UI components for the user name and password fields.

Full: provides a complete standalone View. It doesn't inherit another View's UI components or behavior. When creating a Full View, you must therefore create its Screenlet class, view class, and layout XML file. You should create a Full View when you don't need to inherit another View or when you need to alter the core behavior of a Screenlet by customizing its listeners or calling custom Interactors. For example, you could implement a Full View with a new Interactor for calling a different Liferay Portal instance. Default Views are Full Views.

Liferay Screens Views are organized into *View sets* that contain Views for several Screenlets. Liferay's available View sets are listed here:

- *Default:* a mandatory View Set supplied by Liferay. It's used by a Screenlet if no layout ID is specified or if no View is found with the layout ID. The Default View Set uses a neutral, flat white and blue design with standard UI components. In the Login Screenlet, for example, the Default View uses standard text boxes for the user name and password, but the text boxes are styled with the Default View's flat white and blue design. You can customize this View Set's properties, such as its components' colors, positions, and sizes. See the Default View Set's `styles.xml` file for specific values. Since the Default View Set contains Full Views, you can use them to create your own custom Child and Extended Views.

- *Material:* the View Set containing Views that conform to Android's Material design guidelines.

- *Westeros:* the View Set containing Views for the Bank of Westeros sample app.

For information on creating or customizing Views, see the tutorial Creating Android Views.

Great! Now you know how Liferay Screens for Android is composed. However, there's something you should know before moving on: how Screenlets interact with the Android life cycle.

Screenlet Lifecycle

Liferay Screens automatically saves and restores Screenlets' states using the Android SDK methods `onSaveInstanceState` and `onRestoreInstanceState`. Each Screenlet uses a uniquely generated identifier (`screenletId`) to assign action results to action sources.

The Screenlets' states are restored after the `onCreate` and `onStart` methods, as specified by the standard Android lifecycle. It's a best practice to execute Screenlet methods inside the activity's `onResume` method; this helps assure that actions and tasks find their destinations.

Awesome! Now you know the nitty gritty architectural details of Liferay Screens for Android. Let this tutorial be a resource for you as you work with Liferay Screens.

Related Topics

Using Screenlets in Android Apps

Using Views in Android Screenlets

Creating Android Screenlets

Creating Android Views

76.6 Architecture of Offline Mode in Liferay Screens

Mobile users may encounter difficulty getting or maintaining a network connection at certain locations or times of day. Using offline mode with Screenlets ensures that your app still functions in these situations.

You should note, however, that some difficulties may arise when using an app offline. For example, allowing users to edit data in an app when they're offline may cause synchronization conflicts with the portal when they reconnect. By detailing how offline mode is implemented in Liferay Screens, this tutorial helps you be aware of such difficulties and know how to handle them.

Understanding Offline Mode's Basics

Screenlets in Liferay Screens support the following phases:

1. Get information from the portal.
2. Show information to the user.
3. Collect the user's input (if necessary).
4. Send input to the portal (if necessary).

The following diagram summarizes these phases:

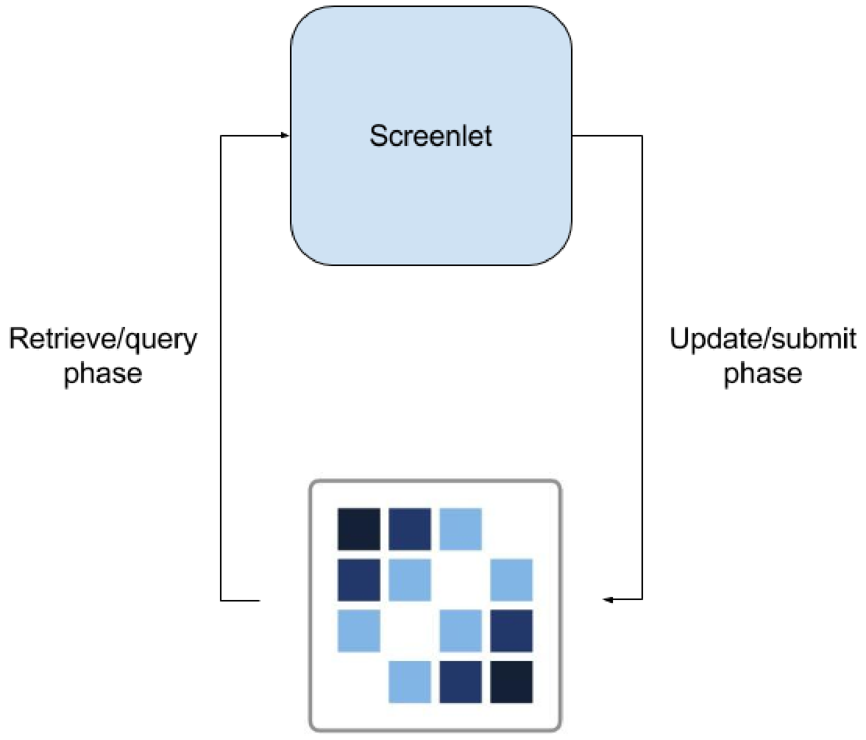


Figure 76.10: A Screenlet's basic phases when requesting and submitting data to the portal.

Note that not all Screenlets need to execute each phase. For example, the Web Content Display Screenlet only needs to retrieve and display portal content. Conversely, Login Screenlet and Sign Up Screenlet only need to handle user input. Only the most complex Screenlets, like the DDL Form Screenlet and the User Portrait Screenlet, need to do both.

So what does all this have to do with offline mode? Liferay Screens's offline infrastructure is a small layer of code that intercepts information going to and coming from the portal. It stores this information in a local data store for use when there's no Internet connection. The following diagram illustrates this, with *Local cache* representing the local data store:

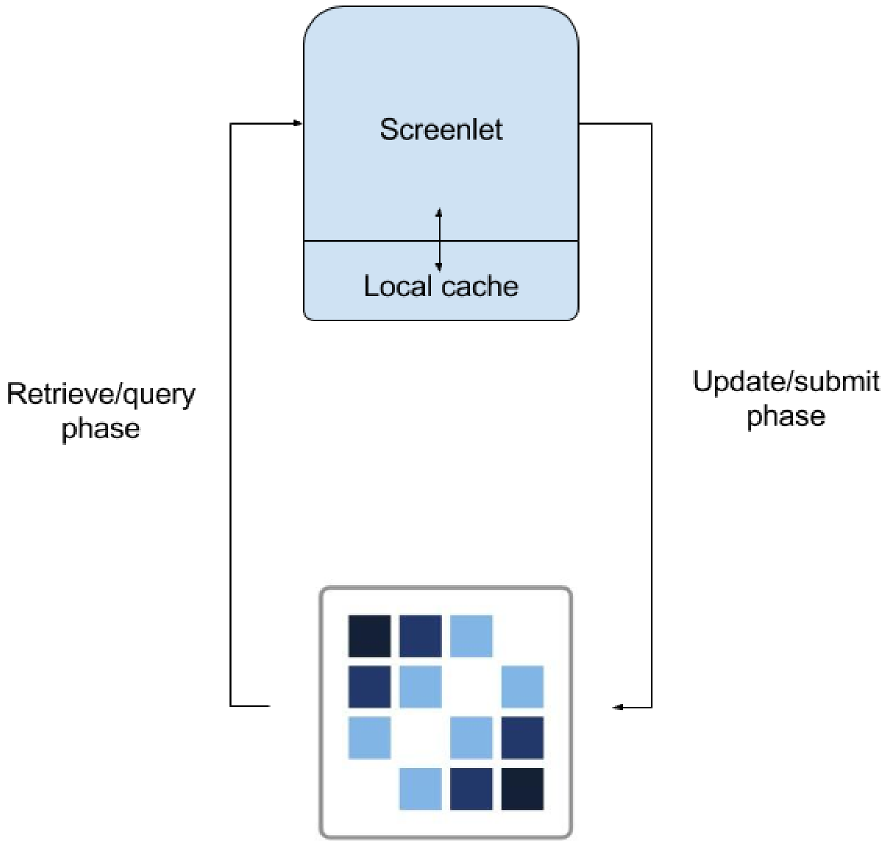


Figure 76.11: This is the same diagram as before, with the addition of the local cache for offline mode.

With offline mode enabled, any Screenlet can persist information exchanged with the portal. You can also configure exactly how offline mode works with the Screenlet you're using. You do this through *policies*.

Using Policies with Offline Mode

Policies configure how a Screenlet behaves with offline mode when it sends or receives data. The Screenlet adheres to the policy even if the data operation fails. Screenlets support the following policies:

remote-only: The Screenlet only uses network connections to load data. Screenlets functioned this way prior to the introduction of offline mode. Use this policy when you want the Screenlet always to use remote content. Your app won't work, however, if a network connection is unavailable. Also, apps using this policy tend to be slower due to network lag. Note that if the request succeeds, the Screenlet stores the data in the local cache for later use.

cache-only: The Screenlet only uses local storage to load data (it doesn't use the network connection). Use this policy when you want the Screenlet to always use offline content. Note that in the app's local cache, some portal data may not exist or may be outdated.

remote-first: The Screenlet first tries to use the network connection to load data. If this fails, it then tries to load data from local storage. Use this policy when you want the Screenlet to use the latest portal data when there's a connection, but also want to support a fallback mechanism when the connection is gone. Note that the Screenlet may use outdated information when there's no connection. In many cases, however, this is better than showing your users no information at all.

cache-first: The Screenlet first tries to load data from local storage. If this fails, it then tries to use the network connection. Use this policy when you want the Screenlet to optimize performance and network efficiency. You can update data in a background process, or let the user update on-demand (via an option, for example). Note that while the information retrieved from local storage may be outdated, loading times and bandwidth consumption are typically lower.

These policies behave a bit differently depending on the data's direction. In other words, when a Screenlet set to a specific policy retrieves information from the portal, it may behave differently than when it submits information to the portal. As an example, consider the possible scenarios for User Portrait Screenlet:

- When loading the portrait:
  - **remote-only:** The Screenlet always attempts to load the portrait from the portal. If the request fails, the operation also fails.
  - **cache-only**: The Screenlet always attempts to load the portrait from the local cache. The operation fails if the portrait doesn't exist there.
  - **remote-first:** The Screenlet first attempts to load the portrait from the portal. If the request succeeds, the Screenlet stores the portrait locally for later use. If the request fails, the Screenlet tries to load the portrait from the local cache. If the local cache doesn't contain the portrait, the Screenlet can't load it, and calls the standard error handling code (call the delegate, use the default placeholder, etc...).
  - **cache-first:** The Screenlet first attempts to load the portrait from the local cache. If the portrait doesn't exist there, it's then requested from the server.
- When submitting the portrait:
  - **remote-only:** The Screenlet first sends the new portrait to the portal. If the submission succeeds, the Screenlet also stores the portrait in the local cache. If the submission fails, the operation also fails.
  - **cache-only**: The Screenlet only stores the portrait locally. The portrait may be loaded from the cache later, or synchronized with the portal.
  - **remote-first:** The Screenlet first tries to send the new portrait to the portal. If this fails due to lack of network connectivity, the Screenlet stores the portrait in the local cache for later synchronization with the portal.
  - **cache-first:** The screenlet first stores the new portrait locally, then sends it to the portal. If the submission fails, the Screenlet still stores the portrait locally, but the send operation fails.

Understanding Synchronization

Synchronization can be a tricky problem to solve. What initially seems straightforward quickly evolves into scenarios where you're not sure which version of the data to use. Having offline users complicates things further. The following diagram illustrates how the Screenlet retrieves and stores portal data.

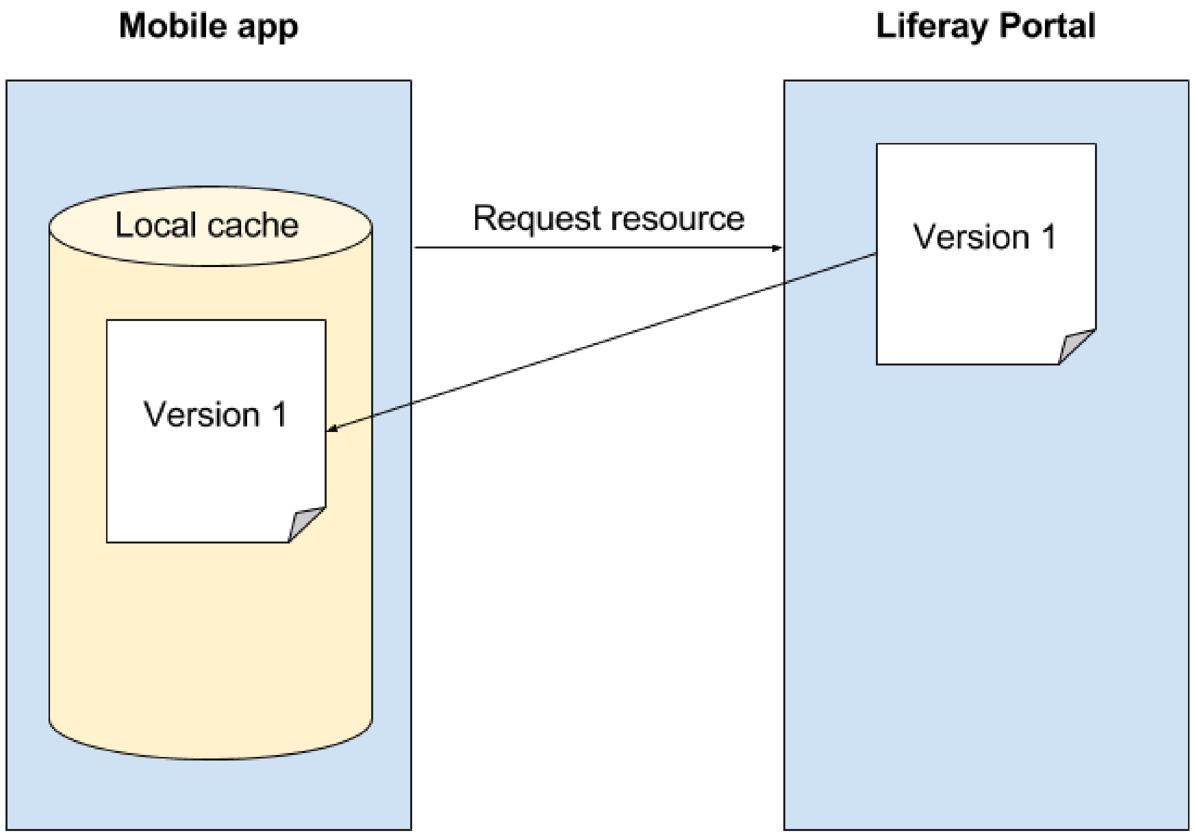


Figure 76.12: The Screenlet requests the resource from the portal and stores it in the app's local cache.

When a user edits the data in the app, the Screenlet needs to send the new data to the portal. But what happens if the user is offline? In this case, the new data can't reach the portal and the local and portal data are out-of-sync. In this scenario, the app has the new data while the portal has the old data. The app's data in this synchronization state is called the *dirty version*. Put away your soap and washcloth. We don't recommend giving your mobile device a bath. In this context, dirty means that the data should be synchronized with the portal as soon as possible. When the Screenlet synchronizes the dirty version, it removes the dirty flag from the local data.

There are other complicated synchronization states. For example, portal data may change while out-of-sync with a Screenlet's local data. To avoid data loss, the local data can't overwrite the portal data, and vice versa. In this situation, the synchronization process produces a conflict when it runs. The following diagram illustrates this.

The developer needs to resolve the conflict by choosing the local data or portal data. Synchronization conflicts have four possible resolutions:

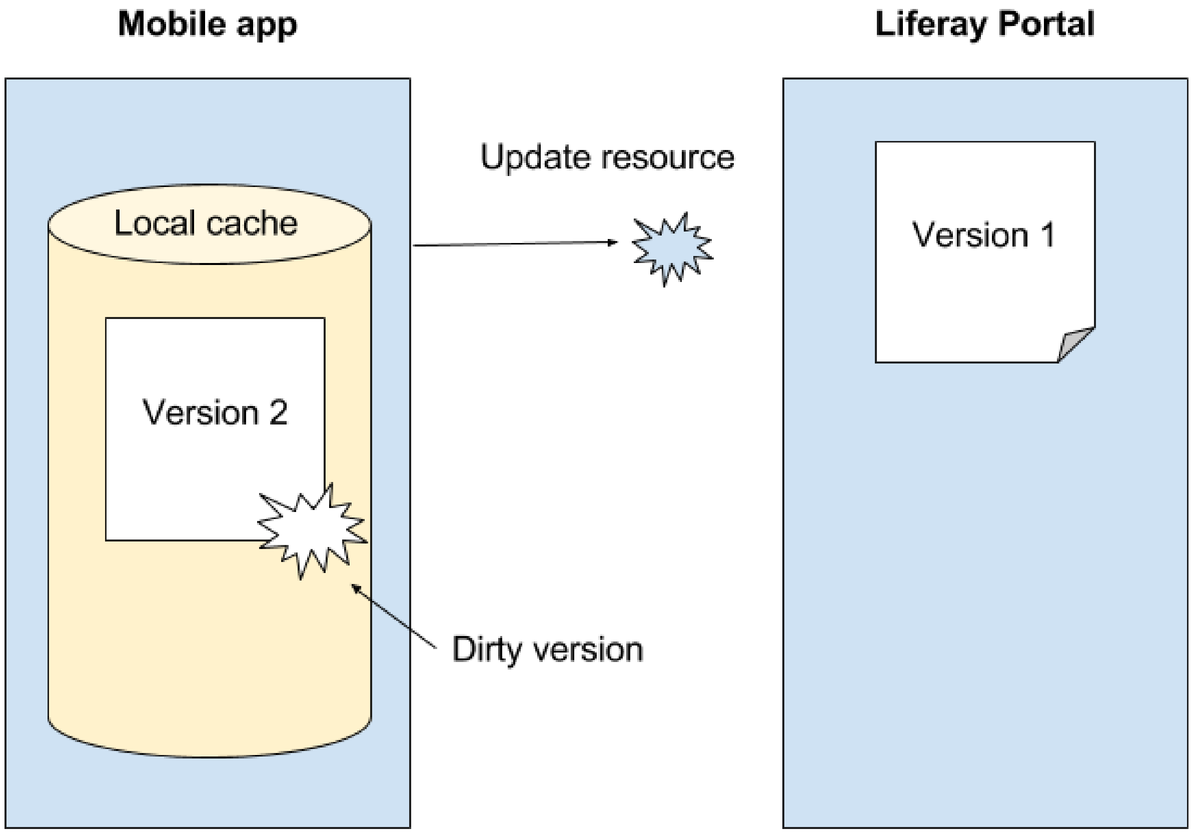


Figure 76.13: The updated data is said to be dirty when the Screenlet can't send it to the portal.

1. **Keep the local version:** The Screenlet overwrites the portal data with the local data. This results in the local cache and the portal having the same version of the data (Version 2 in the above diagram).

2. **Keep remote version:** The Screenlet overwrites the local data with the portal data. This results in the local cache and the portal having the same version of the data (Version 3 in the above diagram).

3. **Discard:** The Screenlet removes the local data, and the portal data isn't overwritten.

4. **Ignore:** The Screenlet doesn't change any data. The next synchronization event reproduces the conflict.

Great! Now that you know how offline mode works, you're ready to put it to use.

Related Topics

Using Offline Mode in Android
Using Offline Mode in iOS
Using Screenlets in Android Apps
Using Screenlets in iOS Apps

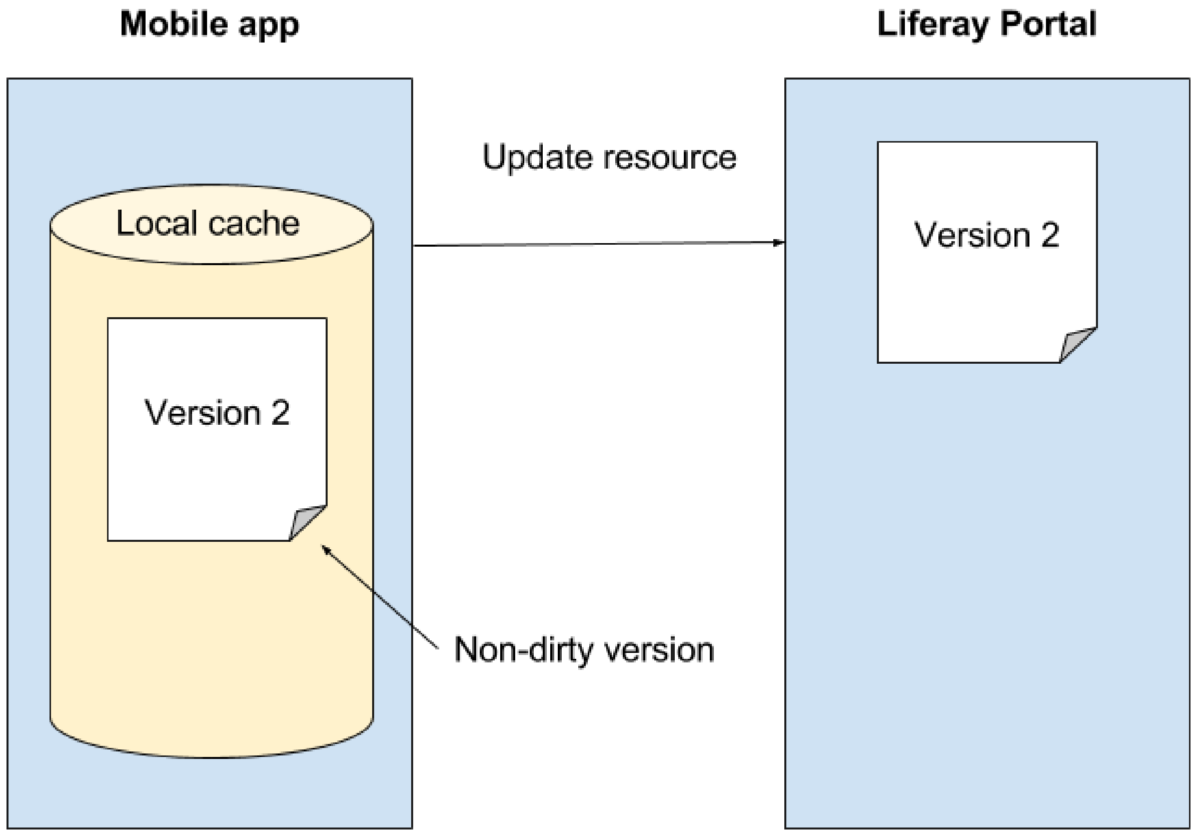


Figure 76.14: The dirty flag is removed once synchronization completes.

76.7 Creating Android Screenlets

The Screenlets that come with Liferay Screens cover common use cases for mobile apps that use Liferay. They authenticate users, interact with Dynamic Data Lists, view assets, and more. However, what if there's no Screenlet for *your* specific use case? No sweat! You can create your own. Extensibility is a key strength of Liferay Screens.

This tutorial explains how to create your own Screenlets. As an example, it references code from the sample Add Bookmark Screenlet, that saves bookmarks to Liferay's Bookmarks portlet.

In general, you use the following steps to create Screenlets:

1. Determine your Screenlet's location. Where you create your Screenlet depends on how you'll use it.

2. Create the Screenlet's UI (its View). Although this tutorial presents all the information you need to create a View for your Screenlet, you may first want to learn how to create a View. For more information on Views in general, see the tutorial on using Views with Screenlets.

3. Create the Screenlet's Interactor. Interactors are Screenlet components that make server calls.

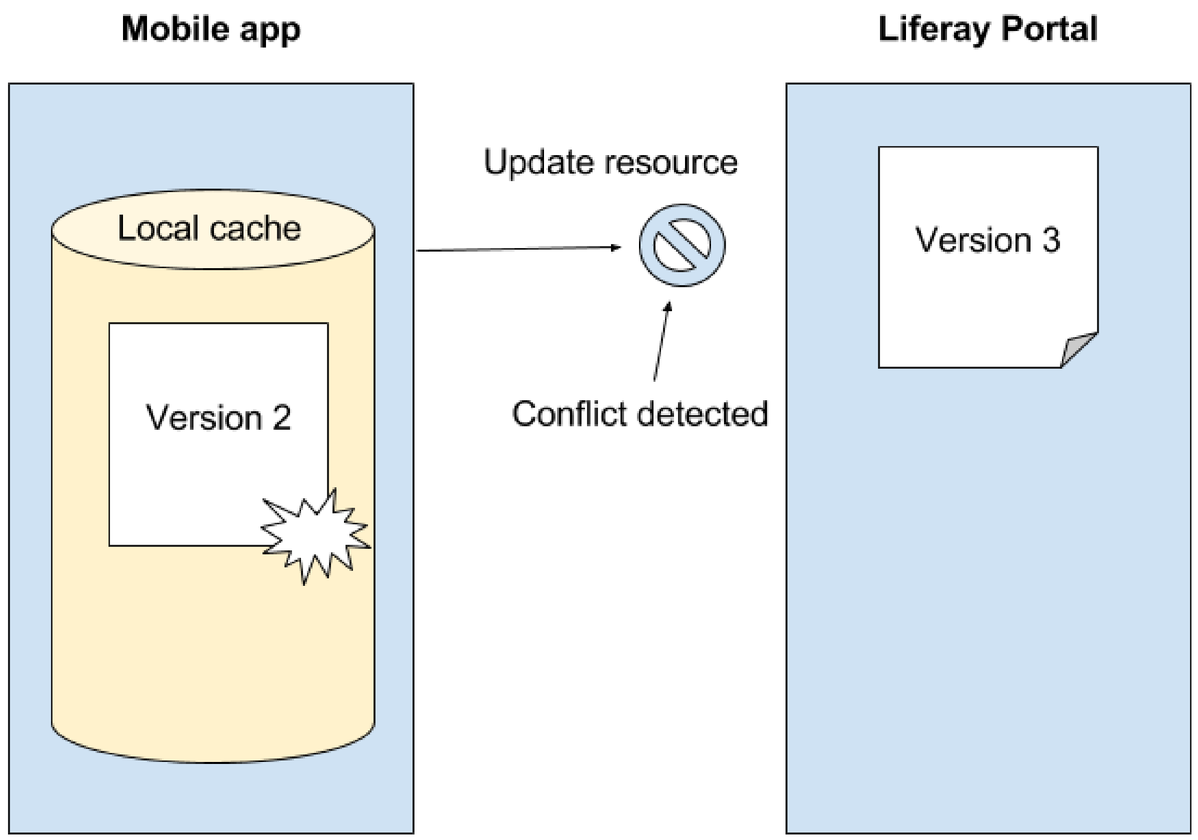


Figure 76.15: Users have changed the data independently in the app and in the portal, causing a synchronization conflict.

4. Define the Screenlet's attributes. These are the XML attributes the app developer can set when inserting the Screenlet's XML. These attributes control aspects of the Screenlet's behavior. You'll add functionality to these attributes in the Screenlet class.
5. Create the Screenlet class. The Screenlet class is the Screenlet's central component. It controls the Screenlet's behavior and is the component the app developer interacts with when inserting a Screenlet.

To understand the components that make up a Screenlet, you should first learn the architecture of Liferay Screens for Android.

Without further ado, let the Screenlet creation begin!

Determining Your Screenlet's Location

Where you should create your Screenlet depends on how you plan to use it. If you don't plan to reuse your Screenlet in another app or don't want to redistribute it, create it in a new package inside your Android app project. This lets you reference and access Liferay Screens's core, in addition to all the View Sets you may have imported.

If you want to reuse your Screenlet in another app, create it in a new Android application module. The tutorial Packaging Android Screenlets explains how to do this. When your Screenlet's project is in place, you can start by creating the Screenlet's UI.

Creating the Screenlet's UI

In Liferay Screens for Android, Screenlet UIs are called Views. Every Screenlet must have at least one View. A View consists of the following components:

- The View Model interface: defines the methods the View needs to update the UI.
- A layout XML file: defines the UI components that the View presents to the end user.
- A View class: renders the UI, handles user interactions, and communicates with the Screenlet class. The View class implements the View Model interface.
- The Screenlet class: Although technically part of a View, the Screenlet class depends on all the other Screenlet components. You therefore won't create the Screenlet class until the end of this tutorial.

The first items to create for a Screenlet's View are its View Model interface and layout. The following steps explain how:

1. To define the methods that every Screenlet's View class must implement, Screens provides the `BaseViewModel` interface. Your View Model interface should extend `BaseViewModel` to define any additional methods needed by your Screenlet. This includes any getters and setters for the attributes you want to use.

   For example, Add Bookmark Screenlet needs attributes for each bookmark's URL and title. Its View Model interface, `AddBookmarkViewModel`, therefore, defines getters and setters for these attributes:

   ```
   public interface AddBookmarkViewModel extends BaseViewModel {
       String getURL();

       void setURL(String value);

       String getTitle();

       void setTitle(String value);
   }
   ```

2. Define your Screenlet's UI by writing a standard Android layout XML file. The layout's root element should be the fully qualified class name of your Screenlet's View class. You'll create that class in the next step, but determine its name now and name the layout's root element after it. Finally, add any UI elements your View needs.

   For example, Add Bookmark Screenlet's layout needs two text fields: one for entering a bookmark's URL and one for entering its title. The layout also needs a button for saving the bookmark. The Screenlet defines this UI in its `bookmark_default.xml` layout file:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <com.your.package.AddBookmarkView
       xmlns:android="http://schemas.android.com/apk/res/android"
       style="@style/default_screenlet">

       <EditText
           android:id="@+id/url"
           android:layout_width="match_parent"
           android:layout_height="wrap_content"
           android:layout_marginBottom="15dp"
           android:hint="URL Address"
           android:inputType="textUri"/>
   ```

```
    <EditText
        android:id="@+id/title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="15dp"
        android:hint="Title"/>

    <Button
        android:id="@+id/add_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Add Bookmark"/>

</com.your.package.AddBookmarkView>
```

Next, you'll create your Screenlet's View class.

Creating the Screenlet's View Class

Your Screenlet needs a View class to support the layout you just created. This class must extend an Android layout class (e.g. `LinearLayout`, `ListView`), implement your View Model interface, and implement a separate listener interface to handle user actions. Follow these steps to create this View class:

1. Create a View class that extends the Android layout class appropriate for your Screenlet's UI. For example, Add Bookmark Screenlet renders its UI components in a single column, so its View class (`AddBookmarkView`) extends Android's `LinearLayout`. Your View class's constructors should call the parent layout class's constructors. For example, `AddBookmarkView`'s constructors call those of `LinearLayout`:

```
public AddBookmarkView(Context context) {
    super(context);
}

public AddBookmarkView(Context context, AttributeSet attributes) {
    super(context, attributes);
}

public AddBookmarkView(Context context, AttributeSet attributes, int defaultStyle) {
    super(context, attributes, defaultStyle);
}
```

2. Add instance variables for your View Model's attributes and `BaseScreenlet`. For example, Add Bookmark Screenlet needs instance variables for a bookmark's URL and title. Because all Screenlet classes extend the `BaseScreenlet` class, a `BaseScreenlet` variable in your View class ensures that your View always has a reference to the Screenlet. For example, here are `AddBookmarkView`'s instance variables:

```
private EditText urlText;
private EditText titleText;
private BaseScreenlet screenlet;
```

3. Implement your View Model interface. Implement your View Model's getter and setter methods to get and set the inner value of each component, respectively. For example, here's `AddBookmarkView`'s implementation of `AddBookmarkViewModel`:

```
public String getURL() {
    return urlText.getText().toString();
}

public void setURL(String value) {
    urlText.setText(value);
}

public String getTitle() {
    return titleText.getText().toString();
}

public void setTitle(String value) {
    titleText.setText(value);
}
```

4. Implement a listener interface to handle user actions in the Screenlet. For example, Add Bookmark Screenlet must detect when the user presses the save button. The `AddBookmarkView` class enables this by implementing Android's `View.OnClickListener` interface, which defines a single method: `onClick`. The Screenlet's `onClick` implementation gets a reference to the Screenlet and calls its `performUserAction()` method (you'll create `performUserAction()` in the Screenlet class shortly):

```
public void onClick(View v) {
    AddBookmarkScreenlet screenlet = (AddBookmarkScreenlet) getParent();

    screenlet.performUserAction();
}
```

You can set the listener to the appropriate UI element by implementing an `onFinishInflate()` method. This method should also retrieve and assign any other UI elements from your layout. For example, the `onFinishInflate()` implementation in `AddBookmarkView` retrieves the URL and title attributes from the layout, and sets them to the `urlText` and `titleText` variables, respectively. This method then retrieves the button from the layout and sets this View class as the button's click listener:

```
protected void onFinishInflate() {
    super.onFinishInflate();

    urlText = (EditText) findViewById(R.id.url);
    titleText = (EditText) findViewById(R.id.title_bookmark);

    Button addButton = (Button) findViewById(R.id.add_button);
    addButton.setOnClickListener(this);
}
```

5. Implement the `BaseViewModel` interface's methods: `showStartOperation`, `showFinishOperation`, `showFailedOperation`, `getScreenlet`, and `setScreenlet`. In the `show*Operation` methods, you can log what happens in your Screenlet when the server operation starts, finishes successfully, or fails, respectively. In the `getScreenlet` and `setScreenlet` methods, you must get and set the `BaseScreenlet` variable, respectively. This ensures that the View always has a Screenlet reference. For example, Add Bookmark Screenlet implements these methods as follows:

```
@Override
public void showStartOperation(String actionName) {

}

@Override
```

```
public void showFinishOperation(String actionName) {
    LiferayLogger.i("Add bookmark successful");
}

@Override
public void showFailedOperation(String actionName, Exception e) {
    LiferayLogger.e("Could not add bookmark", e);
}

@Override
public BaseScreenlet getScreenlet() {
    return screenlet;
}

@Override
public void setScreenlet(BaseScreenlet screenlet) {
    this.screenlet = screenlet;
}
```

Note that although you must implement the `show[something]Operation` methods, you can leave their implementations empty if you don't need to take any specific action.

Click here to see the complete example `AddBookmarkView` class.
Great! Your View class is finished. Now you're ready to create your Screenlet's Interactor class.

Creating the Screenlet's Interactor

A Screenlet's Interactor makes the service call to retrieve the data you need from a Liferay instance. An Interactor is made up of several components:

1. The event class. This class lets you handle communication between the Screenlet's components via event objects that contain the server call's results. Screens uses the EventBus library for this. Screens supplies the `BasicEvent` class and `BaseListEvent` class for communicating `JSONObject` and `JSONArray` results within Screenlets, respectively. You can create your own event classes by extending `BasicEvent`. You should create your own event classes when you must communicate objects other than `JSONObject` or `JSONArray`. The example Add Bookmark Screenlet only needs to communicate `JSONObject` instances, so it uses `BasicEvent`.

2. The listener interface. This defines the methods the app developer needs to respond to the Screenlet's behavior. For example, Login Screenlet's listener defines the `onLoginSuccess` and `onLoginFailure` methods. Screens calls these methods when login succeeds or fails, respectively. By implementing these methods in the activity or fragment class that contains the Screenlet, the app developer can respond to login success and failure. Similarly, the example Add Bookmark Screenlet's listener interface defines two methods: one for responding to the Screenlet's failure to add a bookmark and one for responding to its success to add a bookmark:

   ```
    public interface AddBookmarkListener {

        void onAddBookmarkFailure(Exception exception);

        void onAddBookmarkSuccess();
    }
   ```

3. The Interactor class. This class must extend Screens's `BaseRemoteInteractor` with your listener and event as type arguments. The listener lets the Interactor class send the server call's results to any

classes that implement the listener. In the implementation of the method that makes the server call, the execute method, you must use the Mobile SDK to make an asynchronous service call. This means you must get a session and then make the server call. You make the server call by creating an instance of the Mobile SDK service (e.g., BookmarksEntryService) that can call the Liferay service you need and then making the call. The Interactor class must also process the event object that contains the call's results and then notify the listener of those results. You do this by implementing the onSuccess and onFailure methods to invoke the corresponding getListener() methods.

For example, the AddBookmarkInteractor class is Add Bookmark Screenlet's Interactor class. This class implements the execute method, which adds a bookmark to a folder in a Liferay instance's Bookmarks portlet. This method first validates the bookmark's URL and folder. It then calls the getJSONObject method to add the bookmark, and concludes by returning a new BasicEvent object created from the JSONObject. The if statement in the getJSONObject method checks the Liferay version so it can create the appropriate BookmarksEntryService instance needed to make the server call. Regardless of the Liferay version, the getSession() method retrieves the existing session created by Login Screenlet upon successful login. The session's addEntry method makes the server call. The Screenlet calls the onSuccess or onFailure method to notify the listener of the server call's success or failure, respectively. In either case, the BasicEvent object contains the server call's results. Since this Screenlet doesn't retrieve anything from the server, however, there's no need to process the BasicEvent object in the onSuccess method; calling the listener's onAddBookmarkSuccess method is sufficient. Here's the complete code for AddBookmarkInteractor:

```
public class AddBookmarkInteractor extends BaseRemoteInteractor<AddBookmarkListener, BasicEvent> {

    @Override
    public BasicEvent execute(Object[] args) throws Exception {
        String url = (String) args[0];
        String title = (String) args[1];
        long folderId = (long) args[2];

        validate(url, folderId);

        JSONObject jsonObject = getJSONObject(url, title, folderId);
        return new BasicEvent(jsonObject);
    }

    @Override
    public void onSuccess(BasicEvent event) throws Exception {
        getListener().onAddBookmarkSuccess();
    }

    @Override
    public void onFailure(BasicEvent event) {
        getListener().onAddBookmarkFailure(event.getException());
    }

    private void validate(String url, long folderId) {
        if (url == null || url.isEmpty() || !URLUtil.isValidUrl(url)) {
            throw new IllegalArgumentException("Invalid url");
        } else if (folderId == 0) {
            throw new IllegalArgumentException("folderId not set");
        }
    }

    @NonNull
    private JSONObject getJSONObject(String url, String title, long folderId) throws Exception {
        if (LiferayServerContext.isLiferay7()) {
            return new BookmarksEntryService(getSession()).addEntry(LiferayServerContext.getGroupId(),
                folderId, title, url, "", null);
        } else {
            return new com.liferay.mobile.android.v62.bookmarksentry.BookmarksEntryService(
                getSession()).addEntry(LiferayServerContext.getGroupId(), folderId, title, url, "", null);
        }
    }
}
```

Sweetness! Your Screenlet's Interactor is done. Next, you'll create the Screenlet class.

Defining Screenlet Attributes in Your App

Before creating the Screenlet class, you should define its attributes. These are the attributes the app developer can set when inserting the Screenlet's XML in an activity or fragment layout. For example, to use Login Screenlet, the app developer could insert the following Login Screenlet XML in an activity or fragment layout:

```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:basicAuthMethod="email"
    app:layoutId="@layout/login_default"
    />
```

The app developer can set the `liferay` attributes `basicAuthMethod` and `layoutId` to set Login Screenlet's authentication method and View, respectively. The Screenlet class reads these settings to enable the appropriate functionality.

When creating a Screenlet, you can define the attributes you want to make available to app developers. You do this in an XML file inside your Android project's `res/values` directory. For example, Add Bookmark Screenlet's attributes are defined in the Screenlet's `bookmark_attrs.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="AddBookmarkScreenlet">
        <attr name="layoutId"/>
        <attr name="folderId"/>
        <attr name="defaultTitle" format="string"/>
    </declare-styleable>
</resources>
```

This defines the attributes `layoutId`, `folderId`, and `defaultTitle`. Add Bookmark Screenlet's Screenlet class adds functionality to these attributes. Here's a brief description of what each does:

- `layoutId`: Sets the View that displays the Screenlet. This functions the same as the `layoutId` attribute in Liferay's existing Screenlets.
- `folderId`: Sets the folder ID in the Bookmarks portlet where the Screenlet adds bookmarks.
- `defaultTitle`: Sets each Bookmark's default title.

Now that you've defined your Screenlet's attributes, you're ready to create the Screenlet class.

Creating the Screenlet Class

The Screenlet class is the central hub of a Screenlet. It contains attributes for configuring the Screenlet's behavior, a reference to the Screenlet's View, methods for invoking Interactor operations, and more. When using a Screenlet, app developers primarily interact with its Screenlet class. In other words, if a Screenlet were to become self-aware, it would happen in its Screenlet class (though we're reasonably confident this won't happen).

To make all this possible, your Screenlet class must implement the Interactor's listener interface and extend Screens's `BaseScreenlet` class with the View Model interface and Interactor class as type arguments. Your Screenlet class should also contain instance variables and accompanying getters and setters for the listener and any other attributes that the app developer needs to access. For constructors, you can call `BaseScreenlet`'s constructors.

For example, Add Bookmark Screenlet's Screenlet class extends `BaseScreenlet<AddBookmarkViewModel, AddBookmarkInteractor>` and implements `AddBookmarkListener`. It also contains instance variables for `AddBookmarkListener` and the bookmark's folder ID, and getters and setters for these variables. Also note the constructors call `BaseScreenlet`'s constructors:

```
public class AddBookmarkScreenlet extends
    BaseScreenlet<AddBookmarkViewModel, AddBookmarkInteractor>
    implements AddBookmarkListener {

    private long folderId;
    private AddBookmarkListener listener;

    public AddBookmarkScreenlet(Context context) {
        super(context);
    }

    public AddBookmarkScreenlet(Context context, AttributeSet attributes) {
        super(context, attributes);
    }

    public AddBookmarkScreenlet(Context context, AttributeSet attributes, int defaultStyle) {
        super(context, attributes, defaultStyle);
    }

    public long getFolderId() {
        return folderId;
    }

    public void setFolderId(long folderId) {
        this.folderId = folderId;
    }

    public AddBookmarkListener getListener() {
        return listener;
    }

    public void setListener(AddBookmarkListener listener) {
        this.listener = listener;
    }

    ...
```

Next, implement the Screenlet's listener methods. This lets the Screenlet class receive the server call's results and thus act as the listener. These methods should communicate the server call's results to the View (via the View Model) and any other listener instances (via the Screenlet class's listener instance). For example, here are Add Bookmark Screenlet's listener method implementations:

```
public void onAddBookmarkSuccess() {
    getViewModel().showFinishOperation(null);

    if (listener ≠ null) {
        listener.onAddBookmarkSuccess();
    }
}

public void onAddBookmarkFailure(Exception e) {
    getViewModel().showFailedOperation(null, e);

    if (listener ≠ null) {
        listener.onAddBookmarkFailure(e);
    }
}
```

These methods are called when the server call succeeds or fails, respectively. They first use `getViewModel()` to get a View Model instance and then call the `BaseViewModel` methods `showFinishOperation` and `showFailedOperation` to send the server call's results to the View. The `showFinishOperation` call sends `null` because a successful server call to add a bookmark doesn't return any objects. If a successful server call in your Screenlet returns any objects you need to display, then you should send them in this `showFinishOperation` call. The `showFailedOperation` call sends the `Exception` that results from a failed server call to the View. This lets you display an informative error to the user. The `onAddBookmarkSuccess` and `onAddBookmarkFailure` implementations then call the listener instance's method of the same name. This sends the server call's results to any other classes that implement the listener interface, such as the activity or fragment that uses the Screenlet.

Next, you must implement `BaseScreenlet`'s abstract methods:

- `createScreenletView`: Reads the app developer's Screenlet attribute settings, and inflates the View. You'll use an Android `TypedArray` to retrieve the attribute settings. You should set the attribute values to the appropriate variables, and set any default values you need to display via a View Model reference.

  For example, Add Bookmark Screenlet's `createScreenletView` method gets the app developer's attribute settings via a `TypedArray`. This includes the `layoutId`, `defaultTitle`, and `folderId` attributes. The `layoutId` is used to inflate a View reference (`view`), which is then cast to a View Model instance (`viewModel`). The View Model instance's `setTitle` method is then called with `defaultTitle` to set the bookmark's default title. The method concludes by returning the View reference.

  ```
  @Override
  protected View createScreenletView(Context context, AttributeSet attributes) {
      TypedArray typedArray = context.getTheme()
          .obtainStyledAttributes(attributes, R.styleable.AddBookmarkScreenlet, 0, 0);

      int layoutId = typedArray.getResourceId(R.styleable.AddBookmarkScreenlet_layoutId, 0);

      View view = LayoutInflater.from(context).inflate(layoutId, null);

      String defaultTitle = typedArray.getString(R.styleable.AddBookmarkScreenlet_defaultTitle);

      folderId = castToLong(typedArray.getString(R.styleable.AddBookmarkScreenlet_folderId));

      typedArray.recycle();

      AddBookmarkViewModel viewModel = (AddBookmarkViewModel) view;
      viewModel.setTitle(defaultTitle);

      return view;
  }
  ```

- createInteractor: Instantiates the Screenlet's Interactor. For example, Add Bookmark Screenlet's createInteractor method calls the AddBookmarkInteractor constructor to create a new instance of this Interactor:

  ```
  @Override
  protected AddBookmarkInteractor createInteractor(String actionName) {
      return new AddBookmarkInteractor(getScreenletId());
  }
  ```

- onUserAction: Retrieves any data the user has entered in the View, and starts the Screenlet's server operation via an Interactor instance. If your Screenlet doesn't take user input, this method only needs to do the latter.

  The example Add Bookmark Screenlet takes user input (the bookmark's URL and title), so its onUserAction method must retrieve this data. This method does so via a View Model instance it retrieves with the getViewModel() method. The onUserAction method starts the server operation by calling the Interactor's start method with the user input. Note that the Interactor inherits the start method from the BaseInteractor class. Invoking the start method causes the Interactor's execute method to run in a background thread:

  ```
  @Override
  protected void onUserAction(String userActionName, AddBookmarkInteractor interactor, Object... args) {
      AddBookmarkViewModel viewModel = getViewModel();
      String url = viewModel.getURL();
      String title = viewModel.getTitle();

      interactor.start(url, title, folderId);
  }
  ```

Nice! Your Screenlet is finished! You can now use it the same way you would any other. If you created your Screenlet in its own project, you can also package and distribute it via the Screens project, JCenter, or Maven Central.

To finish the Add Bookmark Screenlet example, the following section shows you how to use this Screenlet. It also shows how you can set default attribute values in an app's server_context.xml file. Although you may not need to do this when using your Screenlets, it might come in handy on your way to becoming a master of Screenlets.

Using Your Screenlet

To use any Screenlet, you must follow these general steps:

1. Insert the Screenlet's XML in the activity or fragment layout you want the Screenlet to appear in. You can fine-tune the Screenlet's behavior by setting the Screenlet XML's attributes.

2. Implement the Screenlet's listener in the activity or fragment class.

As an example of this, the Liferay Screens Test App uses Add Bookmark Screenlet. You can find the following Add Bookmark Screenlet XML in the Test App's add_bookmark.xml layout:

```
<com.liferay.mobile.screens.bookmark.AddBookmarkScreenlet
    android:id="@+id/bookmark_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:folderId="@string/bookmark_folder"
    app:layoutId="@layout/bookmark_default" />
```

Note that the layout specified by `app:layoutId` (`bookmark_default`) matches the layout file of the Screenlet's View (`bookmark_default.xml`). This is how you specify the View that displays your Screenlet. For example, if Add Bookmark Screenlet had another View defined in a layout file named `bookmark_awesome.xml`, you could use that layout by specifying `@layout/bookmark_awesome` as the `app:layoutId` attribute's value.

Also note that the `app:folderId` attribute specifies `@string/bookmark_folder` as the bookmark folder's ID. This is an alternative way of specifying an attribute's value. Instead of specifying the value directly, the Test App specifies the value in its `server_context.xml` file:

```
...
<string name="bookmark_folder">20622</string>
...
```

This `name` attribute's value, `bookmark_folder` is then used in the Screenlet XML to set the `app:folderId` attribute to `20622`.

Great! Now you know how to use the Screenlets you create. You also know a convenient way to specify default values for a Screenlet's attributes.

Related Topics

Using Screenlets in Android Apps
Architecture of Liferay Screens for Android
Creating Android Views
Creating iOS Screenlets

76.8 Creating Android List Screenlets

It's very common for mobile apps to display lists. Liferay Screens lets you display asset lists and DDL lists in your Android app by using Asset List Screenlet and DDL List Screenlet, respectively. Screens also includes list Screenlets for displaying lists of other Liferay entities like web content articles, images, and more. The Screenlet reference documentation lists all the Screenlets included with Liferay Screens. If there's not a list Screenlet for the entity you want to display in a list, you must create your own. A list Screenlet can display any entity from a Liferay instance. For example, you can create a list Screenlet that displays standard Liferay entities like `User`, or custom entities from custom Liferay apps.

This tutorial uses code from the sample Bookmark List Screenlet to show you how to create your own list Screenlet. This Screenlet displays a list of bookmarks from Liferay's Bookmarks portlet. You can find this Screenlet's complete code here in GitHub.

Note that because this tutorial focuses on creating a list Screenlet, it doesn't explain general Screenlet concepts and components. Before beginning, you should therefore read the following tutorials:

- Screens architecture tutorial
- Basic Screenlet creation tutorial

You'll create the list Screenlet by following these steps:

1. Creating the Model Class
2. Creating the View
3. Creating the Interactor
4. Creating the Screenlet Class

First though, you should understand how pagination works with list Screenlets.

Pagination

To ensure that users can scroll smoothly through large lists of items, list Screenlets support fluent pagination. Support for this is built into the list Screenlet framework. You'll see this as you construct your list Screenlet.

Now you're ready to begin!

Creating the Model Class

Entities come back from Liferay in JSON. To work with these results efficiently in your app, you must convert them to model objects that represent the entity in Liferay. Although Screens's `BaseListInteractor` transforms the JSON entities into `Map` objects for you, you still must convert these into proper entity objects for use in your app. You'll do this via a model class.

For example, Bookmark List Screenlet's model class (`Bookmark`) creates `Bookmark` objects that contain a bookmark's URL and other data. To ensure quick access to the URL, the constructor that takes a `Map<String, Object>` extracts it from the `Map` and sets it to the `url` variable. To allow access to any other data, the same constructor sets the entire `Map` to the `values` variable. Besides the getters and setter, the rest of this class implements Android's `Parcelable` interface:

```
import android.os.Parcel;
import android.os.Parcelable;

import java.util.Map;

public class Bookmark implements Parcelable {

    private String url;
    private Map values;

    public static final Creator<Bookmark> CREATOR = new Creator<Bookmark>() {
        @Override
        public Bookmark createFromParcel(Parcel in) {
            return new Bookmark(in);
        }

        @Override
        public Bookmark[] newArray(int size) {
            return new Bookmark[size];
        }
    };

    public Bookmark() {
        super();
    }

    protected Bookmark(Parcel in) {
        url = in.readString();
    }

    public Bookmark(Map<String, Object> stringObjectMap) {
        url = (String) stringObjectMap.get("url");
        values = stringObjectMap;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(url);
    }

    @Override
    public int describeContents() {
        return 0;
    }
    public String getUrl() {
        return url;
    }

    public Map getValues() {
        return values;
    }

    public void setValues(Map values) {
        this.values = values;
    }

}
```

Now that you have your model class, you can create your Screenlet's View.

Creating the Screenlet's View

Recall from the basic Screenlet creation tutorial that a View defines a Screenlet's UI. To accommodate its list, a list Screenlet's View is constructed a bit differently than that of a non-list Screenlet. To create a List Screenlet's View, you'll create the following components:

1. Row Layout: the layout for each list row.
2. Adapter Class: an Android adapter class that populates each list row with data.
3. View Class: the class that controls the View. This class serves the same purpose in list Screenlets as it does in non-list Screenlets.
4. Main Layout: the layout for the list as a whole. Note this is different from the row layout, which defines the UI for individual rows.

First, you'll create the row layout.

Creating the Row Layout

Before constructing the rest of the View, you should first define the layout to use for each row in the list. For example, Bookmark List Screenlet needs to display a bookmark in each row. Its row layout (`res/layout/bookmark_row.xml`) is therefore a `LinearLayout` containing a single `TextView` that displays the bookmark's URL:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
        android:id="@+id/bookmark_url"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

As you can see, this example is very simple. Row layouts, however, can be as simple or complex as you need them to be to display your content.

Next, you'll create the adapter class.

Creating the Adapter Class

Android adapters fill a layout with content. In the example Bookmark List Screenlet, the layout is the row layout (`bookmark_row.xml`) and the content is each list item (a URL). To make list scrolling smooth, the adapter class should use an Android view holder. To make this easier, you can extend the list Screenlet framework's `BaseListAdapter` class with your model class and view holder as type arguments. By extending `BaseListAdapter`, your adapter needs only two methods:

- `createViewHolder`: instantiates the view holder
- `fillHolder`: fills in the view holder for each row

Your view holder should also contain variables for any data each row needs to display. The view holder must assign these variables to the corresponding row layout elements, and set the appropriate data to them.

For example, Bookmark List Screenlet's adapter class (`BookmarkAdapter`) extends `BaseListAdapter` with `Bookmark` and `BookmarkAdapter.BookmarkViewHolder` as type arguments. This class's view holder is an inner class that extends `BaseListAdapter`'s view holder. Since Bookmark List Screenlet only needs to display a URL in each row, the view holder only needs one variable: `url`. The view holder's constructor assigns the `TextView` from `bookmark_row.xml` to this variable. The `bind` method then sets the bookmark's URL as the `TextView`'s text. The other methods in `BookmarkAdapter` leverage the view holder. The `createViewHolder` method instantiates `BookmarkViewHolder`. The `fillHolder` method calls the view holder's `bind` method to set the bookmark's URL as the `url` variable's text:

```
public class BookmarkAdapter extends BaseListAdapter<Bookmark, BookmarkAdapter.BookmarkViewHolder> {

    public BookmarkAdapter(int layoutId, int progressLayoutId, BaseListAdapterListener listener) {
        super(layoutId, progressLayoutId, listener);
    }

    @NonNull
    @Override
    public BookmarkViewHolder createViewHolder(View view, BaseListAdapterListener listener) {
        return new BookmarkAdapter.BookmarkViewHolder(view, listener);
    }

    @Override
    protected void fillHolder(Bookmark entry, BookmarkViewHolder holder) {
        holder.bind(entry);
    }

    public class BookmarkViewHolder extends BaseListAdapter.ViewHolder {

        private final TextView url;

        public BookmarkViewHolder(View view, BaseListAdapterListener listener) {
            super(view, listener);

            url = (TextView) view.findViewById(R.id.bookmark_url);
        }

        public void bind(Bookmark entry) {
            url.setText(entry.getUrl());
        }
    }
}
```

Great! Your adapter class is finished. Next, you'll create the View class.

Creating the View Class

Now that your adapter exists, you can create your list Screenlet's View class. Recall from the basic Screenlet creation tutorial that the View class is the central hub of any Screenlet's UI. It renders the UI, handles user interactions, and communicates with the Screenlet class. The list Screenlet framework provides most of this functionality for you via the `BaseListScreenletView` class. Your View class must extend this class to provide your row layout ID and an instance of your adapter. You'll do this by overriding `BaseListScreenletView`'s `getItemLayoutId` and `createListAdapter` methods. Note that in many cases this is the only custom functionality your View class needs. If it needs more, you can provide it by creating new methods or overriding other `BaseListScreenletView` methods.

Create your View class by extending `BaseListScreenletView` with your model class, view holder, and adapter as type arguments. This is required for your View class to represent your model objects in a view holder, inside an adapter. For example, Bookmark List Screenlet's View class (`BookmarkListView`) must represent `Bookmark` instances in a `BookmarkViewHolder` inside a `BookmarkAdapter`. The `BookmarkListView` class must therefore extend `BaseListScreenletView` parameterized with `Bookmark`, `BookmarkAdapter.BookmarkViewHolder`, and `BookmarkAdapter`. Besides overriding `createListAdapter` to return a `BookmarkAdapter` instance, the only other functionality that this View class needs to support is to get the layout for each row in the list. The overridden `getItemLayoutId` method does this by returning the row layout `bookmark_row`:

```
import android.content.Context;
import android.util.AttributeSet;

import com.liferay.mobile.screens.base.list.BaseListScreenletView;

public class BookmarkListView
    extends BaseListScreenletView<Bookmark, BookmarkAdapter.BookmarkViewHolder, BookmarkAdapter> {

    public BookmarkListView(Context context) {
        super(context);
    }

    public BookmarkListView(Context context, AttributeSet attributes) {
        super(context, attributes);
    }

    public BookmarkListView(Context context, AttributeSet attributes, int defaultStyle) {
        super(context, attributes, defaultStyle);
    }

    @Override
    protected BookmarkAdapter createListAdapter(int itemLayoutId, int itemProgressLayoutId) {
        return new BookmarkAdapter(itemLayoutId, itemProgressLayoutId, this);
    }

    @Override
    protected int getItemLayoutId() {
        return R.layout.bookmark_row;
    }
}
```

Next, you'll create your View's main layout.

Creating the View's Main Layout

Although you already created a layout for your list rows, you must still create a layout to define the list as a whole. This layout must contain:

- The View class's fully qualified name as the layout's first element.

- An Android `RecyclerView` to let your app efficiently scroll through a potentially large list of items.
- An Android `ProgressBar` to indicate progress when loading the list.

Apart from the View class and styling, this layout's code is the same for all list Screenlets. For example, here's Bookmark List Screenlet's layout `res/layout/list_bookmarks.xml`:

```
<com.liferay.mobile.screens.listbookmark.BookmarkListView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/liferay_list_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ProgressBar
        android:id="@+id/liferay_progress"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:visibility="gone"/>

    <android.support.v7.widget.RecyclerView
        android:id="@+id/liferay_recycler_list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:visibility="gone"/>
</com.liferay.mobile.screens.listbookmark.BookmarkListView>
```

Warning: The `android:id` values in your View's layout XML must **exactly** match the ones shown here. These values are hardcoded into the Screens framework and changing them will cause your app to crash.

Great job! Your View is finished. Next, you'll create your Screenlet's Interactor.

Creating the Screenlet's Interactor

Recall from the basic Screenlet creation tutorial that Interactors retrieve and process a server call's results. Also recall that the following components make up an Interactor:

1. Event
2. Listener
3. Interactor Class

These components perform the same basic functions in list Screenlets as they do in non-list Screenlets. Creating them, however, is a bit different. Each of the following sections show you how to create one of these components. First, you'll create the event.

Creating the Event

Screens uses the EventBus library to handle communication within Screenlets. Screenlet components therefore communicate with each other by using event classes that contain the server call's results. Your list Screenlet's event class must extend the `ListEvent` class parameterized with your model class. Your event class should also contain a private instance variable for the model class, a constructor that sets this variable, and a no-argument constructor that calls the superclass constructor. For example, Bookmark List Screenlet's event class (`BookmarkEvent`) communicates `Bookmark` objects. It therefore extends `ListEvent` with `Bookmark` as a type argument, and defines a private `Bookmark` variable that its `BookmarkEvent(Bookmark bookmark)` constructor sets:

```
public class BookmarkEvent extends ListEvent<Bookmark> {

     private Bookmark bookmark;

     public BookmarkEvent() {
          super();
     }

     public BookmarkEvent(Bookmark bookmark) {
         this.bookmark = bookmark;
     }
    ...
```

You must also implement ListEvent's abstract methods in your event class. Note that these methods support offline mode. Although these methods are briefly described here, supporting offline mode in your Screenlets is addressed in detail in a separate tutorial.

- `getListKey`: returns the ID for the cache. This ID is typically the data each list row displays. For example, the `getListKey` method in `BookmarkEvent` returns the bookmark's URL:

  ```
  @Override
  public String getListKey() {
      return bookmark.getUrl();
  }
  ```

- `getModel`: unwraps the model entity to the cache by returning the model class instance. For example, the `getModel` method in `BookmarkEvent` method returns the bookmark:

  ```
  @Override
  public Bookmark getModel() {
      return bookmark;
  }
  ```

Next, you'll create your Screenlet's listener.

Creating the Listener

Recall that listeners let the app developer respond to events that occur in Screenlets. For example, an app developer using Login Screenlet in an activity must implement `LoginListener` in that activity to respond to login success or failure. When creating a list Screenlet, however, you don't have to create a separate listener. Developers can use your list Screenlet in an activity or fragment by implementing the `BaseListListener` interface parameterized with your model class. For example, to use Bookmark List Screenlet in an activity, an app developer's activity declaration could look like this:

```
public class BookmarkListActivity extends AppCompatActivity
    implements BaseListListener<Bookmark> {...
```

The `BaseListListener` interface defines the following methods that the app developer can implement in their activity or fragment:

- `void onListPageFailed(int startRow, Exception e)`: Responds to the Screenlet's failure to retrieve entities from the server.

- `void onListPageReceived(int startRow, int endRow, List<E> entries, int rowCount)`: Responds to the Screenlet's success in retrieving entities from the server.

- void onListItemSelected(E element, View view): Responds to a user selection in the list.

If these methods meet your list Screenlet's needs, then you can move on to the next section in this tutorial. If you want to let app developers respond to more actions, however, you must create your own listener that extends BaseListListener parameterized with your model class. For example, Bookmark List Screenlet contains such a listener: BookmarkListListener. This listener defines a single method that notifies the app developer when the Interactor is called:

```
public interface BookmarkListListener extends BaseListListener<Bookmark> {
    void interactorCalled();
}
```

Next, you'll create the Interactor class.

Creating the Interactor Class

Recall that as an Interactor's central component, the Interactor class makes the service call to retrieve entities from Liferay DXP, and processes the results of that call. The list Screenlet framework's BaseListInteractor class provides most of the functionality that Interactor classes in list Screenlets require. You must, however, extend BaseListInteractor to make your service calls and handle their results via your model and event classes. Your Interactor class must therefore extend BaseListInteractor, parameterized with BaseListInteractorListener<YourModelClass> and your event class. For example, Bookmark List Screenlet's Interactor class, BookmarkListInteractor, extends BaseListInteractor parameterized with BaseListInteractorListener<Bookmark> and BookmarkEvent:

```
public class BookmarkListInteractor extends
    BaseListInteractor<BaseListInteractorListener<Bookmark>, BookmarkEvent> {...
```

Your Interactor must also override the methods needed to make the server call and process the results:

- getPageRowsRequest: Retrieves the specified page of entities. In the example BookmarkListInteractor, this method first uses the args parameter to retrieve the ID of the folder to retrieve bookmarks from. It then sets the comparator (more on this shortly) if the app developer sets one when inserting the Screenlet XML in a fragment or activity. The getPageRowsRequest method finishes by calling BookmarksEntryService's getEntries method to retrieve a page of bookmarks. Note that the service call, like the service call in the basic Screenlet creation tutorial, uses LiferayServerContext.isLiferay7() to check the portal version to make sure the correct service instance is used. This isn't required if you only plan to use your Screenlet with one portal version. Also note that the groupId variable used to make the service calls isn't set anywhere in getPageRowsRequest or BookmarkListInteractor. Interactors that extend BaseListInteractor, like BookmarkListInteractor, inherit this variable via the Screens framework. You'll set it when you create the Screenlet class. Here's BookmarkListInteractor's complete getPageRowsRequest method:

  ```
  @Override
  protected JSONArray getPageRowsRequest(Query query, Object... args) throws Exception {
      long folderId = (long) args[0];

      if (args[1] ≠ null) {
          query.setComparator((String) args[1]);
      }

      if (LiferayServerContext.isLiferay7()) {
          return new BookmarksEntryService(getSession()).getEntries(groupId, folderId,
              query.getStartRow(), query.getEndRow(), query.getComparatorJSONWrapper());
  ```

```
        } else {
            return new com.liferay.mobile.android.v62.bookmarksentry.BookmarksEntryService(
                getSession()).getEntries(groupId, folderId, query.getStartRow(),
                query.getEndRow(), query.getComparatorJSONWrapper());
        }
    }
```

You might now be asking yourself what a comparator is. A comparator is a class in the Liferay DXP instance that sorts a portlet's entities. For example, the Bookmarks portlet contains several comparators that can sort entities by different criteria. Click here to see these comparators. Although it's not required, you can develop your list Screenlet to use a comparator to sort its entities. Since Bookmark List Screenlet supports comparators, you'll see more of this as you progress through this tutorial.

- `getPageRowCountRequest`: Retrieves the number of entities, to enable pagination. In the example `BookmarkListInteractor`, this method first uses the `args` parameter to get the ID of the folder in which to count bookmarks. It then calls `BookmarksEntryService`'s `getEntriesCount` method to retrieve the number of bookmarks:

```
@Override
protected Integer getPageRowCountRequest(Object... args) throws Exception {
    long folderId = (long) args[0];

    if (LiferayServerContext.isLiferay7()) {
        return new BookmarksEntryService(getSession()).getEntriesCount(groupId, folderId);
    } else {
        return new com.liferay.mobile.android.v62.bookmarksentry.BookmarksEntryService(
            getSession()).getEntriesCount(groupId, folderId);
    }
}
```

- `createEntity`: Returns an instance of your event that contains the server call's results. This method receives the results as `Map<String, Object>`, which it uses to instantiate your model class. It then uses this model instance to create the event object. In the example `BookmarkListInteractor`, this method passes the `Map<String, Object>` to the `Bookmark` constructor. It then uses the resulting `Bookmark` to create and return a `BookmarkEvent`:

```
@Override
protected BookmarkEvent createEntity(Map<String, Object> stringObjectMap) {
    Bookmark bookmark = new Bookmark(stringObjectMap);
    return new BookmarkEvent(bookmark);
}
```

- `getIdFromArgs`: a boilerplate method that returns the value of the first object argument as a string. This serves as a cache key for offline mode:

```
@Override
protected String getIdFromArgs(Object... args) {
    return String.valueOf(args[0]);
}
```

  You must implement this method even if you don't intend to support offline mode in your Screenlet. Having this method in your Interactor class makes it simpler to add offline mode functionality later. Supporting offline mode in your Screenlets is addressed in detail in a separate tutorial.

To see the complete `BookmarkListInteractor` class, click here.
Next, you'll create the Screenlet class.

Creating the Screenlet Class

Recall from the basic Screenlet creation tutorial that the Screenlet class serves as your Screenlet's focal point. It governs the Screenlet's behavior and is the primary component the app developer interacts with. As with non-list Screenlets, you should first define any XML attributes that you want to make available to the app developer. For example, Bookmark List Screenlet defines the following attributes:

- `groupId`: the ID of the site containing the Bookmarks portlet
- `folderId`: the ID of the Bookmarks portlet folder to retrieve bookmarks from
- `comparator`: the name of the comparator to use to sort the bookmarks

The Screenlet defines these attributes in its `res/values/bookmark_attrs.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <declare-styleable name="BookmarkListScreenlet">
        <attr name="groupId"/>
        <attr name="folderId"/>
        <attr format="string" name="comparator"/>
    </declare-styleable>
</resources>
```

Now you're ready to create your Screenlet class. Because the `BaseListScreenlet` class provides the basic functionality for all Screenlet classes in list Screenlets, including methods for pagination and other default behavior, your Screenlet class must extend `BaseListScreenlet` with your model class and Interactor as type arguments.

For example, Bookmark List Screenlet's Screenlet class–`BookmarkListScreenlet`–extends `BaseListScreenlet` parameterized with `Bookmark` and `BookmarkListInteractor`:

```
public class BookmarkListScreenlet
    extends BaseListScreenlet<Bookmark, BookmarkListInteractor> {...
```

You must also create instance variables for the XML attributes that you want to pass to your Interactor. For example, recall that the request methods in `BookmarkListInteractor` receive two `Object` arguments: the folder ID and the comparator. The `BookmarkListScreenlet` class must therefore contain variables for these parameters so it can pass them to the Interactor:

```
private long folderId;
private String comparator;
```

For constructors, leverage the superclass constructors. For example, here are `BookmarkListScreenlet`'s constructors:

```
public BookmarkListScreenlet(Context context) {
    super(context);
}

public BookmarkListScreenlet(Context context, AttributeSet attrs) {
    super(context, attrs);
}

public BookmarkListScreenlet(Context context, AttributeSet attrs, int defStyleAttr) {
    super(context, attrs, defStyleAttr);
}

public BookmarkListScreenlet(Context context, AttributeSet attrs, int defStyleAttr,
    int defStyleRes) {
        super(context, attrs, defStyleAttr, defStyleRes);
}
```

Now you must implement the error method. This is a boilerplate method that uses a listener in the Screenlet framework to propagate any exception, and the user action that produced it, that occurs during the service call. This method does this by checking for a listener and then calling its error method with the Exception and userAction:

```
@Override
public void error(Exception e, String userAction) {
    if (getListener() ≠ null) {
        getListener().error(e, userAction);
    }
}
```

Next, override the createScreenletView method to read the values of the XML attributes you defined earlier and create the Screenlet's View. Recall from the basic Screenlet creation tutorial that this method assigns the attribute values to their corresponding instance variables. For example, the createScreenletView method in BookmarkListScreenlet assigns the folderId and comparator attribute values to variables of the same name. This method also sets the local variable groupId. Recall that the Screens framework propagates this variable to your Interactor. Finish the createScreenletView method by calling the superclass's createScreenletView method. This instantiates the View for you:

```
@Override
protected View createScreenletView(Context context, AttributeSet attributes) {
    TypedArray typedArray = context.getTheme().obtainStyledAttributes(attributes,
        R.styleable.BookmarkListScreenlet, 0, 0);
    groupId = typedArray.getInt(R.styleable.BookmarkListScreenlet_groupId,
        (int) LiferayServerContext.getGroupId());
    folderId = typedArray.getInt(R.styleable.BookmarkListScreenlet_folderId, 0);
    comparator = typedArray.getString(R.styleable.BookmarkListScreenlet_comparator);
    typedArray.recycle();

    return super.createScreenletView(context, attributes);
}
```

Next, override the loadRows method to start your Interactor and thereby retrieve the list rows from the server. This method takes an instance of your Interactor as an argument, which you use to call the Interactor's start method. Note that the Interactor inherits start from BaseListInteractor. You can also use the loadRows method to execute any other code that you want to run when the Interactor starts. For example, the loadRows method in BookmarkListScreenlet first retrieves a listener instance so it can call the listener's interactorCalled method. It then starts the server operation to retrieve the list rows by calling the Interactor's start method with folderId and comparator:

```
@Override
protected void loadRows(BookmarkListInteractor interactor) {

    ((BookmarkListListener) getListener()).interactorCalled();

    interactor.start(folderId, comparator);
}
```

Note that if your Interactor doesn't require arguments, then you can pass the start method 0 or null. Calling start with no arguments, however, causes the server call to fail.

Lastly, override the createInteractor method to instantiate your Interactor. Since that's all this method needs to do, call your Interactor's constructor and return the new instance. For example, BookmarkListScreenlet's createInteractor method returns a new BookmarkListInteractor:

```
@Override
protected BookmarkListInteractor createInteractor(String actionName) {
    return new BookmarkListInteractor();
}
```

You're done! Your Screenlet is a ready-to-use component that you can use in your app. You can even package your Screenlet and contribute it to the Screens project, or distribute it in Maven Central or jCenter.

Using the Screenlet

You can now use your new list Screenlet the same way you use any other Screenlet:

1. Insert the Screenlet's XML in the layout of the activity or fragment you want to use the Screenlet in. For example, here's Bookmark List Screenlet's XML:

   ```
   <com.liferay.mobile.screens.listbookmark.BookmarkListScreenlet
       android:id="@+id/bookmarklist_screenlet"
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       app:comparator="FULLY_QUALIFIED_COMPARATOR_CLASS"
       app:folderId="YOUR_FOLDER_ID"
       app:groupId="YOUR_GROUP_ID"
       app:layoutId="@layout/list_bookmarks"/>
   ```

   Note that to set a comparator, you must use its fully qualified class name. For example, to use the Bookmarks portlet's `EntryURLComparator`, set `app:comparator` in the Screenlet XML as follows:

   ```
   app:comparator="com.liferay.bookmarks.util.comparator.EntryURLComparator"
   ```

2. Implement the Screenlet's listener in the activity or fragment class. If your list Screenlet doesn't have a custom listener, then you can do this by implementing `BaseListListener` parameterized with your model class. For example:

   ```
   public class YourListActivity extends AppCompatActivity
       implements BaseListListener<YourModelClass> {...
   ```

   If you created a custom listener for your list Screenlet, however, then your activity or fragment must implement it instead. For example, recall that the example Bookmark List Screenlet's listener is `BookmarkListListener`. To use this Screenlet, you must therefore implement this listener in the class of the activity or fragment that you want to use the Screenlet in. For example:

   ```
   public class ListBookmarksActivity extends AppCompatActivity
       implements BookmarkListListener {...
   ```

   See the full example of this here in GitHub.

Well done! Now you know how to create list Screenlets.

Related Topics

Creating Android Screenlets

Architecture of Liferay Screens for Android

Packaging Your Android Screenlets

Using Views in Android Screenlets

Using Screenlets in Android Apps

76.9 Creating Android Views

By creating your own Views, you can customize your mobile app's layout, style, and functionality. You can create them from scratch or use an existing View as a foundation. Views include a View class for implementing Screenlet behavior, a Screenlet class for notifying listeners and invoking Interactors, and an XML file for specifying the UI. The four Liferay Screens View types support different levels of customization and parent View inheritance. Here's what each View type offers:

Themed View: presents the same structure as the current View, but alters the theme colors and tints of the View's resources. All existing Views can be themed with different styles. The View's colors reflect the current value of the Android color palette. If you want to use one View Set with another View Set's colors, you can use those colors in your app's theme (e.g. `colorPrimary_default`, `colorPrimary_material`, `colorPrimary_westeros`).

Child View: presents the same UI components as its parent View, but lets you change their appearance and position.

Extended View: inherits its parent View's functionality and appearance, but lets you add to and modify both.

Full View: provides a complete standalone View for a Screenlet. A full View is ideal for implementing completely different functionality and appearance from a Screenlet's current theme.

This tutorial explains how to create all four types of Views. To understand View concepts and components, you might want to examine the architecture of Liferay Screens for Android. And the tutorial Creating Android Screenlets can help you create or extend any Screenlet classes your View requires. Now get ready to create some great Views!

Determining Your View's Location

First, decide whether you'll reuse your view or if it's just for your current app. If you don't plan to reuse it in another app or don't want to redistribute it, create it in your app project.

If you want to reuse your View in another app, create it in a new Android application module; the tutorial Packaging Android Screenlets explains how. When your View's project is in place, you can start creating it.

First, you'll learn how to create a Themed View.

Themed View

Screens provides several existing View Sets that you can reuse and customize in your app to create a Themed View. If you use or override the Android color palette's values (for example, `primaryColor`, `secondaryColor`, etc...), you'll reuse the View Set's general structure, but be able to use the new colors (also with tinted resources). Note that you must create Themed Views inside your app. This is because Themed Views depend on the app or activity theme.

Each View Set has its own Android theme. These are listed here:

- **Default View Set:** `default_theme`
- **Material View Set:** `material_theme`
- **Westeros View Set:** `westeros_theme`

You can easily style all your Screenlets by setting your app or activity theme to inherit a View Set's Android theme. For example, you can use the following code to reuse the styles (and layouts) from `material_theme` in your own theme:

```
<style name="AppTheme.NoActionBar" parent="material_theme">
    <item name="colorPrimary">#B91D6D</item>
    <item name="colorPrimaryDark">#670E3B</item>
    <item name="colorAccent">#BBBBBB</item>
</style>

<application android:theme="@style/AppTheme.NoActionBar"
    ...
>
```

Note that this code overrides the `AppTheme.NoActionBar` theme's colors with your own color settings for `colorPrimary`, `colorPrimaryDark`, and `colorAccent`. Screenlets will also use these new colors, and tint images and other resources accordingly. Liferay Screens uses the default Android color palette names from the Support Library.

You can also override only the parent View Set's theme colors. This way you can set a default color palette and override only the View Set colors you want. The color names for each View Set are the default Android names, followed by an underscore and the View Set's lowercase name (`_default`, `_material`, and `_westeros`). For example, the following code overrides `colorPrimary`, `colorPrimaryDark`, and `colorAccent` for only the `material_theme`:

```
<resources>
    <color name="colorPrimary_material">#B91D6D</color>
    <color name="colorPrimaryDark_material">#670E3B</color>
    <color name="colorAccent_material">#BBBBBB</color>
</resources>
```

Liferay Screens also lets you use one View Set's layout with a Screenlet, and use another View Set's general style and colors. To do this, pass a `layoutId` attribute to a Screenlet that is already styled with another View Set's theme. The Screenlet uses the layout structure specified in `layoutId`, but inherits the general style and colors from the View Set's theme. For example, this code tells Login Screenlet to use the Default View Set's layout structure, but use the styles and colors defined earlier in `AppTheme.NoActionBar`:

```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:basicAuthMethod="email"
    app:layoutId="@layout/login_default"
    app:credentialsStorage="shared_preferences" />

<application android:theme="@style/AppTheme.NoActionBar"
    ...
>
```

Next, you'll learn how to create a Child View.

Child View

A Child View presents the same behavior and UI components as its parent, but can change the UI components' appearance and position. It can't add or remove any UI components. A Child View specifies visual changes in its own layout XML file; it inherits the parent's View class and Screenlet class. The parent must be a Full View.

The Child View discussed here presents the same UI components as the Login Screenlet's Default View, but uses a more compact layout.

You can follow these steps to create a Child View:

1. Create a new layout XML file named after the View's Screenlet and its intended use case. A good way to start building your UI is to duplicate the parent's layout XML file and use it as a template. However

you start building your UI, name the root element after the parent View's fully-qualified class name and specify the parent's UI components with the same IDs.

In the example here, the Child View's layout file `login_compact.xml` resembles its parent's layout file `login_default.xml`– the layout of the Login Screenlet's Default View. The child View's name *compact* describes its use case: display the Screenlet's components in a more compact layout. The IDs of its `EditText` and `Button` components match those of the parent View. Its root element uses the parent View class's fully-qualified name:

```
<?xml version="1.0" encoding="utf-8"?>
<com.liferay.mobile.screens.viewsets.defaultviews.auth.login.LoginView
    xmlns:android="http://schemas.android.com/apk/res/android"
    style="@style/default_screenlet">

    <EditText
        android:id="@+id/liferay_login"
        style="@style/default_edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="15dp"
        android:drawableLeft="@drawable/default_mail_icon"
        android:hint="@string/email_address"
        android:inputType="text" />

    <EditText
        android:id="@+id/liferay_password"
        style="@style/default_edit_text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="15dp"
        android:drawableLeft="@drawable/default_lock_icon"
        android:hint="@string/password"
        android:inputType="textPassword" />

    <Button
        android:id="@+id/liferay_login_button"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        style="@style/default_button"
        android:text="@string/sign_in" />

</com.liferay.mobile.screens.viewsets.defaultviews.auth.login.LoginView>
```

You can browse other layouts for Screens's Default Views on GitHub.

2. Insert your View's Screenlet in any of your activities or fragments, using your new layout's name as the `liferay:layoutId` attribute's value. For example, to use the new `login_compact` layout, insert `LoginScreenlet` in an activity or fragment and set `liferay:layoutId="@layout/login_compact"`.

Another good Child View layout file to examine is `sign_up_material.xml`. It presents the same UI components and functionality as the Sign Up Screenlet's Default View, but using Android's Material design.

Stupendous! Now you know how to create Child Views. Next, you'll learn how to create Extended Views.

Extended View

An Extended View inherits the parent View's behavior and appearance, but lets you change and add to both. You can do so by writing a custom View class and a new layout XML file. An Extended View inherits all of the parent View's other classes, including its Screenlet, listeners, and Interactors. An Extended View's parent must be a Full View.

The example Extended View discussed here presents the same UI components as the Login Screenlet's Default View, but adds functionality: computing password strength. Of course, you're not restricted to password strength computations; you can implement anything you want.

1. Create a new layout XML file named after the View's Screenlet and its intended use case. A good way to start building your UI is to duplicate the parent's layout XML file and use it as a template. The new layout file for the Login Screenlet's Extended View is called `login_password.xml`, because it's based on the Login Screenlet's Default View layout file `login_default.xml` and it adds a password strength computation.

2. Create a new custom View class that extends the parent View class. Name it after the Screenlet and the functionality you'll add or override. The example View class `LoginCheckPasswordView` extends the Default View's `LoginView` class, overriding the `onClick` method to compute password strength:

```
public class LoginCheckPasswordView extends LoginView {

    // parent's constructors go here...

    @Override
    public void onClick(View view) {
        // compute password strength

        if (passwordIsStrong) {
            super.onClick(view);
        }
        else {
            // Present user message
        }
    }

}
```

3. Rename the layout XML file's root element after your custom View's fully-qualified class name. For example, the root element in `login_password.xml` is `com.your.package.LoginCheckPasswordView`:

```
<com.your.package.LoginCheckPasswordView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
    ...
```

4. Insert your View's Screenlet in any of your activities or fragments, using your new layout's name as the `liferay:layoutId` attribute's value. For example, to use the new `login_password` layout, insert `LoginScreenlet` in an activity or fragment, and set `liferay:layoutId="@layout/login_password"`.

The Bank of Westeros sample app's Westeros View Set has a couple of Extended Views that you can examine. It has an Extended View that adds a new button to show the password in the clear for the Login Screenlet. The View uses custom layout file `login_westeros.xml` and custom View class `LoginView`. The Westeros View Set also contains an Extended View for the User Portrait Screenlet; it changes the border color and width of the user's portrait picture and it uses the custom layout file `userportrait_westeros.xml` and the custom View class `UserPortraitView`.

Awesome! Now you know how to create Extended Views. Next, you can learn how to create a Full View.

Full View

A Full View has a unique Screenlet class, a View class, and layout XML file. It's standalone and doesn't inherit from any View. You should create a Full View if there's no other View that you can extend to meet your needs or if your Screenlet's behavior can only be augmented by customizing its listeners or calling custom Interactors. To create a Full View, you must create its Screenlet class, View class, and layout XML file. The example Full View here for the Login Screenlet presents a single `EditText` component for the user name. For the password, it uses `Secure.ANDROID_ID`. The Screens Test App uses this Full View.

You can follow these steps to create a Full View:

1. Create a new layout XML file and build your UI in it. A good way to start building your UI is to duplicate another View's layout XML file and use it as a template. Name your layout XML file after the View's Screenlet and intended use case. Name its root element after the fully-qualified class name of your custom View (you'll create this next).

   The Test App's Full View layout XML file for the Login Screenlet is called `login_full.xml`. It specifies `EditText` and `Button` elements copied from the LongScreenlet's Default View file `login_default.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<com.your.package.LoginFullView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical">

<EditText
        android:id="@+id/liferay_login"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="20dp"
        android:hint="Email Address"
        android:inputType="textEmailAddress"/>

<Button
        android:id="@+id/liferay_login_button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Sign In"/>

</com.your.package.LoginFullView>
```

2. Create a new custom View class named after the layout's root element. The tutorial on creating Android Screenlets explains how to create a View class. Note that you don't have to extend a View class to implement a View Model interface, but you might want to for convenience. The custom View class `LoginFullView`, for example, implements the `LoginViewModel` interface by extending the Default `LoginView` class. To return the `ANDROID_ID`, the `LoginFullView` custom View class overrides the `getPassword()` method.

3. Create a new Screenlet class that inherits the base Screenlet class. This new class is where you can add custom behavior to the listeners or call custom Interactors. The Screenlet class `LoginFullScreenlet`, for example, extends `LoginScreenlet` and overrides the `onUserAction` method to log Interactor calls.

4. Insert your View's Screenlet in any of your activities or fragments, using your new layout's name as the `liferay:layoutId` attribute's value. For example, to use the new `login_password` layout, insert `LoginScreenlet` in an activity or fragment, and set `liferay:layoutId="@layout/login_password"`.

The Westeros View Set's full view for the Sign Up Screenlet uses a custom Screenlet class to add a new listener. The custom Screenlet class also adds a new user action that calls the base Interactor `SignUpInteractor`.

Sweetness! Now you know how to create a Full View. Next, you'll learn how to package Views for distribution.

Packaging Your Views

If you want to distribute or reuse Views, you should package them in a module that is then added as an app's project dependency. To do this, use the material sub-project as a template for your new `build.gradle` file.

To use a packaged View, you must import its module into your project by specifying its location in your `settings.gradle` file. The Bank of Westeros and test-app projects use custom Views `westeros` and `material`, respectively. These projects exemplify using independent Views in a project.

If you want to redistribute your View and let others use it, you can upload it to jCenter or Maven Central. In the example `build.gradle` file, after entering your bintray api key you can execute `gradlew bintrayupload` to upload your project to jCenter. When finished, anyone can use the View as they would any Android dependency by adding the repository, artifact, group ID, and version to their Gradle file.

Super! Now you know how to create and package Views in Liferay Screens for Android. This gives you extensive control over your app's visual design and behavior and also lets you distribute and reuse your Views.

Related Topics

Using Views in Android Screenlets

Architecture of Liferay Screens for Android

Creating Android Screenlets

76.10 Packaging Your Android Screenlets

To reuse your Screenlet in another app or distribute it, you can package it in a module (Android library). You can optionally share it with other developers via jCenter or Maven Central. Developers can then use your Screenlet by adding its module as a project dependency in their apps. This tutorial explains how to package and distribute Screenlets by following these steps:

1. Create a new Android module.
2. Configure dependencies between each module.
3. Distribute the module by uploading it to jCenter or Maven Central.

Now get ready to package and distribute Screenlets like a pro!

Create a New Android Module

Android Studio's *Create New Module* wizard can automatically create a module and add it to your `settings.gradle` file. Go to *File → New Module...*, select *Android Library* in the More Modules section, and click *Next*. Then name your module and click *Next*. The wizard's final step lets you add a new activity. Since your module doesn't need one, select *Blank Activity* and click *Finish*. Android Studio creates a new `build.gradle` file with an Android Library configuration and adds the new module to your `settings.gradle` file.

If you prefer to create a new module manually, examine the `build.gradle` file from the Material View set or Westeros app as an example. After creating the module, import it into your project by specifying its location in `settings.gradle`. Here's an example configuration:

```
// Change YOUR_MODULE_NAME and RELATIVE_ROUTE_TO_YOUR_MODULE to match your module

include ':YOUR_MODULE_NAME'
project(':YOUR_MODULE_NAME').projectDir = new File(settingsDir, 'RELATIVE_ROUTE_TO_YOUR_MODULE')
```

Now that you have a module, you're ready to configure its dependencies.

Configure Dependencies Between Each Module

Next, you must configure your app to use the module. To do so, add a project implementation statement to your `build.gradle` file's dependencies:

```
// Change YOUR_MODULE_NAME to match your module's name

dependencies {
    ...
    implementation project (':YOUR_MODULE_NAME')
    ...
}
```

Your module must also specify dependencies for overriding existing Screenlets and creating new ones. This usually requires adding Liferay Screens and the View Sets your Screenlet currently uses to your `build.gradle` file's dependencies. To add Liferay Screens as a dependency, add to your `build.gradle` file's dependencies the following project implementation statement:

```
implementation 'com.liferay.mobile:liferay-screens:+'
```

Awesome! Now you're ready to share your Screenlet with the world!

Upload the Module to jCenter or Maven Central

To make your module available to anyone, you can upload your module to jCenter or Maven Central. Before doing so, you must configure your `build.gradle` file appropriately for those repositories. Use the material or Westeros View Set's `build.gradle` file as an example. After entering your bintray api key, execute `gradlew bintrayupload` to upload your project to jCenter. Developers can then use your Screenlet as any other Android dependency by specifying its repository, artifact, group ID, and version in their Gradle files. Congratulations on publishing your Screenlet!

Related Topics

Creating Android Screenlets
  Preparing Android Projects for Liferay Screens
  Using Screenlets in Android Apps
  Creating Android Views

76.11 Using Liferay Push in Android Apps

Liferay Screens supports push notifications in Android apps. To use them, you must configure some APIs and modify your app to consume and/or produce push notifications. This tutorial shows how to do all these things.

Configuring to Use Liferay Push Notifications

Your first step is to create and configure a Google project to use Google Cloud Messaging (GCM). You also need to configure the Liferay Push app to use the project's GCM API.

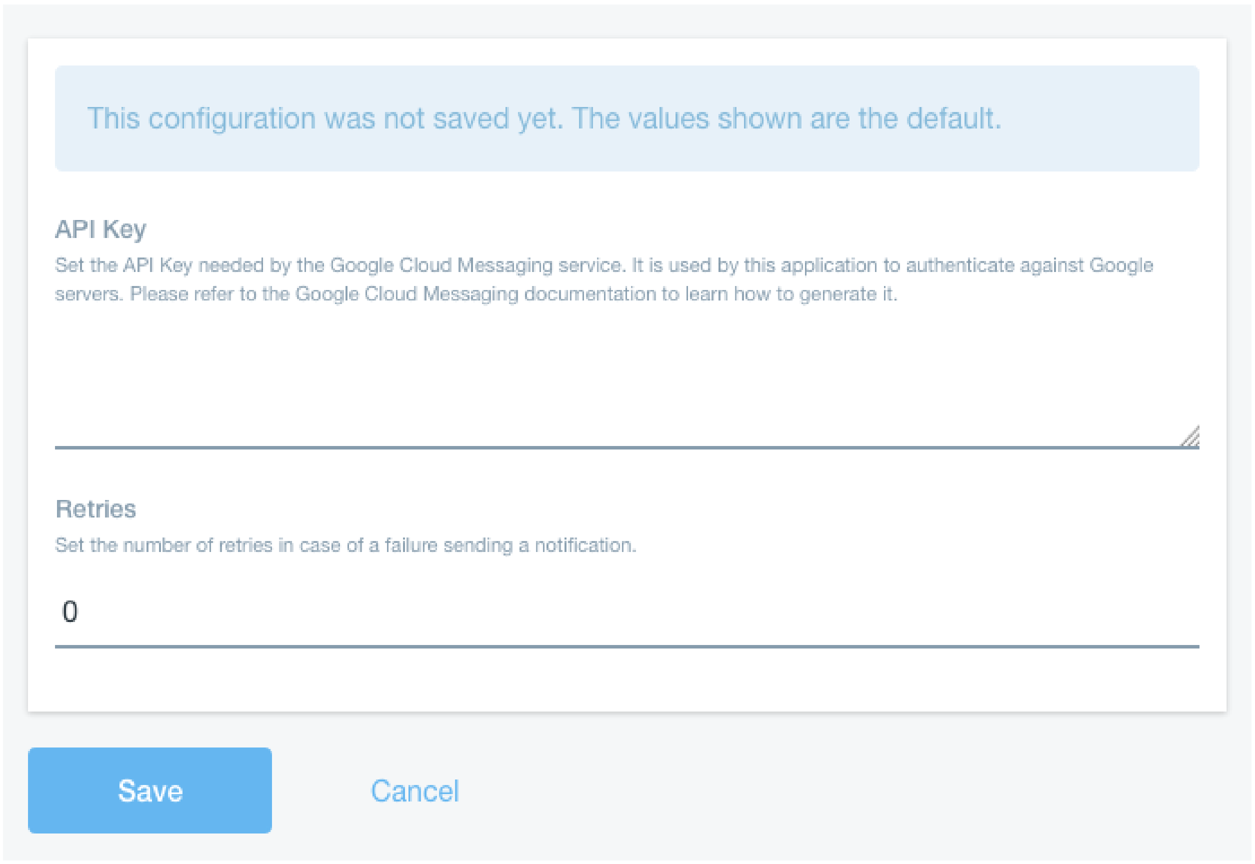
Follow these steps to create and configure a Google project to support cloud messaging:

1. On the Google Cloud Messaging page, create a configuration file by clicking *Get a Configuration File*. On the screen that appears, set your *App name* and *Android package name*, and then click *Continue To Choose and Configure Services*. On the next screen, click *Enable Google Cloud Messaging*.

2. Copy and save the *Server API Key* and *Sender ID* values you're presented with. You'll need to use these values later as the push notifications API keys for Liferay Push.

   `![ You need the Server API Key and Sender ID to enable Liferay Push.](./images/screens-android-push-project-and-server-key.png)`

Now that you've set up your Google project, you can configure the Liferay Push app to use the project's GCM API. Install the Liferay Push app from the Liferay Marketplace. In your Liferay DXP instance's Control Panel, navigate to *Configuration* → *System Settings*, select the *Other* tab, then select *Android Push Notifications Sender*. Set the push notifications *API Key* to the value of the Server API Key you generated in your Google project. You can also set the number of retries in the event that sending a notification fails.


This configuration was not saved yet. The values shown are the default.

API Key

Set the API Key needed by the Google Cloud Messaging service. It is used by this application to authenticate against Google servers. Please refer to the Google Cloud Messaging documentation to learn how to generate it.

Retries

Set the number of retries in case of a failure sending a notification.

0

Figure 76.16: Set the API key and number of retries in your Liferay DXP instance.

Great! Your Liferay DXP instance is now ready to send push notifications to your Android apps!

Receiving and Sending Push Notifications

The Liferay Push Client for Android streamlines registering a device with the portal for receiving and sending push notifications. Although the information below contains the main steps needed to use the client, the readme explains them in detail.

In your Android application's Gradle build file, add a new dependency on the Liferay Push Client for Android:

```
dependencies {

    ...

    implementation 'com.liferay.mobile:liferay-push:1.2.1'
}
```

Make sure your app's `liferay-plugin-package.properties` file specifies the Push Notifications portlet as a required deployment context:

```
required-deployment-contexts=\
    push-notifications-portlet\
    ...
```

Next, you'll learn how to register listeners for push notifications.

Receiving Push Notifications

First, register your device in GCM with the `SENDER_ID` you generated previously:

```
Session session = new SessionImpl(YOUR_SERVER, new BasicAuthentication(YOUR_USER, YOUR_PASSWORD));

Push.with(session).register(this, YOUR_SENDER_ID);
```

If you're using Liferay Screens to maintain a session, you can retrieve it and use it instead of creating a new one:

```
Push.with(SessionContext.createSessionFromCurrentSession()).register(this, YOUR_SENDER_ID);
```

If you use these example lines of code, make sure to replace `YOUR_SERVER`, `YOUR_USER`, `YOUR_PASSWORD`, and `YOUR_SENDER_ID` with your own values.

That's it! You can attach a listener to store the registration ID or to process the notification sent to the activity (using `onPushNotification()`). You can also register a receiver and service to process the notification. You can refer to the Push Notifications project as an example push notifications implementation.

Next, you'll learn how to send push notifications.

Sending Push Notifications

Using the Liferay Push app, sending notifications to your app's users is straightforward. You can specify the user IDs along with the message content:

```
PushNotificationsDeviceLocalServiceUtil.sendPushNotification(userIds, content);
```

This example hook plugin sends a push notification each time a user creates a new DDL record or updates an existing one.

In your app's `portal.properties` file, you can add a listener for a class by creating a *value.object.listener* property, set to a comma separated list of intended listener classes. Here's an example listener setting for `DDLRecord` objects:

```
value.object.listener.com.liferay.portlet.dynamicdatalists.model.DDLRecord=com.liferay.push.hooks.DDLRecordModelListener
```

Great! Now you know how to configure your Android apps to receive push notifications from Liferay DXP.

In this tutorial, you've configured your portal to accommodate push notifications, registered notification listeners, and implemented sending push notifications. Way to go!

Related Topics

Preparing Android Projects for Liferay Screens

Using Screenlets in Android Apps

76.12 Accessing the Liferay Session in Android

A session is a conversation state between the client and server. It typically consists of multiple requests and responses between the two. To facilitate this communication, the session must have the server IP address, and a user's login credentials. Liferay Screens uses a Liferay Session to access and query the JSON web services provided by Liferay Portal. When you log in using a Liferay Session, the portal returns the user's information (name, email, user ID, etc…). Screens stores this information and the active Liferay Session in Screens's `SessionContext` class.

The `SessionContext` class is very powerful and lets you use Screens in many different scenarios. For example, you can use `SessionContext` to request information with the JSON WS API provided by Liferay. You can also use `SessionContext` to create anonymous sessions, or to log in a user without showing a Login Screenlet.

This tutorial explains some common `SessionContext` use cases, and and also describes the class's most important methods.

Creating a Session from an Existing Session

When working with Liferay Screens, you may wish to call the remote JSON web services provided by the Liferay Mobile SDK. Every operation with the Liferay Mobile SDK needs a Liferay Session to provide the server address, user credentials, and any other required parameters. Since the Login Screenlet creates a session when a user successfully logs in, you can retrieve this session with the `SessionContext` method `createSessionFromCurrentSession()`. You can then use that session to make the Mobile SDK service call. The following example shows this for calling the Mobile SDK's `BookmarksEntryService`:

```
Session sessionFromCurrentSession = SessionContext.createSessionFromCurrentSession();
sessionFromCurrentSession.setCallback(callback);

new BookmarksEntryService(sessionFromCurrentSession).methodCall()
```

If you need to check first to see if a user has logged in, you can use `SessionContext.isLoggedIn()`.

Great! Now you know how to retrieve an existing session in your app. But what if you're not using the Login Screenlet? There won't be an existing session to retrieve. No sweat! You can still use `SessionContext` to create one manually. The next section shows you how to do this.

Creating a Session Manually

If you don't use the Login Screenlet, then SessionContext doesn't have a session for you to retrieve. In this case, you must create one manually. You can do this with the SessionContext method createBasicSession. The method takes a username and password as parameters, and creates a session with those credentials. If you also need to access a user's information, you must manually call the User JSON web service, or call SessionContext.setLoggedUser(). The following code creates a session with createBasicSession and then uses setLoggedUser to set the user in SessionContext:

```
LiferayScreensContext.init(this);

Session session = SessionContext.createBasicSession(USERNAME, PASSWORD);
SessionContext.setLoggedUser(USER);
```

Note that you can achieve the same thing by calling the interactor directly:

```
LoginBasicInteractor loginBasicInteractor = new LoginBasicInteractor(0);
loginBasicInteractor.onScreenletAttached(this);
loginBasicInteractor.setLogin(USERNAME);
loginBasicInteractor.setPassword(PASSWORD);
loginBasicInteractor.login();
```

Super! Now you know how to create a session manually. The next section shows you how to implement auto-login, and save or restore a session.

Implementing Auto-login and Saving or Restoring a Session

Although the Login Screenlet is awesome, your users may not want to enter their credentials every time they open your app. It's very common for apps to only require a single login. To implement this in your app, see this video.

In short, you need to pass a storage type to the Login Screenlet, and then use SessionContext.isLoggedIn() to check for a session. If a session doesn't exist, load the stored session from CredentialsStorage with loadStoredCredentials(StorageType storageType). The following code shows a typical implementation of this:

```
LiferayScreensContext.init(this); // If you haven't called a Screenlet yet
SessionContext.loadStoredCredentials(SHARED_PREFERENCES);

if (SessionContext.isLoggedIn()) {
    // logged in
    // consider doing a relogin here (see next section)
} else {
    // send user to login form
}
```

Awesome! Now you know how to implement auto-login in your Liferay Screens apps. For more information on available SessionContext methods, see the Methods section at the end of this tutorial. Next, you'll learn how to implement relogin for cases where a user's credentials change on the server while they're logged in.

Implementing Relogin

A session, whether created via Login Screenlet or auto-login, contains basic user data that verifies the user in the Liferay instance. If that data changes in the server, then your session is outdated, which may cause your app to behave inconsistently. Also, if a user is deleted, deactivated, or otherwise changes their credentials in the server, the auto-login feature won't deny access because it doesn't perform server transactions: it only retrieves an existing session from local storage. This isn't an optimal situation!

For such scenarios, you can use the relogin feature. This feature is implemented in a simple method that determines if the current session is still valid. If the session is still valid, the user's data is updated with the most recent data from the server. If the session isn't valid, the user is logged out and must then log in again to create a new session.

To use this feature, call the `SessionContext` method `relogin`, with an object that implements the `LoginListener` interface as an argument:

```
SessionContext.relogin(listener);
```

This method handles success or failure via the listener's `onLoginSuccess` and `onLoginFailure` methods, respectively. Note that this operation is done asynchronously in a background thread, so the listener is called in that thread. If you also want to perform any UI operations, you must do so in your UI thread. For example:

```
@Override
public void onLoginSuccess(final User user) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            // do any UI operation here
        }
    });
}
```

Great! Now you know how to implement relogin in your app. You've also seen how handy `SessionContext` can be. It can do even more! The next section lists some additional `SessionContext` methods, and some more detail on the ones used in this tutorial.

Methods

Method | Return Type | Explanation | `logout()` | `void` | Clears the stored user attributes and session. | `relogin(LoginListener)` | `void` | Refreshes user data from the server. This recreates the `currentUser` object if successful, or calls `logout()` on failure. When the server data is received, the listener method `onLoginSuccess` is called with received user's attributes. If an error occurs, the listener method `onLoginFailure` is called. | `isLoggedIn()` | `boolean` | Returns `true` if there is a stored Liferay Session in `SessionContext`. | `createBasicSession(String username, String password)` | `Session` | Creates a Liferay Session using the default server and the supplied username and password. | `createSessionFromCurrentSession()` | `Session` | Creates a Liferay Session based on the stored credentials and server. | `getCurrentUser()` | `User` | Returns a `User` object containing the server attributes of the logged-in user. This includes the user's email, user ID, name, and portrait ID. | `storeCredentials(StorageType storageType)` | `void` | Stores the current session in the `StorageType` supplied as a parameter. | `removeStoredCredentials(StorageType storageType)` | `void` | Clears the `StorageType` of any user information and session. | `loadStoredCredentials(StorageType storageType)` | `void` | Loads the session and user information from the `StorageType` parameter, and uses it as the current session and user. |

For more information, see the `SessionContext` source code in GitHub.

Related Topics

Login Screenlet for Android

Using Screenlets in Android Apps

76.13 Adding Custom Interactors to Android Screenlets

Interactors are Screenlet components that implement server communication for a specific use case. For example, the Login Screenlet's interactor calls the Liferay Mobile SDK service that authenticates a user to the portal. Similarly, the interactor for the Add Bookmark Screenlet calls the Liferay Mobile SDK service that adds a bookmark to the Bookmarks portlet.

That's all fine and well, but what if you want to customize a Screenlet's server call? What if you want to use a different back-end with a Screenlet? No problem! You can implement a custom interactor for the Screenlet. You can plug in a different interactor that makes its server call by using custom logic or network code. To do this, you must implement the current interactor's interface and then pass it to the Screenlet you want to override. You should do this inside your app's code, either in an inner class or a separate class.

In this tutorial, you'll see an example interactor that overrides the Login Screenlet to always log in the same user, without a password. You can find the complete code in the test-app on GitHub.. Note that this example implements the custom interactor in an inner class of an activity.

Implementing a Custom Interactor

1. Implement your custom interactor. You must inherit the original interactor's interface, as shown here:

```
private class CustomLoginInteractor extends LoginBasicInteractor {

    public CustomLoginInteractor(int targetScreenletId) {
        super(targetScreenletId);
    }

    @Override
    public void login() throws Exception {
        //custom implementation
    }
}
```

2. Call the interactor's listener. In your custom logic, you must call the interactor's listener. In this example, you must call `onLoginFailure` and `onLoginSuccess`, depending on your custom logic's result:

```
if (SUCCESS) {
    getListener().onLoginSuccess(fakeUser);
}
else {
    getListener().onLoginFailure(new AuthenticationException("bad login"));
}
```

3. Return your interactor in the custom listener. You must use `setCustomInteractorListener` to set a specific listener that expects an Interactor created with `actionName` (a string):

```
_screenlet.setCustomInteractorListener(this);

@Override
public LoginInteractor createInteractor(String actionName) {
    return new CustomLoginInteractor(_loginScreenlet.getScreenletId());
}
```

Great! Now you know how to implement custom interactors for Android Screenlets. The next example builds on this by showing you how to access non-Liferay backends with a custom interactor.

Using Custom Interactors to Access Other Backends

Custom interactors are also capable of communicating with non-Liferay backends. The following example illustrates this by creating a custom interactor for the Add Bookmark Screenlet that can store bookmarks at Delicious. You can find this example's complete code at this gist.

1. Create a new custom interactor. This interactor inherits `BaseRemoteInteractor`, the base class of all interactors, with `AddBookmarkListener` as a type parameter. It also implements the `AddBookmarkInteractor` class. The base code for this new interactor is shown here:

```
public class AddDeliciousInteractorImpl extends BaseRemoteInteractor<AddBookmarkListener>
    implements AddBookmarkInteractor {

    public AddDeliciousInteractorImpl(int targetScreenletId) {
        super(targetScreenletId);
    }

    public void addBookmark(final String url, final String title, long folderId) throws Exception {
        ...
    }
}
```

2. Implement your custom logic. In this example, you must implement the code for accessing Delicious and inserting a new bookmark with the Delicious API. You can use the OkHttp library to pass the API your bookmark's URL and description. The following code shows this:

```
new Thread(new Runnable() {
    @Override
    public void run() {
        try {

            Headers headers = Headers.of("Authorization", "Bearer _OAUTH_TOKEN_");

            OkHttpClient client = new OkHttpClient();

            Request add = new Request.Builder()
                .url("https://api.del.icio.us/api/v1/posts/add?url=" + url + "&description=" + title)
                .headers(headers)
                .build();

            com.squareup.okhttp.Response response = client.newCall(get).execute();

            String text = response.body().string();

            ...

        }
        catch (IOException e) {
            LiferayLogger.e("Error sending", e);
            ...
        }
    }
}).start();
```

3. Notify your app of the results. You should use the `EventBusUtil` class to post an event for this. Use the event to let other classes listen for the event. The following code uses `EventBusUtil.post(text)` to post the event, and the `onEvent` method to notify the listener:

```
EventBusUtil.post(text);

...

public void onEvent(String text) {
    getListener().onAddBookmarkSuccess();
}
```

Note that the code in the gist uses the custom `BookmarkAdded` class to model the operation's results.

4. In the activity or fragment you're using the Screenlet in, implement `CustomInteractorListener`. You must also reference your new custom interactor and connect it to the Screenlet:

```
_screenlet.setCustomInteractorListener(this);

@Override
public Interactor createInteractor(String actionName) {
    return new AddDeliciousInteractorImpl(_screenlet.getScreenletId());
}
```

Awesome! Now you know how to create a custom interactor that can communicate with a non-Liferay backend. This opens up even more possibilities for your apps.

Related Topics

Architecture of Liferay Screens for Android

Creating Android Screenlets

76.14 Rendering Web Content in Your Android App

Liferay DXP represents web content articles as `JournalArticle` entities. Liferay Screens provides several ways to render these entities in your apps.

The simplest way to display a `JournalArticle`'s HTML in your app is to use Web Content Display Screenlet. This Screenlet is very powerful and supports several complex use cases to fit your needs. You can also use Web Content List Screenlet to display lists of web content articles. This tutorial shows you how to use both Screenlets to display web content in your apps.

Retrieving Basic Web Content

Web Content Display Screenlet's simplest use case is to render a `JournalArticle`'s HTML in an Android `WebView`. Simply provide the `JournalArticle`'s `articleId` in the Screenlet XML, and the Screenlet takes care of the rest (including decorating itself with the CSS needed to render it in a small display). The following Screenlet XML shows this:

```
<com.liferay.mobile.screens.webcontent.display.WebContentDisplayScreenlet
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:articleId="YOUR_ARTICLE_ID" />
```

To render the content *exactly* as it appears on your mobile site, however, you must provide the CSS inline or use a template. The HTML returned isn't aware of a Liferay instance's global CSS.

You can also use a listener to modify the HTML, as explained in the Screenlet reference documentation.

In the default security policy, an Android `WebView` doesn't execute a page's JavaScript. You can enable such JavaScript execution by setting the `javascriptEnabled` property via XML:

```
<com.liferay.mobile.screens.webcontent.display.WebContentDisplayScreenlet
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:articleId="YOUR_ARTICLE_ID"
    app:javascriptEnabled="true" />
```

Alternatively, you can set this property in your app's fragment or activity class that contains the Screenlet:

```
...
screenlet.setJavascriptEnabled(true);
...
```

You can also use the `isJavascriptEnabled()` method to check this property's setting.

As you can see, this is all straightforward. What could go wrong? Famous last words. A common mistake is to use the default `groupId` instead of the one for the site that contains your `JournalArticle` entities.

If you need to use a default `groupId` in the rest of your app, but render another site's HTML, you can set the Web Content Display Screenlet's `groupId` with the `app:groupId` attribute. You can alternatively use the `setGroupId` method in the activity or fragment code that uses the Screenlet.

Using Templates

Web Content Display Screenlet can also use templates to render `JournalArticle` entities. For example, your Liferay instance may have a custom template specifically designed to display content on mobile devices.

To use a template, specify its ID in the Screenlet XML's `templateId` property:

```
<com.liferay.mobile.screens.webcontent.display.WebContentDisplayScreenlet
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    app:articleId="YOUR_ARTICLE_ID"
    app:templateId="YOUR_TEMPLATE_ID" />
```

Using Structures

Since mobile devices have limited screen space, you must often display only the most important parts of a web content article. If your web content is structured, you can do this by using Web Content Display Screenlet to display only specific fields from a `JournalArticle`'s structure. The simplest way to do this is to specify the structure's ID and a comma-delimited list of fields in the Screenlet XML's `structureId` and `labelFields` attributes, respectively. The following example illustrates this:

```
<com.liferay.mobile.screens.webcontent.display.WebContentDisplayScreenlet
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    liferay:articleId="YOUR_ARTICLE_ID"
    liferay:labelFields="YOUR_LABELS"
    liferay:layoutId="@layout/webcontentdisplay_structured_default"
    liferay:structureId="YOUR_STRUCTURE_ID" />
```

You can also use your own layout to render the structure fields exactly how you want. To do this, your layout should inherit from `WebContentStructuredDisplayView` and read the information parsed and stored in the `webContent` entity. By displaying two structure fields with such a custom layout, the test app contains a complete example of this:

1. The layout file `webcontentdisplaystructured_example.xml` defines the custom layout:

```
<com.liferay.mobile.screens.testapp.webviewstructured.WebContentDisplayView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/web_content_first_field"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:color/holo_red_light" />

    <TextView
        android:id="@+id/web_content_second_field"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@android:color/holo_green_light" />

</com.liferay.mobile.screens.testapp.webviewstructured.WebContentDisplayView>
```

2. The `WebContentDisplayView` class sets the custom layout's functionality:

```
public class WebContentDisplayView extends WebContentStructuredDisplayView {

    ...

    @Override
    public void showFinishOperation(WebContent webContent) {
        super.showFinishOperation(webContent);

        DDMStructure ddmStructure = webContent.getDDMStructure();

        TextView firstField = (TextView) findViewById(R.id.first_field);
        firstField.setText(String.valueOf(ddmStructure.getField(0).getCurrentValue()));

        TextView secondField = (TextView) findViewById(R.id.second_field);
        secondField.setText(String.valueOf(ddmStructure.getField(1).getCurrentValue()));
    }
}
```

3. The Screenlet XML's `layoutId` attribute specifies the custom layout to use:

```
<com.liferay.mobile.screens.webcontent.display.WebContentDisplayScreenlet
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    liferay:articleId="@string/liferay_article_structured_article_id"
    liferay:labelFields="@string/liferay_article_structured_label_fields_first_field"
    liferay:layoutId="@layout/webcontentdisplaystructured_example"
    liferay:offlinePolicy="REMOTE_FIRST"
    liferay:structureId="@string/liferay_article_structured_structure_id" />
```

Great! Now you know how to use structured web content with Web Content Display Screenlet. Next, you'll learn how to display a list of web content articles in your app.

Displaying a List of Web Content Articles

The preceding examples show you how to use Web Content Display Screenlet to display a single web content article's contents in your app. But what if you want to display a list of articles instead? No problem! You can use Web Content List Screenlet for this. Web Content List Screenlet can retrieve the contents of a web content folder and display only the labels you want. The Screenlet is also aware of structured content, so you can render each row with certain structure fields. You can also do this via a custom layout.

To use a web content folder with Web Content List Screenlet, specify the folder's ID in the Screenlet XML's `folderId` attribute. To render a specific structure field for each article in the list, specify that field in the Screenlet XML's `labelFields` attribute. For example:

```
<com.liferay.mobile.screens.webcontent.list.WebContentListScreenlet
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:folderId="YOUR_FOLDER_ID"
    app:labelFields="Text" />
```

You can also see an example of this in the test app's `web_content_display_list.xml` layout file.

Also note that several methods in Screens's `WebContent` class help you render content from different locales. For example, `getLocalized(name)` receives a field's name and returns the value in the mobile device's current locale. Such methods help you render a custom view without worrying about the underlying structure, XML parsing, or HTTP calls.

Displaying a List of Assets

To render a list of different assets in your app, including web content articles, you can use Asset List Screenlet. Asset List Screenlet can display a list of any assets from a Liferay instance. Like Web Content List Screenlet, you can also access a web content article's structure fields, or use a custom layout to render each asset type. For more information, see the reference documentation for Asset List Screenlet.

Related Topics

Using Screenlets in Android Apps

Using Views in Android Screenlets

Web Content Display Screenlet for Android

Web Content List Screenlet for Android

Asset List Screenlet for Android

76.15 Rendering Web Pages in Your Android App

The Rendering Web Content tutorial shows you how to display web content from a Liferay DXP site in your Android app. Displaying content is great, but what if you want to display an entire page? No problem! Web Screenlet lets you display any web page. You can even customize the page by injecting local or remote JavaScript and CSS files. When combined with Liferay DXP's server-side customization features (e.g., Application Display Templates), Web Screenlet gives you almost limitless possibilities for displaying web pages in your Android apps.

In this tutorial, you'll learn how to use Web Screenlet to display web pages in your Android app.

Inserting Web Screenlet in Your App

Inserting Web Screenlet in your app is the same as inserting any Screenlet in your app:

1. Insert the Screenlet's XML in the layout of the activity or fragment you want to use the Screenlet in. Also be sure to set any attributes that you need. For a list of Web Screenlet's available attributes, see the Attributes section of the Web Screenlet reference doc.

   For example, here's Web Screenlet's XML with the Screenlet's `layoutId` and `autoLoad` attributes set to `web_default` and `false`, respectively:

```
<com.liferay.mobile.screens.web.WebScreenlet
    android:id="@+id/web_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layoutId="@layout/web_default"
    app:autoLoad="false"
    />
```

Note that `web_default` specifies the Screenlet's Default View, which is part of the Default View Set.

2. To use a View that is part of a View Set, like the Default View, the app or activity theme must inherit the theme that sets the View Set's styles. For the Default View Set, this is `default_theme`. For example, to set the app's theme to inherit `default_theme`, open `res/values/styles.xml` and set the base app theme's parent to `default_theme`. In this example, the base app theme is `AppTheme`:

```
<style name="AppTheme" parent="default_theme">
    ...
```

Next, you'll implement Web Screenlet's listener.

Implementing Web Screenlet's Listener

To use any Screenlet in an activity or fragment, you must also implement the Screenlet's listener in that activity or fragment's class. Web Screenlet's listener is `WebListener`. Follow these steps to implement `WebListener`:

1. Change the class declaration to implement `WebListener`, and import `com.liferay.mobile.screens.web.WebListener`:

```
...
import com.liferay.mobile.screens.web.WebListener;
...

public class YourActivity extends AppCompatActivity implements WebListener {...
```

2. Implement `WebListener`'s `onPageLoaded` method. This method is called when the Screenlet loads the page successfully. How you implement it depends on what (if anything) you want to happen upon page load. For example, this `onPageLoaded` implementation displays a toast message indicating success:

```
@Override
public void onPageLoaded(String url) {
    Toast.makeText(this, "Page load successful!", Toast.LENGTH_SHORT).show();
}
```

3. Implement `WebListener`'s `onScriptMessageHandler` method. This method is called when the Screenlet's `WebView` sends a message. The `namespace` argument is the source namespace key, and the `body` argument is the source namespace body. For example, this `onScriptMessageHandler` implementation parses data from the source namespace body if it matches a specific namespace, and then starts a new activity with that data via an intent:

```
@Override
public void onScriptMessageHandler(String namespace, String body) {
    if ("gallery".equals(namespace)) {
        String[] allImgSrc = body.split(",");
        int imgSrcPosition = Integer.parseInt(allImgSrc[allImgSrc.length - 1]);

        Intent intent = new Intent(getApplicationContext(), DetailMediaGalleryActivity.class);
```

```
            intent.putExtra("allImgSrc", allImgSrc);
            intent.putExtra("imgSrcPosition", imgSrcPosition);
            startActivity(intent);
        }
    }
```

4. Implement the `error` method. This method is called when an error occurs in the process. The `e` argument contains the exception, and the `userAction` argument distinguishes the specific action in which the error occurred. In most cases, you'll use these arguments to log or display the error. For example, this `error` implementation displays a toast message with the exception's message:

```
@Override
public void error(Exception e, String userAction) {
    Toast.makeText(this, "Bad things happened: " + e.getMessage(), Toast.LENGTH_LONG).show();
}
```

5. Get a `WebScreenlet` reference and set the activity or fragment class as its listener. To do so, import `com.liferay.mobile.screens.web.WebScreenlet` and add the following code to the end of the `onCreate` method:

```
WebScreenlet screenlet = (WebScreenlet) findViewById(R.id.web_screenlet);
screenlet.setListener(this);
```

   Note that the `findViewById` references the `android:id` value set in the Screenlet's XML.

Next, you'll use the same `WebScreenlet` reference to set the Screenlet's parameters.

Setting Web Screenlet's Parameters

Web Screenlet has `WebScreenletConfiguration` and `WebScreenletConfiguration.Builder` objects that supply the parameters the Screenlet needs to work. These parameters include the URL of the page to load and the location of any JavaScript or CSS files that customize the page. You'll set most of these parameters via `WebScreenletConfiguration.Builder`'s methods.

Note: For a full list of `WebScreenletConfiguration.Builder`'s methods, and a description of each, see the table in the Configuration section of Web Screenlet's reference doc.

To set Web Screenlet's parameters, follow these steps in the method that initializes the activity or fragment containing the Screenlet (e.g., `onCreate` in activities, `onCreateView` in fragments). You can, however, do this in other methods as needed.

1. Use `WebScreenletConfiguration.Builder(<url>)`, where `<url>` is the web page's URL string, to create a `WebScreenletConfiguration.Builder` object. If the page requires Liferay DXP authentication, then the user must be logged in via Login Screenlet or a `SessionContext` method, and you must provide a relative URL to the `WebScreenletConfiguration.Builder` constructor. For example, if such a page's full URL is `http://your.liferay.instance/web/guest/blog`, then the constructor's argument is `/web/guest/blog`. For any other page that doesn't require Liferay DXP authentication, you must supply the full URL to the constructor.

2. Call the `WebScreenletConfiguration.Builder` methods to set the parameters that you need.

```
**Note:** If the URL you supplied to the `WebScreenletConfiguration.Builder`
constructor is to a page that doesn't require Liferay DXP authentication, then
you must call the `WebScreenletConfiguration.Builder` method
`setWebType(WebScreenletConfiguration.WebType.OTHER)`. The default `WebType`
is `LIFERAY_AUTHENTICATED`, which is required to load Liferay DXP pages that
require authentication. If you need to set `LIFERAY_AUTHENTICATED` manually,
call `setWebType(WebScreenletConfiguration.WebType.LIFERAY_AUTHENTICATED)`.
```

3. Call the `WebScreenletConfiguration.Builder` instance's `load()` method, which returns a `WebScreenletConfiguration` object.
4. Use Web Screenlet's `setWebScreenletConfiguration` method to set the `WebScreenletConfiguration` object to the Web Screenlet instance.
5. Call the Web Screenlet instance's `load()` method.

Here's an example snippet of these steps in the `onCreate()` method of an activity in which the Web Screenlet instance is `screenlet`, and the `WebScreenletConfiguration` object is `webScreenletConfiguration`:

```
WebScreenletConfiguration webScreenletConfiguration =
            new WebScreenletConfiguration.Builder("/web/westeros-hybrid/companynews")
                .addRawCss(R.raw.portlet, "portlet.css")
                .addLocalCss("gallery.css")
                .addLocalJs("gallery.js")
                .load();

        screenlet.setWebScreenletConfiguration(webScreenletConfiguration);
        screenlet.load();
```

There are a few things to note about this example:

- The relative URL `/web/westeros-hybrid/companynews` supplied to the `WebScreenletConfiguration.Builder` constructor, and the lack of a `setWebType(WebScreenletConfiguration.WebType.OTHER)` call, indicates that this Web Screenlet instance loads a Liferay DXP page that requires authentication.
- The `addRawCss` method adds the CSS file `portlet.css` from the app's `res/raw` folder. Any files that you add via the methods `addRawCss` or `addRawJs` must be located in `res/raw` (create this folder if it doesn't exist). Also note that you must reference these files with `R.raw.yourfilename`. For instance, the `portlet.css` file in this is example is referenced with `R.raw.portlet`.
- The `addLocalCss` and `addLocalJs` methods add the local files `gallery.css` and `gallery.js`, respectively. Any files that you add via these methods must be in the first level of your app's `assets` folder. This folder must exist at the same level as your app's `res` folder. Create the `assets` folder in that location if it doesn't exist.

Great! Now you know how to use Web Screenlet in your Android apps.

Related Topics

Web Screenlet for Android

Using Web Screenlet with Cordova in Your Android App

Using Screenlets in Android Apps

Rendering Web Content in Your Android App

76.16 Using Web Screenlet with Cordova in Your Android App

By using Cordova plugins in Web Screenlet, you can extend the functionality of the web page that the Screenlet renders. This lets you tailor that page to your app's needs.

You'll get started by installing and configuring Cordova. There are two ways to do this: automatically, or manually. The automatic method is covered first.

Installing and Configuring Cordova Automatically

Follow these steps to automatically create an empty Android project configured to use Cordova. Note that you must have git, Node.js, and npm installed.

1. Install `screens-cli`:

   ```
   npm install -g screens-cli
   ```

2. Create the file `.plugins.screens` in the folder you want to create your project in. In this file, add all the Cordova plugins you want to use in your app. For example, you can add plugins from Cordova or GitHub:

   ```
   https://github.com/apache/cordova-plugin-wkwebview-engine.git
   cordova-plugin-call-number
   cordova-plugin-camera
   ```

3. In the folder containing your `.plugins.screens` file, run `screens-cli` to create your project:

   ```
   screens-cli android <project-name>
   ```

   This creates your project in the folder `platforms/android/<project-name>`. You can open it with Android Studio.

Installing and Configuring Cordova Manually

To install and configure Cordova manually, follow these steps:

1. Follow the Cordova getting started guide to install Cordova, create a Cordova project, and add the Android platform to your Cordova project.

2. Install any Cordova plugins you want to use in your app. For example, this command adds the Cordova plugin `cordova-plugin-call-number`:

   ```
   cordova plugin add cordova-plugin-call-number
   ```

   You can use `cordova plugin` to view the currently installed plugins.

3. Copy the following files and folders from your Cordova project to your Android project's root folder:

   - `/platforms/android/res/xml/config.xml`
   - `/platforms/android/assets/www`

   You should also review other files like `AndroidManifest.xml`, resource files, and so on. Some plugins add permissions or styles in such files that you may need to copy for those plugins to work correctly in your Android app.

Using Cordova in Web Screenlet

Now that you've installed and configured Cordova in your Android project, you're ready to use it with Web Screenlet. Follow these steps to do so:

1. Insert and configure Web Screenlet in your app.

2. When you set Web Screenlet's parameters via the `WebScreenletConfiguration.Builder` object, you must enable Cordova by calling the `enableCordova` method with a `CordovaLifeCycleObserver` argument. `CordovaLifeCycleObserver` informs Cordova about the activity lifecycle. You can create an instance of this observer by using its no-argument constructor.

   For example, this code creates a `CordovaLifeCycleObserver` object that it then uses with `enableCordova` when setting Web Screenlet's parameters:

   ```
   CordovaLifeCycleObserver observer = new CordovaLifeCycleObserver();

   WebScreenletConfiguration configuration =
           new WebScreenletConfiguration
                   .Builder("/")
                   .addLocalJs("call.js")
                   .enableCordova(observer)
                   .load();

   webScreenlet.setWebScreenletConfiguration(configuration);
   webScreenlet.load();
   ```

3. Override the following `Activity` methods to call their corresponding observer methods:

   ```
   @Override
   protected void onStart() {
       super.onStart();
       observer.onStart();
   }

   @Override
   protected void onStop() {
       super.onStop();

       observer.onStop();
   }

   @Override
   public void onPause() {
       super.onPause();

       observer.onPause();
   }

   @Override
   public void onResume() {
       super.onResume();

       observer.onResume();
   }

   @Override
   public void onDestroy() {
       super.onDestroy();

       observer.onDestroy();
   }
   ```

```
    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);

        observer.onSaveInstanceState(outState);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);

        observer.onActivityResult(requestCode, resultCode, data);
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
        @NonNull int[] grantResults) {
            super.onRequestPermissionsResult(requestCode, permissions, grantResults);

            observer.onRequestPermissionsResult(requestCode, permissions, grantResults);
    }

    @Override
    public void onConfigurationChanged(Configuration newConfig) {
        super.onConfigurationChanged(newConfig);

        observer.onConfigurationChanged(newConfig);
    }
```

That's it! Note, however, that you may also need to invoke Cordova from a JavaScript file, depending on what you're doing. For example, to use the Cordova plugin `cordova-plugin-call-number` to call a number, you must add a JavaScript file with the following code:

```
function callNumber() {
    //This line triggers the Cordova plugin and makes a call
    window.plugins.CallNumber.callNumber(null, function(){ alert("Calling failed.") }, "900000000", true);
}

setTimeout(callNumber, 3000);
```

If you run the app containing this code and wait three seconds, the plugin activates and calls the number in the JavaScript file.

Great! Now you know how to use Web Screenlet with Cordova.

Related Topics

Rendering Web Pages in Your Android App

Web Screenlet for Android

76.17 Adding Offline Mode Support to Your Android Screenlet

Offline mode lets Screenlets function without a network connection. For offline mode to work with your Screenlet, you must manually add support for it. Fortunately, Liferay Screens 2.0 introduced a simpler way of implementing offline mode support in Android Screenlets:

- Update your Screenlet's classes to leverage the offline mode cache
- Create an event class (if your Screenlet doesn't already have one)

Implementing these steps, however, differs somewhat depending on how your Screenlet communicates with the server:

- **A write Screenlet:** writes data to a server. The Add Bookmark Screenlet created in the basic Screenlet creation tutorial is a good example of a simple write Screenlet. It asks the user to enter a URL and a title, which it then sends to the Bookmarks portlet in Liferay DXP to create a bookmark.
- **A read Screenlet:** reads data from a server. The Web Content Display Screenlet included with Liferay Screens is a good example of a read Screenlet. It retrieves web content from Liferay DXP for display in an Android app. Click here to see Web Content Display Screenlet's documentation.

This tutorial shows you how to add offline mode support to both kinds of Screenlets. You'll start with write Screenlets, using Add Bookmark Screenlet as an example. Before getting started, be sure to read the basic Screenlet creation tutorial to familiarize yourself with Add Bookmark Screenlet's code. You'll conclude by learning how offline mode implementation in read Screenlets differs from that of write Screenlets.

Adding Offline Mode Support to Write Screenlets

To add offline mode support to write Screenlets, you'll follow these steps:

1. Create or update the event class.
2. Update the listener interface.
3. Update the Interactor class.
4. Update the Screenlet class.
5. Sync the cache with the server.

Each of the sections that follow detail one of these steps. You'll begin by creating or updating the event class.

Create or Update the Event Class

Recall from the basic Screenlet creation tutorial that an event class is required to handle communication between Screenlet components. Also recall that many Screenlets can use the event class included with Screens, `BasicEvent`, as their event class. For offline mode to work, however, you must create an event class that extends `CacheEvent` (click here to see `CacheEvent`). Your event class has one primary responsibility: store and provide access to the arguments passed to the Interactor. To accomplish this, your event class should do these things:

- Extend `CacheEvent`. For the arguments, define variables and public getter methods.
- Define a no-argument constructor that only calls the corresponding superclass constructor.
- Define a constructor that sets the Interactor's arguments.

In the case of Add Bookmark Screenlet, the arguments are the bookmark's URL, folder ID, and title. For example, here's the full code for this Screenlet's event class, `BookmarkEvent`:

```
public class BookmarkEvent extends CacheEvent {

    private String url;
    private String title;
    private long folderId;

    public BookmarkEvent() {
        super();
    }

    public BookmarkEvent(String url, String title, long folderId) {

        this.url = url;
        this.title = title;
        this.folderId = folderId;
    }

    public String getURL() {
        return url;
    }

    public String getTitle() {
        return title;
    }

    public long getFolderId() {
        return folderId;
    }
}
```

Next, you'll update the listener.

Update the Listener

Recall from the basic Screenlet creation tutorial that the listener interface defines a success method and a failure method. This lets implementing classes respond to the server call's success or failure. Listeners that support offline mode offer the same functionality, although differently. Offline mode listeners must extend `BaseCacheListener`, which defines only this `error` method:

```
void error(Exception e, String userAction);
```

By extending `BaseCacheListener`, your listener no longer needs an explicit failure method because it inherits the `error` method instead. This `error` method also includes an argument for the user action that triggered the exception.

You can therefore update your listener to support offline mode by extending `BaseCacheListener` and deleting the failure method. For example, here's Add Bookmark Screenlet's listener, `AddBookmarkListener`, after being updated to support offline mode:

```
public interface AddBookmarkListener extends BaseCacheListener {

    onAddBookmarkSuccess();
}
```

Note that you must also remove any failure method implementations (such as in an activity or fragment that implements the listener), and replace any failure method calls with `error` method calls. You'll do the latter next when updating the Interactor class.

Update the Interactor Class

Recall from the basic Screenlet creation tutorial that Interactor classes extend `BaseRemoteInteractor` with the listener and event as type arguments. To support offline mode, your Interactor class must instead extend one of the following classes. Which one depends on whether your Interactor writes data to or reads data from a server:

- `BaseCacheWriteInteractor`: writes data to a server. Extend this class if your Screenlet is a write Screenlet. Click here to see this class.

- BaseCacheReadInteractor: reads data from a server. Extend this class if your Screenlet is a read Screenlet. Click here to see this class.

In either case, the type arguments are the same: the listener and the event. Note, however, that the event must extend CacheEvent as described above. For example, since Add Bookmark Screenlet is a write Screenlet, to support offline mode its Interactor class must extend BaseCacheWriteInteractor with AddBookmarkListener and AddBookmarkEvent as type arguments:

```
public class AddBookmarkInteractor extends
    BaseCacheWriteInteractor<AddBookmarkListener, BookmarkEvent> {...
```

You must also make a few changes to the Interactor class's code. The main change is that the execute method now takes the event instead of var args. You can then retrieve the data you need from the event. For example, to support offline mode, the execute method in AddBookmarkInteractor takes BookmarkEvent as an argument. The bookmark's URL, title, and folder ID are then retrieved from the event for use in the getJSONObject method that makes the server call. The execute method finishes by setting the resulting JSONObject to the event, and then returning the event:

```
@Override
public BookmarkEvent execute(BookmarkEvent bookmarkEvent) throws Exception {

    validate(bookmarkEvent.getUrl(), bookmarkEvent.getFolderId());

    JSONObject jsonObject = getJSONObject(bookmarkEvent.getUrl(), bookmarkEvent.getTitle(),
        bookmarkEvent.getFolderId());
    bookmarkEvent.setJSONObject(jsonObject);
    return bookmarkEvent;
}
```

You should also change the onSuccess method to take an instance of your event class instead of BasicEvent. This is the only change you need to make to this method. For example, the onSuccess method in AddBookmarkInteractor supports offline mode by taking a BookmarkEvent instead of a BasicEvent:

```
@Override
public void onSuccess(BookmarkEvent event) {
    getListener().onAddBookmarkSuccess();
}
```

Now make the same change to the onFailure method, but replace the listener's failure method call with a call to the error method inherited from BaseCacheListener (see the listener section above for an explanation of this method). For the error method's arguments, you can retrieve the exception from the event and define a string to use as the user action. For example, to support offline mode the onFailure method in AddBookmarkInteractor takes a BookmarkEvent instead of a BasicEvent. Also, the method's error call defines the "ADD_BOOKMARK" string to indicate that the error occurred while trying to add a bookmark to the server:

```
@Override public void onFailure(BookmarkEvent event) {
    getListener().error(event.getException(), "ADD_BOOKMARK");
}
```

Update the Screenlet Class

Updating the Screenlet class for offline mode is straightforward. In the Screenlet class's `onUserAction` method, you'll change the call to the Interactor's `start` method so that it takes only an event as an argument. Before doing this, however, you should create an event instance and set its cache key. A cache key is a value that identifies an entity in the local cache. This lets you retrieve the entity from the cache for later synchronization with the server.

In Add Bookmark Screenlet, for example, a bookmark's URL makes a good cache key. To support offline mode, the `onUserAction` method in `AddBookmarkScreenlet` creates a new `BookmarkEvent` instance with a bookmark's data and then uses the `setCacheKey` method to set the bookmark's URL as the event's cache key. The Interactor's start method takes this event as its argument:

```
BookmarkEvent event = new BookmarkEvent(url, title, folderId);
event.setCacheKey(url);
interactor.start(event);
```

Note that you don't have to set a cache key to use offline mode. Instead, you can pass the event to the `start` method without calling `setCacheKey`. However, this means that you'll only be able to access the most recent entity in the cache.

That's it! Your write Screenlet now supports offline mode. There's one more detail to keep in mind, however, when using the Screenlet: syncing the cache with the server. You'll learn about this next.

Sync the Cache with the Server

When using a write Screenlet that supports offline mode, new data written to the cache must also be synced with the server. The write Screenlets included with Liferay Screens do this for you. However, you must do this manually when using a custom write Screenlet. You should do this in the activity or fragment that uses the Screenlet–exactly where in this activity or fragment is up to you though.

To sync a write Screenlet's data with the server manually, follow these steps:

1. Retrieve the event that needs to be synced with the server. To do this, you must first get the cache key associated with the event. Then use the key as an argument to the `Cache.getObject` method.
2. Call the Interactor with the event. This syncs the data with the server.

For example, the following code uses the `Cache.findKeys` method to retrieve all `BookmarkEvent` keys in the cache. The loop that follows then retrieves the event that corresponds to each key, and syncs it to the server by calling the Interactor:

```
String[] keys = Cache.findKeys(BookmarkEvent.class, groupId, userId, locale, 0,
    Integer.MAX_VALUE);
for (String key : keys) {

    BookmarkEvent event = Cache.getObject(BookmarkEvent.class, groupId, userId, key);
    new AddBookmarkInteractor().execute(event);
}
```

Note that if you opted not to set a cache key in your Screenlet class, you can pass `null` in place of a key.

Also note that you can use Android's `SharedPreferences` APIs as an alternative way to store and retrieve cache keys. For example, the following code stores cache keys in shared preferences:

```
SharedPreferences sharedPreferences = getSharedPreferences("MY_PREFERENCES", Context.MODE_PRIVATE);
HashSet<String> values = new HashSet<>();
sharedPreferences.edit().putStringSet("keysToSync", values).apply();
```

You can then retrieve the keys as you would retrieve any other key-value set from shared preferences:

```
SharedPreferences sharedPreferences = getSharedPreferences("MY_PREFERENCES", Context.MODE_PRIVATE);
HashSet<String> keysToSync = sharedPreferences.getStringSet("keysToSync", new HashSet<>());
```

Next, you'll learn how to add offline mode support to read Screenlets.

Adding Offline Mode Support to Read Screenlets

Implementing offline mode support in a read Screenlet is almost identical to doing so in a write Screenlet. There are two small differences, though:

1. You can still pass arguments to the Interactor with var args instead of an event.

2. The Interactor class must extend `BaseCacheReadInteractor`, which forces you to implement the `getIdFromArgs` method. This method takes the var args passed to the Interactor so you can return the argument that identifies your entity. Note that because this method requires you to return a `String`, you'll often use `String.valueOf` to return non-string arguments as a string. For example, the `getIdFromArgs` implementation in Comment Display Screenlet's `CommentLoadInteractor` retrieves the comment ID (a `long`) from the first argument and then returns it as a `String`:

```
    @Override
    protected String getIdFromArgs(Object... args) {
        long commentId = (long) args[0];
        return String.valueOf(commentId);
    }
```

That's it! Next, you'll learn about list Screenlets and offline mode support.

Adding Offline Mode Support to List Screenlets

A list Screenlet is a special type of read Screenlet that displays entities in a list. Recall from the list Screenlet creation tutorial that list Screenlets have a model class that encapsulates entities retrieved from the server. To support offline mode, a list Screenlet's event class must extend `ListEvent` with the model class as a type argument. This event class also needs three things:

1. A default constructor
2. A `getListKey` method that returns a unique ID to store the entity with
3. A `getModel` method that returns the model instance

The list Screenlet creation tutorial contains example model and event classes that support offline mode for Bookmark List Screenlet. Click the following links to see the sections in the tutorial that show you how to create these classes:

- Creating the Model Class
- Creating the Event

And that's all! Now you know how to support offline mode in your Screenlets.

Related Topics

Using Offline Mode in Android
Architecture of Offline Mode in Liferay Screens
Creating Android Screenlets
Creating Android List Screenlets

76.18 Android Best Practices

When developing Android projects with Liferay Screens, there are a few best practices that you should follow to ensure your code is as clean and bug-free as possible. This tutorial lists these.

Update Your Tools

You should first make sure that you have the latest tools installed. You should use the latest Android API level with the latest version of Android Studio. Although Screens *may* work with Eclipse ADT or manual Gradle builds, Android Studio is the preferred IDE.

See the Breaking Changes Document

When updating an app or Screenlet to a new version of Liferay Screens, make sure to see the Android breaking changes reference article. This article lists changes to Screens that break functionality in prior versions. In most cases, updating your code is relatively straightforward.

Naming Conventions

Using the naming conventions described here leads to consistency and a better understanding of the Screens library. This makes working with your Screenlets much simpler.

Also note that Liferay Screens follows Square's Java conventions for Android, with tabs as separator. The configuration for IDEA, findbugs, PMD, and checkstyle is available in the project's source code.

Screenlet Folder

Your Screenlet folder's name should indicate your Screenlet's functionality. For example, Login Screenlet's folder is named `login`.

If you have multiple Screenlets that operate on the same entity, you can place them inside a folder named for that entity. For example, Asset Display Screenlet and Asset List Screenlet both work with Liferay assets. They're therefore in the Screens library's `asset` folder.

Screenlets

Naming Screenlets properly is very important; they're the main focus of Liferay Screens. You should name your Screenlet with its principal action first, followed by *Screenlet*. Its Screenlet class should also follow this pattern. For example, Login Screenlet's principal action is to log users into a Liferay DXP installation. This Screenlet's Screenlet class is therefore `LoginScreenlet`.

View Models

Name your View models the same way you name Screenlets, but substitute `ViewModel` for `Screenlet`. Also, place your View Models in a `view` folder in your Screenlet's root folder. For example, Login Screenlet's View Model is named `LoginViewModel` and is in the `login/view` folder.

Interactors

Place your Screenlet's Interactors in a folder named `interactor` in your Screenlet's root folder. Name each Interactor first with the object it operates on, followed by its action and the suffix *Interactor*. If you wish, you can also put each Interactor in its own folder named after its action. For example, Rating Screenlet has three Interactors. Each is in its own folder inside the `interactor` folder:

- `delete/RatingDeleteInteractor`: Deletes an asset's ratings
- `load/RatingLoadInteractor`: Loads an asset's ratings
- `update/RatingUpdateInteractor`: Updates an asset's ratings

Views

Place Views in a `view` folder in the Screenlet's root folder. If you're creating a View Set, however, you can place its Views in a separate `viewsets` folder outside your Screenlets' root folders. This is what the Screens Library does for its Material and Westeros View Sets. The `material` and `westeros` folders contain those View Sets, respectively. Also note that in each View, each Screenlet's View class is in its own folder. For example, the View class for Forgot Password Screenlet's Material View is in the folder `viewsets/material/src/main/java/com/liferay/mobile/screens/viewsets/material/auth/forgotpassword`. Note that the `auth` folder in this path is the Screenlet's module. Creating your Screenlets and Views in modules isn't required. Also note that the View's layout file `forgotpassword_material.xml` is in `viewsets/material/src/main/res/layout`.

Name a View's layout XML and View class after your Screenlet, substituting *View* for *Screenlet* where necessary. The layout's filename should also be suffixed with `_yourViewName`. For example, the XIB file and View class for Forgot Password Screenlet's Material View are `forgotpassword_material.xml` and `ForgotPasswordView.java`, respectively.

Avoid Hard Coded Elements

Using constants instead of hard-coded elements is a simple way to avoid bugs. Constants reduce the likelihood that you'll make a typo when referring to common elements. They also gather these elements in a single location. For example, DDL Form Screenlet's Screenlet class defines the following constants for the user action names:

```
public static final String LOAD_FORM_ACTION = "loadForm";
public static final String LOAD_RECORD_ACTION = "loadRecord";
public static final String ADD_RECORD_ACTION = "addRecord";
public static final String UPDATE_RECORD_ACTION = "updateRecord";
public static final String UPLOAD_DOCUMENT_ACTION = "uploadDocument";
```

Avoid State in Interactors

Liferay Screens uses EventBus to ensure that the network or background operation isn't lost when the device changes orientation. For this to work, however, you must ensure that your Interactor's request is stateless.

If an Interactor needs some piece of information, you should pass it to the Interactor via the `start` call and then attach it to the event. You can see an example of this in the sample Add Bookmark Screenlet from the Screenlet creation tutorial. The onUserAction method in the Screenlet class (AddBookmarkScreenlet) passes a Bookmark's URL and title from the View Model to the Interactor via the Interactor's start method:

```
@Override
protected void onUserAction(String userActionName, AddBookmarkInteractor interactor,
    Object... args) {
        AddBookmarkViewModel viewModel = getViewModel();
        String url = viewModel.getURL();
        String title = viewModel.getTitle();

        interactor.start(url, title, folderId);
}
```

The start method calls the Interactor's execute method in a background thread. The execute method in Add Bookmark Screenlet's Interactor (AddBookmarkInteractor) creates a BasicEvent object that contains the start method's arguments:

```
@Override
public BasicEvent execute(Object[] args) throws Exception {
    String url = (String) args[0];
    String title = (String) args[1];
    long folderId = (long) args[2];

    validate(url, folderId);

    JSONObject jsonObject = getJSONObject(url, title, folderId);
    return new BasicEvent(jsonObject);
}
```

Stay in Your Layer

When accessing variables that belong to other Screenlet components, you should avoid those outside your current Screenlet layer. This achieves better decoupling between the layers, which tends to reduce bugs and simplify maintenance. For an explanation of the layers in Liferay Screens, see the architecture tutorial. For example, don't directly access View variables from an Interactor. Instead, pass data from a View Model to the Interactor via the Interactor's start method. The example onUserAction method in the previous section illustrates this.

Related Topics

Liferay Screens for Android Troubleshooting and FAQs
    Architecture of Liferay Screens for Android
    Creating Android Screenlets
    Android Breaking Changes

76.19 Liferay Screens for Android Troubleshooting and FAQs

Even though Liferay developed Screens for Android with great care, you may still run into some common issues. Here are solutions and tips for solving these issues. You'll also find answers to common questions about Screens for Android.

General Troubleshooting

Before delving into specific issues, you should first make sure that you have the latest tools installed and know where to get additional help if you need it. You should use the latest Android API level, with the latest version of Android Studio. Although Screens *can* work with Eclipse ADT or manual Gradle builds, Android Studio is the preferred IDE.

If you're having trouble using Liferay Screens, it may help to investigate the sample apps developed by Liferay. Both serve as good examples of how to use Screenlets and Views:

- Westeros Bank
- Test App

When updating an app or Screenlet to a new version of Liferay Screens, make sure to see the Android breaking changes reference article. This article lists changes to Screens that break functionality in prior versions. In most cases, updating your code is relatively straightforward.

If you get stuck at any point, you can post your question on our forum. We're happy to assist you! If you found a bug or want to suggest an improvement, file a ticket in our Jira. Note that you must log in first to be able to see the project.

Common Issues

This section contains information on common issues that can occur when using Liferay Screens.

1. Could not find `com.liferay.mobile:liferay-screens`

   This error occurs when Gradle isn't able to find Liferay Screens or the repository. First, check that the Screens version number you're trying to use exists in jCenter. You can use this link to see all uploaded versions.

   It's also possible that you're using an old Gradle plugin that doesn't use jCenter as the default repository. Screens uses version 1.2.3 and later. You can add jCenter as a new repository by placing this code in your project's `build.gradle` file:

   ```
   repositories {
       jcenter()
   }
   ```

2. Failed to resolve: `com.android.support:appcompat-v7`

   Liferay Screens uses the appcompat library from Google, as do all new Android projects created with the latest Android Studio. The appcompat library uses a custom repository maintained by Google, so it must be updated manually using the Android SDK Manager.

   In the Android SDK Manager (located at *Tools → Android → SDK Manager* in Android Studio), you must install at least version 14 of the Android Support Repository (in the *Extras* menu), and version 22.1.1 of the Android Support Library.

3. Duplicate files copied in `APK META-INF…`

   This is a common Android error when using libraries. It occurs because Gradle can't merge duplicated files such as license or notice files. To prevent this error, add the following code to your `build.gradle` file:

```
android {
    ...
    packagingOptions {
        exclude 'META-INF/LICENSE'
        exclude 'META-INF/NOTICE'
    }
    ...
}
```

This error may not happen right away, but may only appear later on in the development process. For this reason, it's recommended that you put the above code in your `build.gradle` file after creating your project.

4. Connect failed: ECONNREFUSED (Connection refused), or `org.apache.http.conn.HttpHostConnectException`

   This error occurs when Liferay Screens and the underlying Liferay Mobile SDK can't connect to the Liferay Portal instance. If you get this error, you should first check the IP address of the server to make sure it's available. If you've overridden the default IP address in `server_context.xml`, you should check to make sure that you've set it to the correct IP. Also, if you're using the Genymotion emulator, you must use `192.168.56.1` instead of localhost for your app to communicate with a local Liferay instance.

5. `java.io.IOException`: open failed: EACCES (Permission denied)

   Some Screenlets use temporary files to store information, such as when the User Portrait Screenlet uploads a new portrait, or the DDL Form Screenlet uploads new files to the portal. Your app needs to have the necessary permissions to use a specific Screenlet's functionality. In this case, add the following line to your `AndroidManifest.xml`:

   ```
   <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
   ```

   If you're using the device's camera, you also need to add the following permission:

   ```
   <uses-permission android:name="android.permission.CAMERA"/>
   ```

6. No JSON web service action with path …

   This error most commonly occurs if you haven't installed the Liferay Screens Compatibility Plugin. This plugin adds new API calls to Liferay Portal.

FAQs

1. Do I have to use Android Studio?

   No, Liferay Screens also works with Eclipse ADT. You can also compile your project manually with Gradle or another build system. Just make sure you use the compiled aar in your project's `lib` folder.

   We *strongly* recommend, however, that you use Android Studio. Android Studio is the official IDE for developing Android apps, and Eclipse ADT is no longer supported. Using Eclipse ADT or compiling manually may cause unexpected issues that are difficult to overcome.

2. How does Screens handle orientation changes?

   Liferay Screens uses an event bus, the `EventBus` library, to notify activities when the interactor has finished its work.

3. How can I use a Liferay feature not available in Screens?

   There are several ways you can use Liferay features not currently available in Screens. The Liferay Mobile SDK gives you access to all of Liferay's remote APIs. You can also create a custom Screenlet to support any features not included in Screens by default.

4. How do I create a new Screenlet?

   Screenlet creation is explained in detail here.

5. How can I customize a Screenlet?

   You can customize Screenlets by creating new Views. Fortunately, there's a tutorial for this!

6. Does Screens have offline support?

   Yes, since Liferay Screens 1.3!

Related Topics

Preparing Android Projects for Liferay Screens
    Creating Android Screenlets
    Creating Android Views
    Mobile SDK
    Android Breaking Changes

Chapter 77

iOS Apps with Liferay Screens

Liferay Screens speeds up and simplifies developing native mobile apps that use Liferay. Its power lies in its *Screenlets*. A Screenlet is a visual component that you insert into your native app to leverage Liferay Portal's content and services. On iOS, Screenlets are available to log in to your portal, create accounts, submit forms, display content, and more. You can use any number of Screenlets in your app; they're independent, so you can use them in modular fashion. Screenlets on iOS also deliver UI flexibility with pluggable *Themes* that implement elegant user interfaces. Liferay's reference documentation for iOS Screenlets describes each Screenlet's features and Themes.

You might be thinking, "These Screenlets sound like the greatest thing since taco Tuesdays, but what if they don't fit in with my app's UI? What if they don't behave exactly how I want them to? What if there's no Screenlet for what I want to do?" Fret not! You can customize Screenlets to fit your needs by changing or extending their UI and behavior. You can even write your own Screenlets! What's more, Screens seamlessly integrates with your existing iOS projects.

Screenlets leverage the Liferay Mobile SDK to make server calls. The Mobile SDK is a low-level layer on top of the Liferay JSON API. To write your own Screenlets, you must familiarize yourself with Liferay's remote services. If no existing Screenlet meets your needs, consider customizing an existing Screenlet, creating a Screenlet, or directly using the Mobile SDK. Creating a Screenlet involves writing Mobile SDK calls and constructing the Screenlet; if you don't plan to reuse or distribute the implementation then you may want to forgo writing a Screenlet and, instead, work with the Mobile SDK. A benefit of integrating an existing Screenlet into your app, however, is that the Mobile SDK's details are abstracted from you.

These tutorials show you how to use, customize, create, and distribute Screenlets for iOS. They show you how to create Themes too. There's even a tutorial that explains the nitty-gritty details of the Liferay Screens architecture. No matter how deep you want to go, you'll use Screenlets in no time. Start by preparing your iOS project to use Liferay Screens.

77.1 Preparing iOS Projects for Liferay Screens

To develop iOS apps with Liferay Screens, you must first install and configure Screens in your iOS project. Screens is released as a standard CocoaPods dependency. You must therefore install Screens via CocoaPods. After completing the installation, you must configure your iOS project to communicate with your Liferay DXP instance. This tutorial walks you through these processes. You'll be up and running in no time!

First, you'll review the requirements for Liferay Screens.

Sign Up
Lorem ipsum dolor sit amet, consectetur adipisicing elit. Donec auctor neque sed pretium luctus.
First Name
Last Name
Email Address
Password
Sign Up

Figure 77.1: Here's an app that uses a Liferay Screens Sign Up Screenlet.

Requirements

Liferay Screens for iOS includes the Component Library (the Screenlets) and some sample projects written in Swift. Screens is developed using Swift and development techniques that leverage functional Swift code and the Model View Presenter architecture. You can use Swift or Objective-C with Screens, and you can run Screens apps on iOS 9 and above.

Liferay Screens for iOS requires the following software:

- Xcode 9.3 or newer
- iOS 11 SDK
- CocoaPods 1 or newer
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, or Liferay DXP 7.0
- Liferay Screens Compatibility Plugin (CE or DXP/EE, depending on your portal edition). This app is preinstalled in Liferay CE Portal 7.0/7.1 CE and Liferay DXP 7.0.

Securing JSON Web Services

Each Screenlet in Liferay Screens calls one or more of Liferay DXP's JSON web services, which are enabled by default. The Screenlet reference documentation lists the web services that each Screenlet calls. To use a

Screenlet, its web services must be enabled in the portal. It's possible, however, to disable the web services needed by Screenlets you're not using. For instructions on this, see the tutorial Configuring JSON Web Services. You can also use Service Access Policies for more fine-grained control over accessible services.

Configuring Your Project with CocoaPods

To use CocoaPods to prepare your iOS 9.0 (or above) project for Liferay Screens, follow these steps:

1. In your project's root folder, add the following code to the file named `Podfile`, or create this file if it doesn't exist. Be sure to replace `Your  Target` with your target's name:

   ```
   source 'https://github.com/CocoaPods/Specs.git'

   platform :ios, '9.0'
   use_frameworks!

   target "Your Target" do
       pod 'LiferayScreens'
   end

   # the rest of your podfile
   ```

   Note that Liferay Screens and some of its dependencies aren't compatible with Swift 3.2 or Swift 4.0. If your iOS project is compiled in Swift 3.2 or Swift 4.0, then your `Podfile` must specify Screens and those dependencies for compilation in Swift 4.2. The `post_install` code in the following `Podfile` does this. You must therefore use this `Podfile` if you want to use Screens in a Swift 3.2 or Swift 4.0 app:

   ```
   source 'https://github.com/CocoaPods/Specs.git'

   platform :ios, '9.0'
   use_frameworks!

   target "Your Target" do
       pod 'LiferayScreens'
   end

   post_install do |installer|
     incompatiblePods = [
       'Cosmos',
       'CryptoSwift',
       'KeychainAccess',
       'Liferay-iOS-SDK',
       'Liferay-OAuth',
       'LiferayScreens',
       'Kingfisher'
     ]

     installer.pods_project.targets.each do |target|
       if incompatiblePods.include? target.name
         target.build_configurations.each do |config|
           config.build_settings['SWIFT_VERSION'] = '4.2'
         end
       end
       target.build_configurations.each do |config|
           config.build_settings['CONFIGURATION_BUILD_DIR'] = '$PODS_CONFIGURATION_BUILD_DIR'
       end
     end
   end

   # the rest of your podfile
   ```

2. On the terminal, navigate to your project's root folder and run this command:

   ```
   pod repo update
   ```

   This ensures you have the latest version of the CocoaPods repository on your machine. Note that this command can take a while to run.

3. Still in your project's root folder in the terminal, run this command:

   ```
   pod install
   ```

   Once this completes, quit Xcode and reopen your project by using the `*.xcworkspace` file in your project's root folder. From now on, you must use this file to open your project.

Great! To configure your project's communication with Liferay DXP, you can skip the next section and follow the instructions in the final section.

Configuring Communication with Liferay DXP

Configuring communication between Screenlets and Liferay DXP is easy. Liferay Screens uses a property list (`.plist`) file to access your Liferay DXP instance. It must include the server's URL, the portal's company ID, and the site's group ID. Create a `liferay-server-context.plist` file and specify values required for communicating with your Liferay DXP instance. As an example, refer to `liferay-server-context-sample.plist`.

The values you need to specify in your `liferay-server-context.plist` are:

- `server`: Your Liferay DXP instance's URL.
- `version`: Your Liferay DXP instance's version. Supported values are `71` for Liferay CE Portal 7.1 or Liferay DXP 7.1, `70` for Liferay CE Portal 7.0 or Liferay DXP 7.0, and `62` for Liferay Portal 6.2 CE/EE.
- `companyId`: Your Liferay DXP instance's identifier. You can find this value in the *Instance ID* column of *Control Panel* → *Portal Instances*.
- `groupId`: The ID of the default site you want Screens to communicate with. You can find this value in the Site ID field of the site's *Site Administration* → *Configuration* → *Site Settings* menu.
- `connectorFactoryClassName`: Your Connector's factory class name. This is optional. If you don't include it, the `version` value is used to determine which factory is the most suitable for that version of Liferay DXP.

Great! Your iOS project is ready for Liferay Screens.

Related Topics

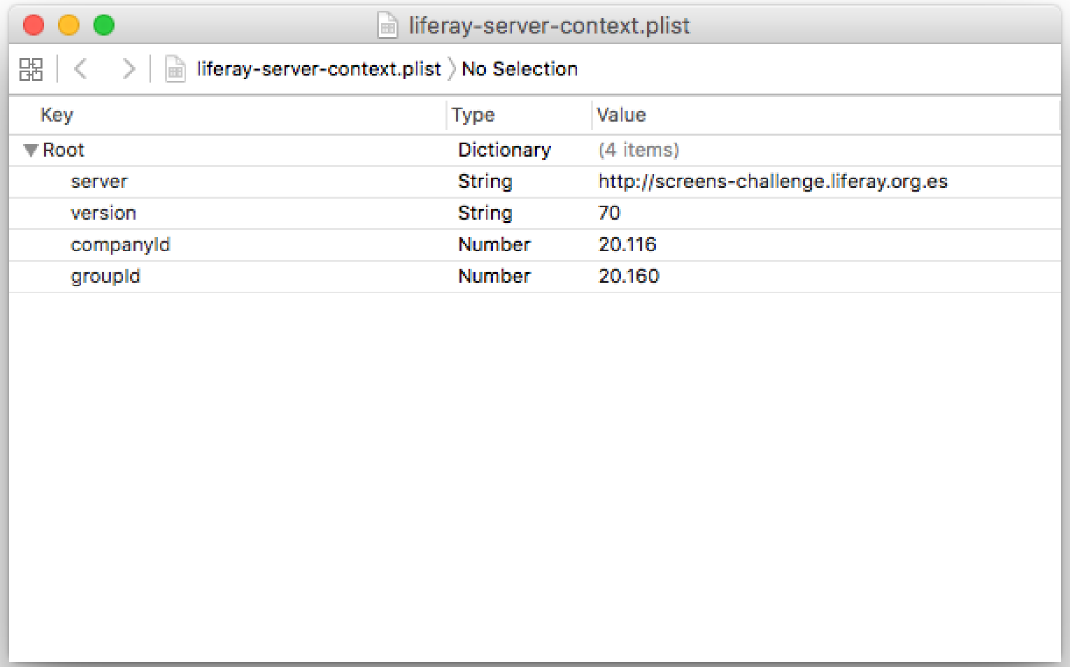
Using Screenlets in iOS Apps

Using Themes in iOS Screenlets

Preparing Android Projects for Liferay Screens

77.2 Using Screenlets in iOS Apps

Once you've prepared your iOS project to use Liferay Screens, you can use Screenlets in your app. There are plenty of Liferay Screenlets available, and they're described in the Screenlet reference documentation. This tutorial shows you how to insert and configure Screenlets in iOS apps written in Swift and Objective-C. It also explains how to localize them. You'll be a Screenlet master in no time!


liferay-server-context.plist
liferay-server-context.plist 〉 No Selection

Key	Type	Value
▼ Root	Dictionary	(4 items)
server	String	http://screens-challenge.liferay.org.es
version	String	70
companyId	Number	20.116
groupId	Number	20.160

Figure 77.2: Here's a property list file, called `liferay-context.plist`.

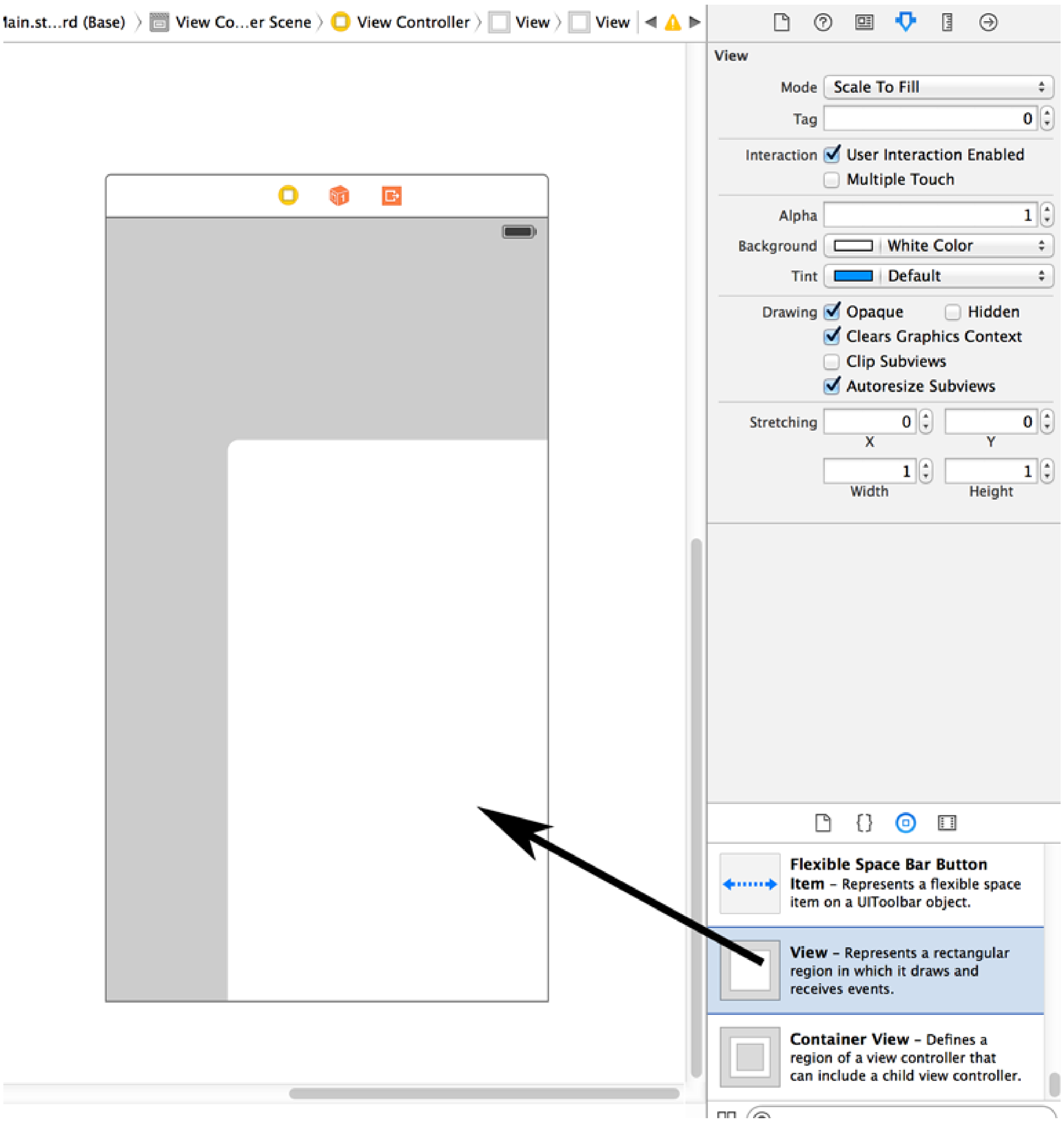
Inserting and Configuring Screenlets in iOS Apps

The first step to using Screenlets in your iOS project is to add a new UIView to your project. In Interface Builder, insert a new UIView into your Storyboard or XIB file. Figure 1 shows this.

Next, enter the Screenlet's name as the Custom Class. For example, if you're using the Login Screenlet, then enter Login Screenlet as the class.

Now you need to conform the Screenlet's delegate protocol in your `ViewController` class. For example, the Login Screenlet's delegate class is `LoginScreenletDelegate`. This is shown in the code that follows. Note that you need to implement the functionality of `onLoginResponse` and `onLoginError`. This is indicated by the comments in the code here:

```
class ViewController: UIViewController, LoginScreenletDelegate {

    ...

   func screenlet(screenlet: BaseScreenlet,
           onLoginResponseUserAttributes attributes: [String:AnyObject]) {
       // handle succeeded login using passed user attributes
    }

    func screenlet(screenlet: BaseScreenlet,
            onLoginError error: NSError) {
        // handle failed login using passed error
    }
```



View

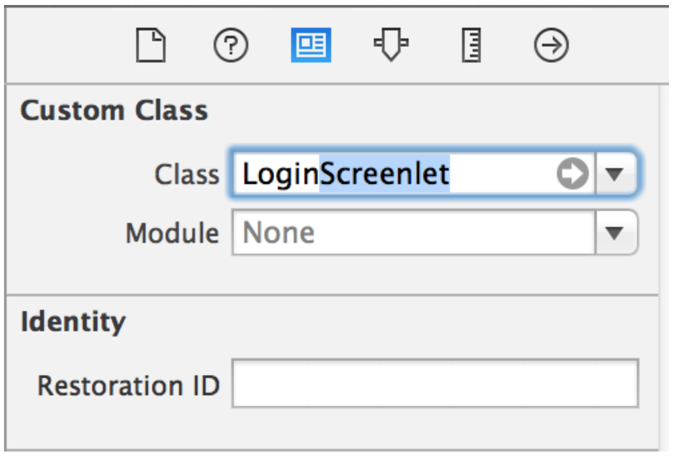
Mode: Scale To Fill
Tag: 0
Interaction: ✓ User Interaction Enabled
☐ Multiple Touch
Alpha: 1
Background: White Color
Tint: Default
Drawing: ✓ Opaque ☐ Hidden
✓ Clears Graphics Context
☐ Clip Subviews
✓ Autoresize Subviews
Stretching: X 0, Y 0, Width 1, Height 1

Flexible Space Bar Button Item – Represents a flexible space item on a UIToolbar object.

View – Represents a rectangular region in which it draws and receives events.

Container View – Defines a region of a view controller that can include a child view controller.

Figure 77.3: Add a new UIView to your project.


Custom Class

Class LoginScreenlet

Module None

Identity

Restoration ID

Figure 77.4: Change the Custom Class to match the Screenlet.

```
    ...
```

If you're using CocoaPods, you need to import Liferay Screens in your View Controller:

```
import LiferayScreens
```

Now that the Screenlet's delegate protocol conforms in your `ViewController` class, go back to Interface Builder and connect the Screenlet's delegate to your View Controller. If the Screenlet you're using has more outlets, you can assign them as well.

Note that currently Xcode has some issues connecting outlets to Swift source code. To get around this, you can change the delegate data type or assign the outlets in your code. In your View Controller, follow these steps:

1. Declare an outlet to hold a reference to the Screenlet. You can connect it in Interface Builder without any issues.

2. Assign the Screenlet's delegate the `viewDidLoad` method. This is the connection typically done in Interface Builder.

These steps are shown in the following code for Login Screenlet's View Controller.

```
class ViewController: UIViewController, LoginScreenletDelegate {

    @IBOutlet var screenlet: LoginScreenlet?

    override func viewDidLoad() {
        super.viewDidLoad()
        self.screenlet?.delegate = self
    }

    ...
```

Awesome! Now you know how to use Screenlets in your apps. If you need to use Screenlets from Objective-C code, follow the instructions in the next section.

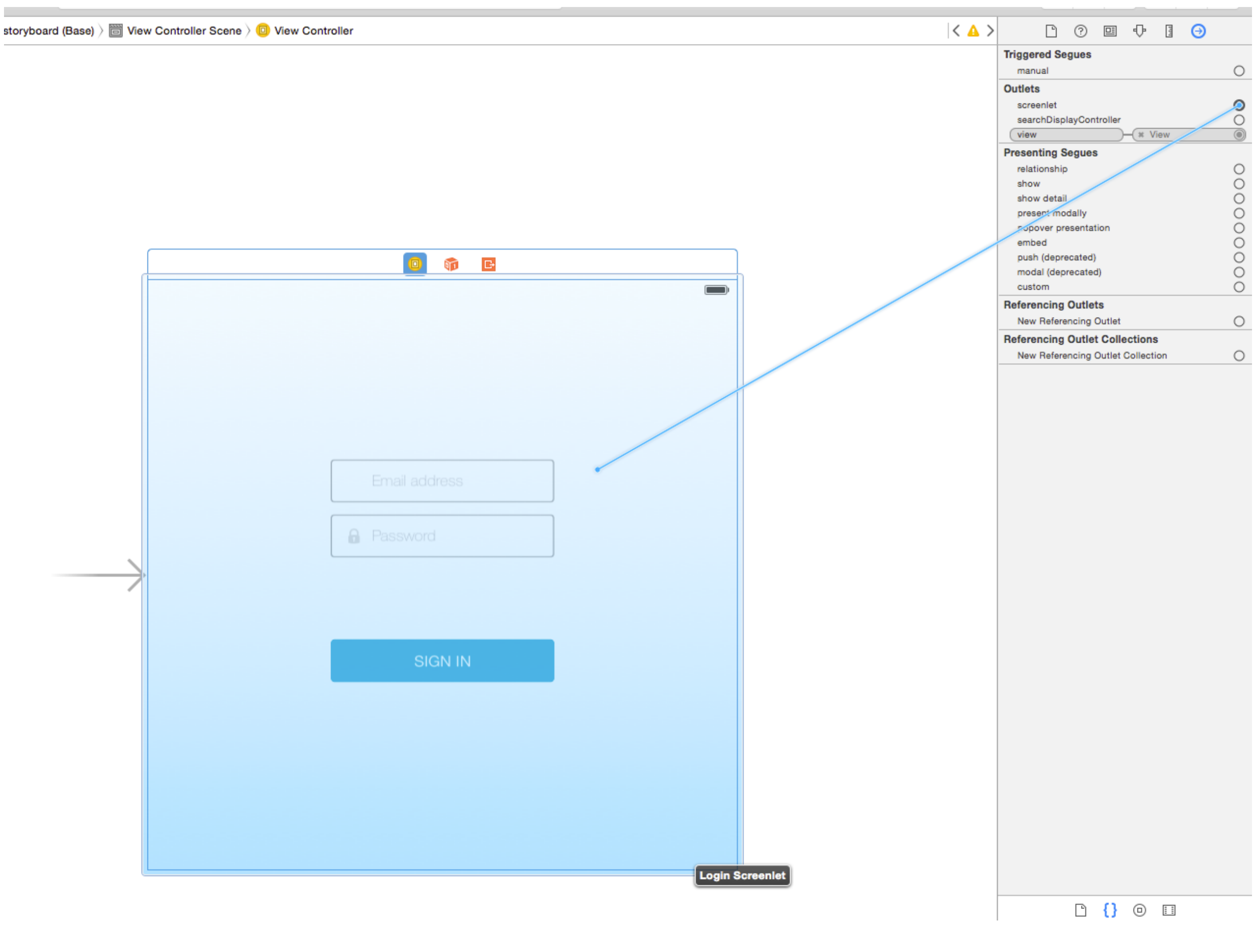


Figure 77.5: Connect the outlet with the Screenlet reference.

Using Screenlets from Objective-C

If you want to invoke Screenlet classes from Objective-C code, there is an additional header file that you must import. You can import the header file `LiferayScreens-Swift.h` in all your Objective-C files or configure a precompiler header file.

The first option involves adding the following import line all of your Objective-C files:

```
#import "LiferayScreens-Swift.h"
```

Alternatively, you can configure a precompiler header file by following these steps:

1. Create a precompiler header file (e.g., `PrefixHeader.pch`) and add it to your project.

2. Import `LiferayScreens-Swift.h` in the precompiler header file you just created.

3. Edit the following build settings of your target. Remember to replace `path/to/your/file/` with the path to your `PrefixHeader.pch` file:

    - Precompile Prefix Header: `Yes`
    - Prefix Header: `path/to/your/file/PrefixHeader.pch`

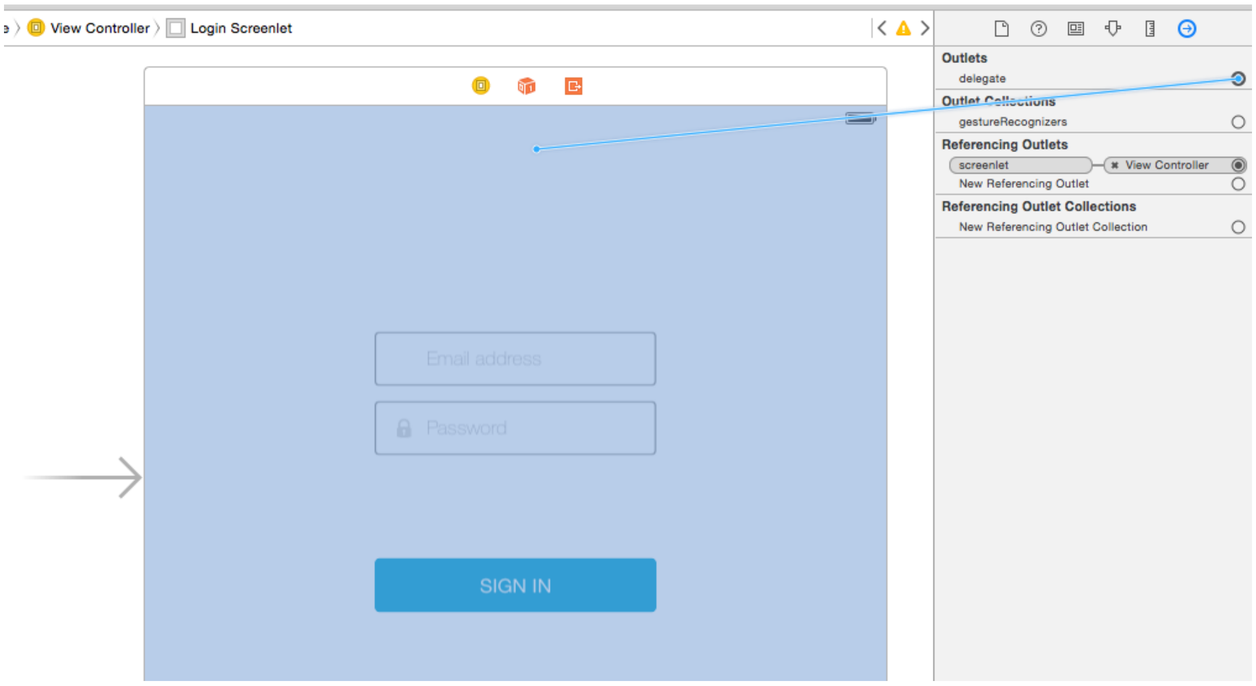
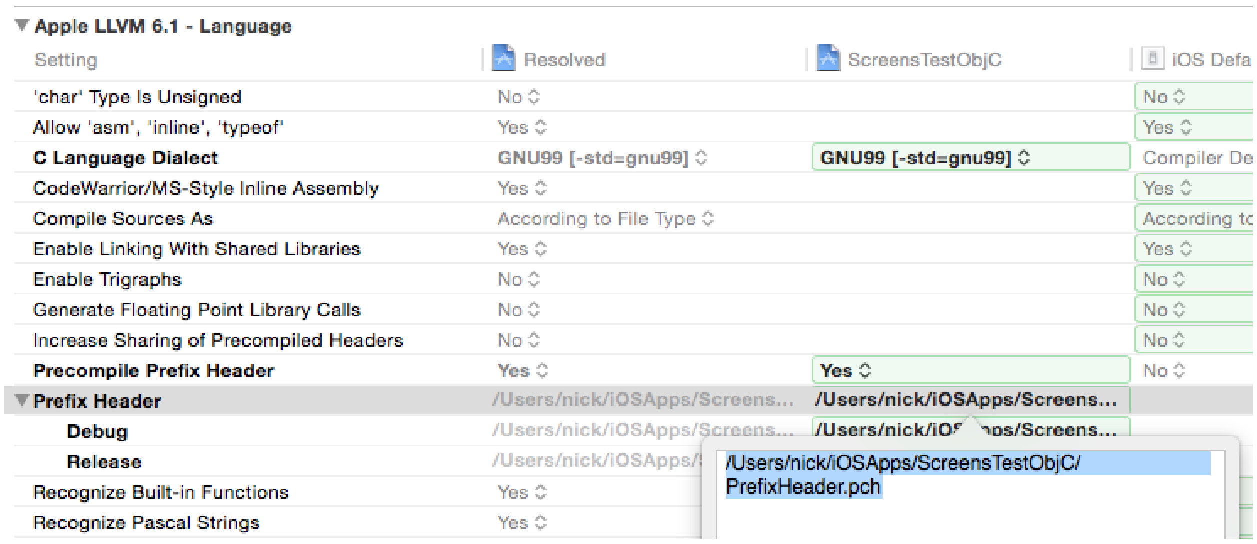


Figure 77.6: Connect the Screenlet's delegate in Interface Builder.



▼ Apple LLVM 6.1 - Language

Setting	Resolved	ScreensTestObjC	iOS Defa
'char' Type Is Unsigned	No ◊		No ◊
Allow 'asm', 'inline', 'typeof'	Yes ◊		Yes ◊
C Language Dialect	GNU99 [-std=gnu99] ◊	**GNU99 [-std=gnu99] ◊**	Compiler De
CodeWarrior/MS-Style Inline Assembly	Yes ◊		Yes ◊
Compile Sources As	According to File Type ◊		According to
Enable Linking With Shared Libraries	Yes ◊		Yes ◊
Enable Trigraphs	No ◊		No ◊
Generate Floating Point Library Calls	No ◊		No ◊
Increase Sharing of Precompiled Headers	No ◊		No ◊
Precompile Prefix Header	**Yes ◊**	**Yes ◊**	No ◊
▼**Prefix Header**	/Users/nick/iOSApps/Screens...	**/Users/nick/iOSApps/Screens...**	
Debug	/Users/nick/iOSApps/Screens...	**/Users/nick/iO ps/Screens...**	
Release	/Users/nick/iOSApps/		
Recognize Built-in Functions	Yes ◊		
Recognize Pascal Strings	Yes ◊		

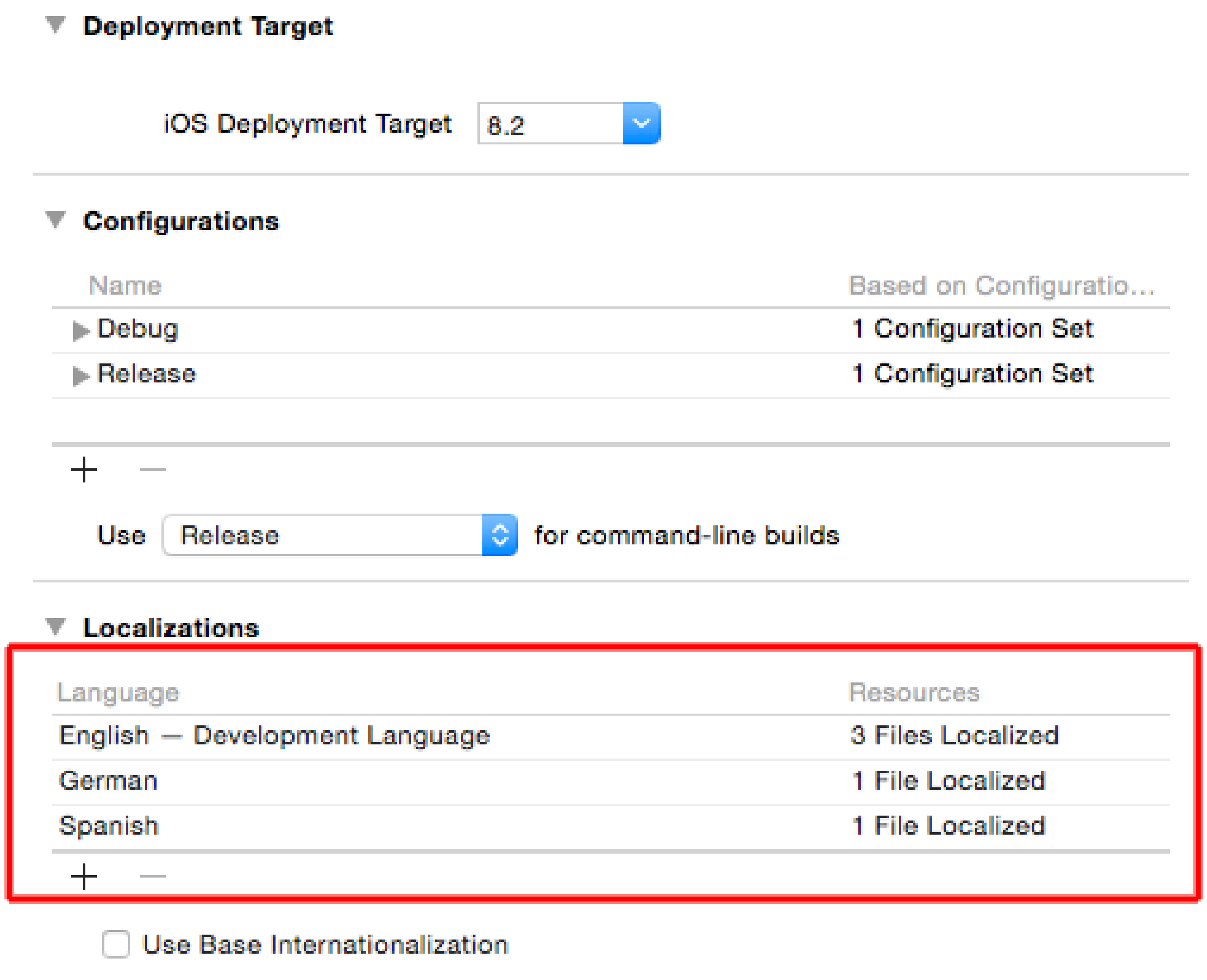
/Users/nick/iOSApps/ScreensTestObjC/
PrefixHeader.pch

Figure 77.7: The `PrefixHeader.pch` configuration in Xcode settings.

You can use the precompiler header file `PrefixHeader.pch` as a template.
Super! Now you know how to use Screenlets from Objective-C code in your apps.

Localizing Screenlets

Follow Apple's standard mechanism to implement localization in your Screenlet. Note: even though a Screenlet may support several languages, you must also support those languages in your app. In other words, a Screenlet's support for a language is only valid if your app supports that language. To support a language, make sure to add it as a localization in your project's settings.



Deployment Target

iOS Deployment Target: 8.2

Configurations

Name	Based on Configuratio...
Debug	1 Configuration Set
Release	1 Configuration Set

Use Release for command-line builds

Localizations

Language	Resources
English — Development Language	3 Files Localized
German	1 File Localized
Spanish	1 File Localized

☐ Use Base Internationalization

Figure 77.8: The Xcode localizations in the project's settings.

Way to go! You now know how to use Screenlets in your iOS apps.

Related Topics

Preparing iOS Projects for Liferay Screens
Using Themes in iOS Screenlets
Creating iOS Screenlets

Using Screenlets in Android apps

77.3 Using Themes in iOS Screenlets

Using a Liferay Screens *Theme*, you can set your Screenlet's UI components, style, and behavior. They let you focus on a Screenlet's UI and UX, without having to worry about its core functionality. Liferay's Screenlets come with several Themes, and more are being developed by Liferay and the community. A Liferay Screenlet's Themes are specified in its reference documentation. This tutorial shows you how to use Themes in your iOS Screenlets.

To install a Theme in your iOS app's Screenlet, you have two options, depending on how the Theme has been published:

1. If the Theme has been packaged as a CocoaPods pod dependency, you can install it by adding a line to your Podfile:

   ```
   pod 'LiferayScreensThemeName'
   ```

   Make sure to replace `LiferayScreensThemeName` with the Theme's CocoaPods project name.

2. If the Theme isn't available through CocoaPods, you can drag and drop the Theme's folder into your project. Liferay Screens detects the new classes and then applies the new design in the runtime and in Interface Builder.

To use the installed Theme, specify its name in the *Theme Name* property field of the *Base Screenlet* in Interface Builder. The names of each Screenlet's Themes are listed in the *Themes* section of the Screenlet's reference documentation. If you leave the Theme name property blank or enter a name for a Theme that can't be found, the Screenlet's Default Theme is used.

The initial release of Liferay Screens for iOS includes the following Themes for its Screenlets:

- *Default*: Comes standard with a Screenlet. It's used by a Screenlet if no Theme name is specified or the named Theme can't be found. The Default Theme can be used as the parent Theme for your custom Themes. Refer to the architecture tutorial for more details.
- *Flat7*: Demonstrates a Theme made from scratch. Refer to the Theme creation tutorial for instructions on creating your own Theme.
- *Westeros*: Customizes the behavior and appearance of the Westeros Bank demo app.

That's all there is to it! Great! Now you know how to use Themes to dress up Screenlets in your iOS app. This opens up a world of possibilities–like writing your own Themes.

Related Topics

Preparing iOS Projects for Liferay Screens

Creating iOS Themes

Using Screenlets in iOS Apps

Architecture of Liferay Screens for iOS

Using Views in Android Screenlets

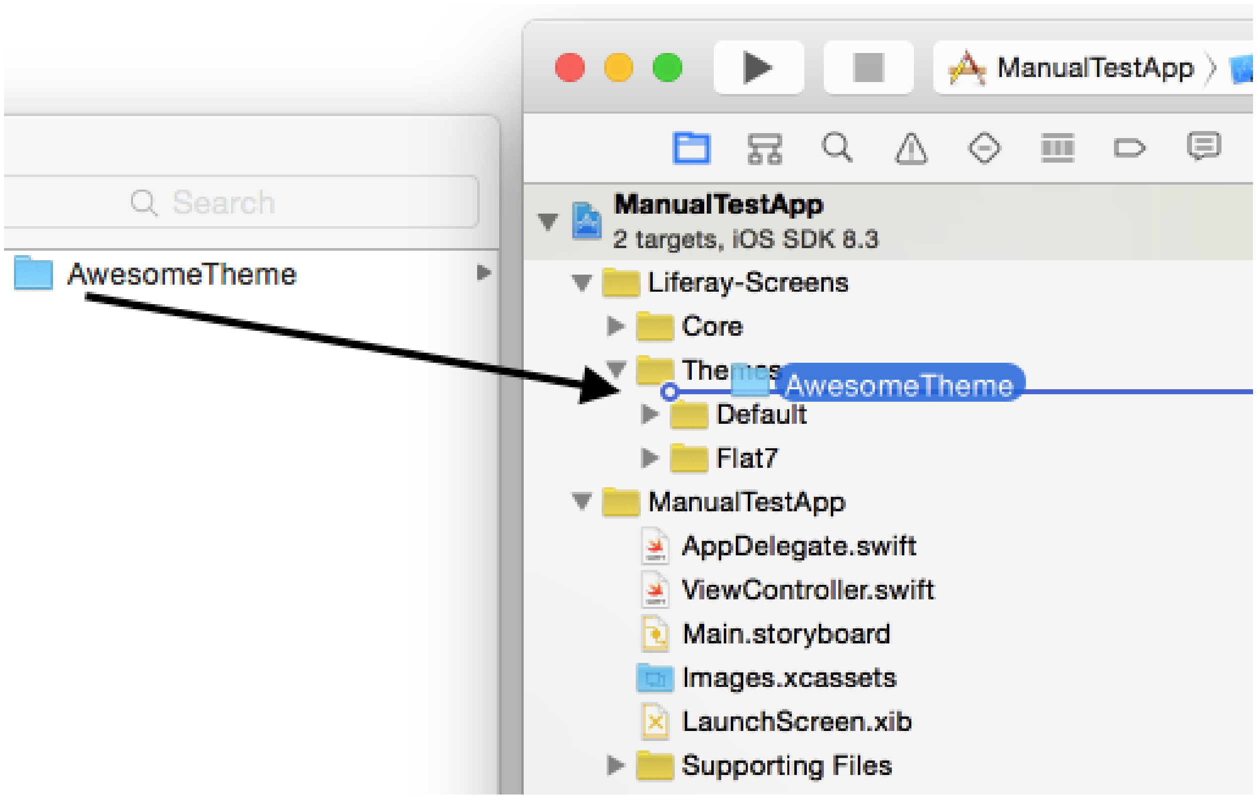
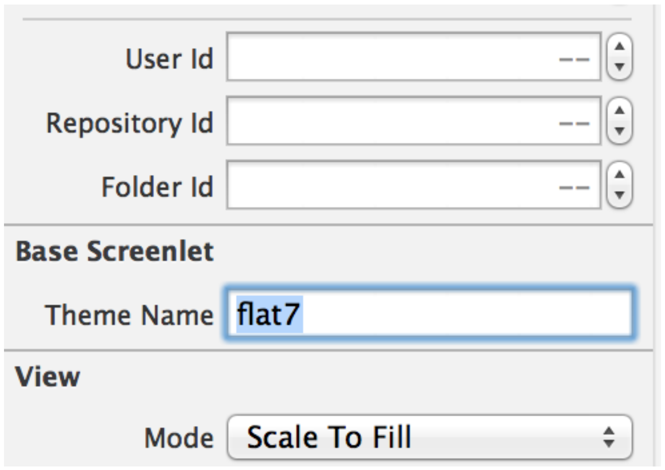


Figure 77.9: To install a Theme into an Xcode project, drag and drop the Theme's folder into it.



User Id	--
Repository Id	--
Folder Id	--

Base Screenlet

Theme Name	flat7

View

Mode	Scale To Fill

Figure 77.10: In Interface Builder, you specify a Screenlet's Theme by entering its name in the *Theme Name* field; this sets the Screenlet's `themeName` property.

77.4 Using Offline Mode in iOS

Offline mode in Liferay Screens lets your apps function when connectivity is unavailable or intermittent. Even though the steady march of technology makes connections more stable and prevalent, there are still plenty of places the Internet has trouble reaching. Areas with complex terrain, including cities with large buildings, often lack stable connections. Remote areas typically don't have connections at all. Using Screens's offline mode in your apps gives your users flexibility in these situations.

This tutorial shows you how to use offline mode in Screenlets. For an explanation of how offline mode works, see the tutorial Architecture of Offline Mode in Liferay Screens. Offline mode's architecture is the same on iOS and Android, although its use on these platforms differs.

Configuring Screenlets for Offline Mode

If you want to enable the offline mode in any of your screenlets, you must configure the attribute `offlinePolicy`. This attribute can take four possible values. For a description of these values, see the section Using Policies with Offline Mode in the offline mode architecture tutorial. Note that each Screenlet behaves a bit differently with offline mode. For specific details, see the Screenlet reference documentation.

Handling Synchronization

Under some scenarios, values stored in the local cache need to be synchronized with the portal. For that purpose you must use the `SyncManager` class. This class is responsible for sending the information stored in the local cache that hasn't been sent to the portal yet.

Use the following steps to start a synchronization process:

1. Create an instance of the `SyncManager` class. You must pass a `CacheManager` object in the constructor. You can get the current cache manager using `SessionContext.currentCacheManager`.

2. Set the delegate property. This delegate object receives the events produced in the synchronization process. For more details on the delegate's methods, see the API reference documentation for the `SyncManagerDelegate` class.

3. Make sure you keep a strong reference to the `SyncManager` object while the process is running.

Related Topics

Architecture of Offline Mode in Liferay Screens

Using Screenlets in iOS Apps

Using Offline Mode in Android

Using Screenlets in Android Apps

77.5 Architecture of Liferay Screens for iOS

Liferay Screens separates its presentation and business-logic code using ideas from Model View Presenter, Model View ViewModel, and VIPER. However, Screens isn't a canonical implementation of these architectures because they're designed for apps. Screens isn't an app; it's a suite of visual components intended for use *in* apps.

This tutorial explains the architecture of Liferay Screens for iOS. It begins with an overview of the high level components that make up the system. This includes the Core, Screenlets, and Themes. These components are then described in detail in the sections that follow. After you get done examining Screens's building blocks, you'll be ready to create some amazing Screenlets and Themes!

High Level Architecture of Liferay Screens for iOS

Liferay Screens for iOS is composed of a Core, a Screenlet layer, a View layer, and Server Connectors. Server Connectors are technically part of the Core, but are worth describing separately. They facilitate interaction with local and remote data sources and communication between the Screenlet layer and the Liferay Mobile SDK.

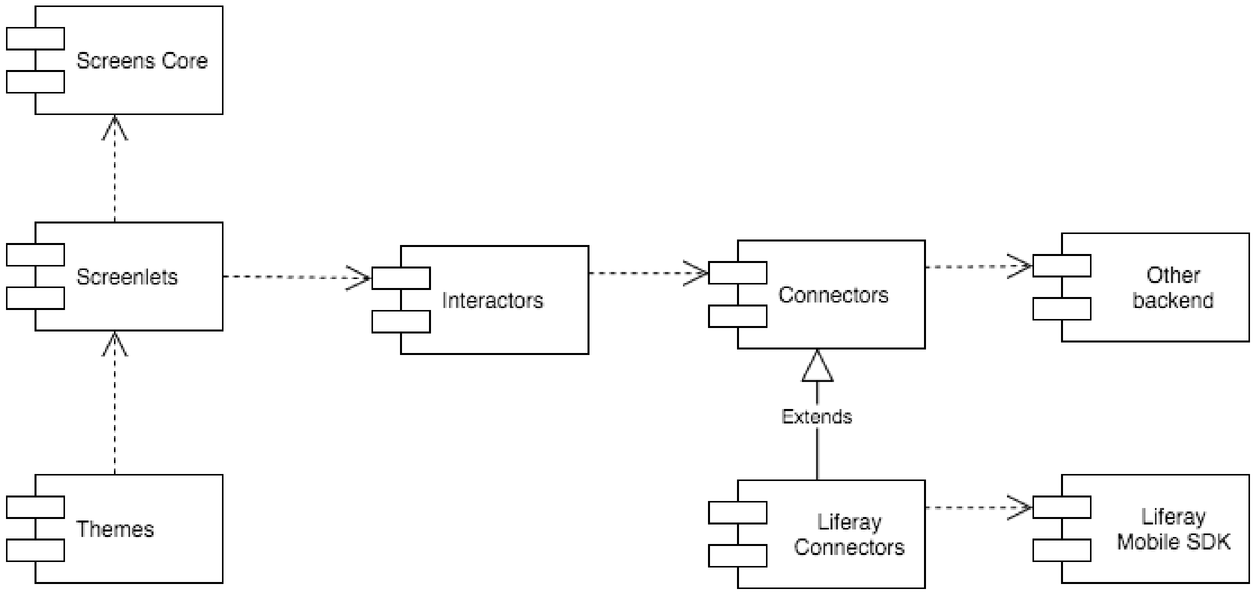


Figure 77.11: The high level components of Liferay Screens for iOS.

Each component is described below.

Core: includes all the base classes for developing other Screens components. It's a micro-framework that lets developers write their own Screenlets, views, and Server Connector classes.

Screenlets: Swift classes for inserting into any `UIView`. They render a selected Theme in the runtime and in Interface Builder. They also react to UI events to start server requests (via Server Connectors), and define a set of `@IBInspectable` properties that can be configured from Interface Builder. The Screenlets bundled with Liferay Screens are known as the Screenlet library.

Interactors: implement specific use cases for communicating with servers or any other data store. Interactors can use local and remote data sources by using *Server Connectors* or custom classes. If a user action or use case needs to execute more than one query on a local or remote store, the sequence is done in the corresponding Interactor. If a Screenlet supports more than one user action or use case, an Interactor must be created for each.

Connectors (or Server Connectors): a collection of classes that can interact with local and remote data sources and Liferay instances. Liferay's own set of Connectors, Liferay Connector, use the Liferay Mobile SDK. All Server Connectors can be run concurrently since they use the `NSOperation` framework. It's very easy to define priorities and dependencies between Connectors, so you can build your own graph of Connectors (aka operations) that can be resolved by the framework. Connectors are always created using a factory class so they can be injected by the app developer.

Themes: a set of XIB files and accompanying `UIView` classes that present Screenlets to the user.

The next section describes the Core in detail.

Core

The Core is the micro-framework that lets developers write Screenlets in a structured and isolated way. All Screenlets share a common structure based on the Core classes, but each Screenlet can have a unique purpose and communication API.

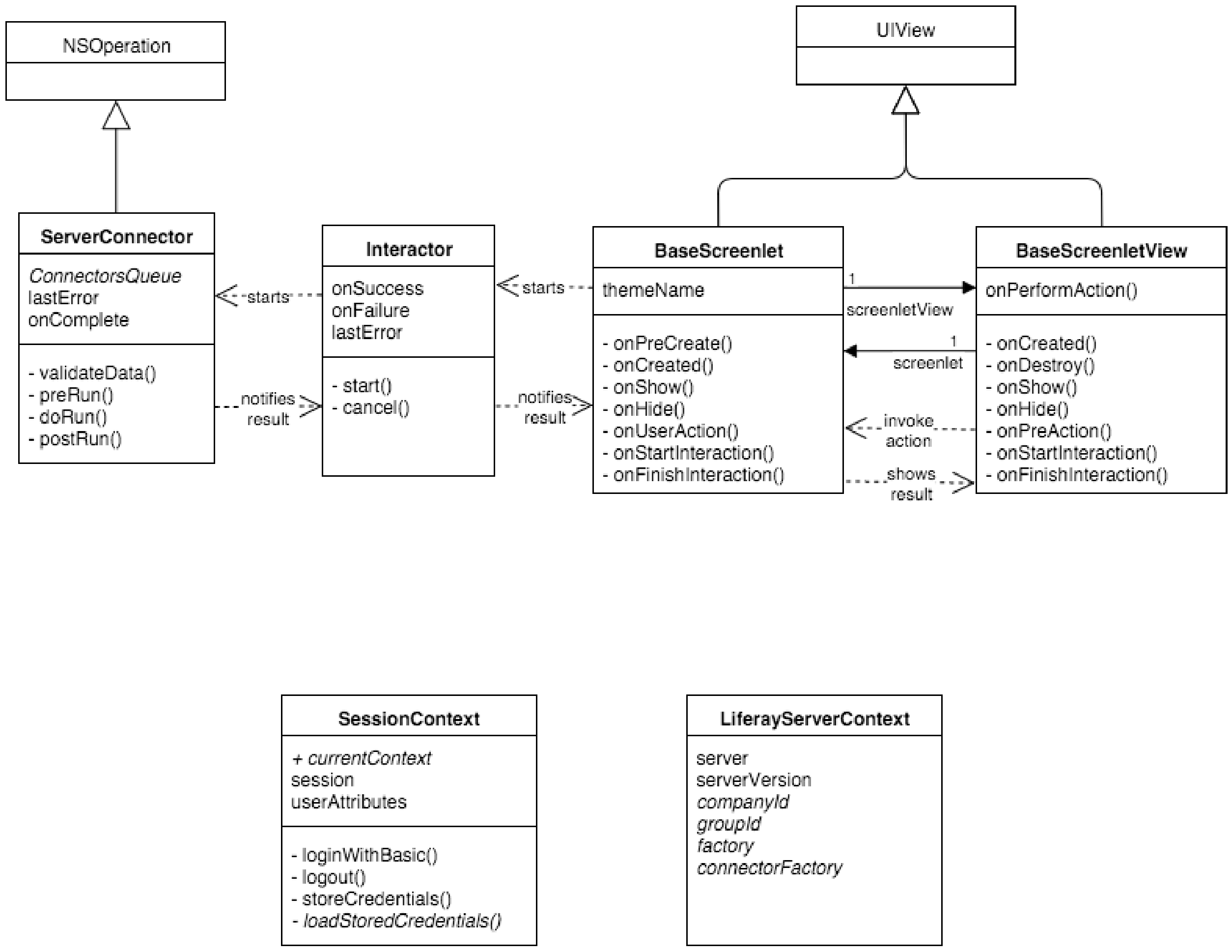


Figure 77.12: Here's the core layer of Liferay Screens for iOS.

From right to left, these are the main components:

BaseScreenletView: the base class for all Screenlet View classes. Its child classes belong to the Theme layer. View classes use standard XIB files to render a UI and then update it when the data changes. The `BaseScreenletView` class contains template methods that child classes may overwrite. When developing your own Theme from a parent Theme, you can read the Screenlet's properties or call its methods from this class. Any user action in the UI is received in this class, and then redirected to the Screenlet class.

BaseScreenlet: the base class for all Screenlet classes. Screenlet classes receive UI events through the `ScreenletView` class, then instantiate Interactors to process and respond to that UI event. When the

Interactor's result is received, the `ScreenletView` (the UI) is updated accordingly. The `BaseScreenlet` class contains a set of template methods that child classes may overwrite.

Interactor: the base class for all Interactors that a Screenlet supports. The Interactor class implements a specific use case supported by the Screenlet. If the Screenlet supports several use cases, it needs different Interactors. If the Interactor needs to retrieve remote data, it uses a Server Connector to do so. When the Server Connector returns the operation's result, the Interactor returns that result to the Screenlet. The Screenlet then changes the `ScreenletView` (the UI) status.

ServerConnector: the base class for all Liferay Connectors that a Screenlet supports. Connectors retrieve data asynchronously from local or remote data sources. The Interactor classes instantiate and start these Connector classes.

SessionContext: an object (typically a singleton) that holds the logged in user's session. Apps can use an implicit login, invisible to the user, or a login that relies on explicit user input to create the session. User logins can be implemented with Login Screenlet. This is explained in detail here.

LiferayServerContext: a singleton object that holds server configuration parameters. It's loaded from the `liferay-server-context.plist` file. Most Screenlets use these parameters as default values.

Now that you know what the Core contains, you're ready to learn the Screenlet layer's details.

Screenlet Layer

The Screenlet layer contains the available Screenlets in Liferay Screens for iOS. The following diagram shows the Screenlet layer in relation to the Core, Interactor, Theme, and Connector layers. The Screenlet classes in the diagram are explained in this section.

Screenlets are comprised of several Swift classes and an XIB file:

MyScreenletViewModel: a protocol that defines the attributes shown in the UI. It typically accounts for all the input and output values presented to the user. For example, `LoginViewModel` includes attributes like the user name and password. A Connector can be configured by reading and validating these values. Also, the Screenlet can change these values based on any default values and operation results.

MyScreenlet: a class that represents the Screenlet component the app developer interacts with. It includes the following things:

- Inspectable parameters for configuring the Screenlet's behavior. The initial state can be set in the Screenlet's data.
- A reference to the Screenlet's View, based on the selected Theme. To meet the Screenlet's requirements, all Themes must implement the `ViewModel` protocol.
- Any number of methods for invoking Connectors. You can optionally make them public for app developers to call.

- An optional (but recommended) delegate object the Screenlet can call on for particular events.

MyUserCaseInteractor: Each Interactor runs the operations that implement the use case. These can be local operations, remote operations, or a combination thereof. Operations can be executed sequentially or in parallel. The final results are stored in a `result` object that can be read by the Screenlet when notified. The number of Interactor classes a Screenlet requires depends on the number of use cases it supports.

MyOperationConnector: This is related to the Interactor, but has one or more Connectors. If the Server Connector is a back-end call, then there's typically only a single request. Each Server Connector is responsible for retrieving a set of related values. The results are stored in a `result` object that can be read by the Interactor when notified. The number of Server Connector classes an Interactor requires depends on the number of endpoints you need to query, or even the number of different servers you need to support. Connectors are

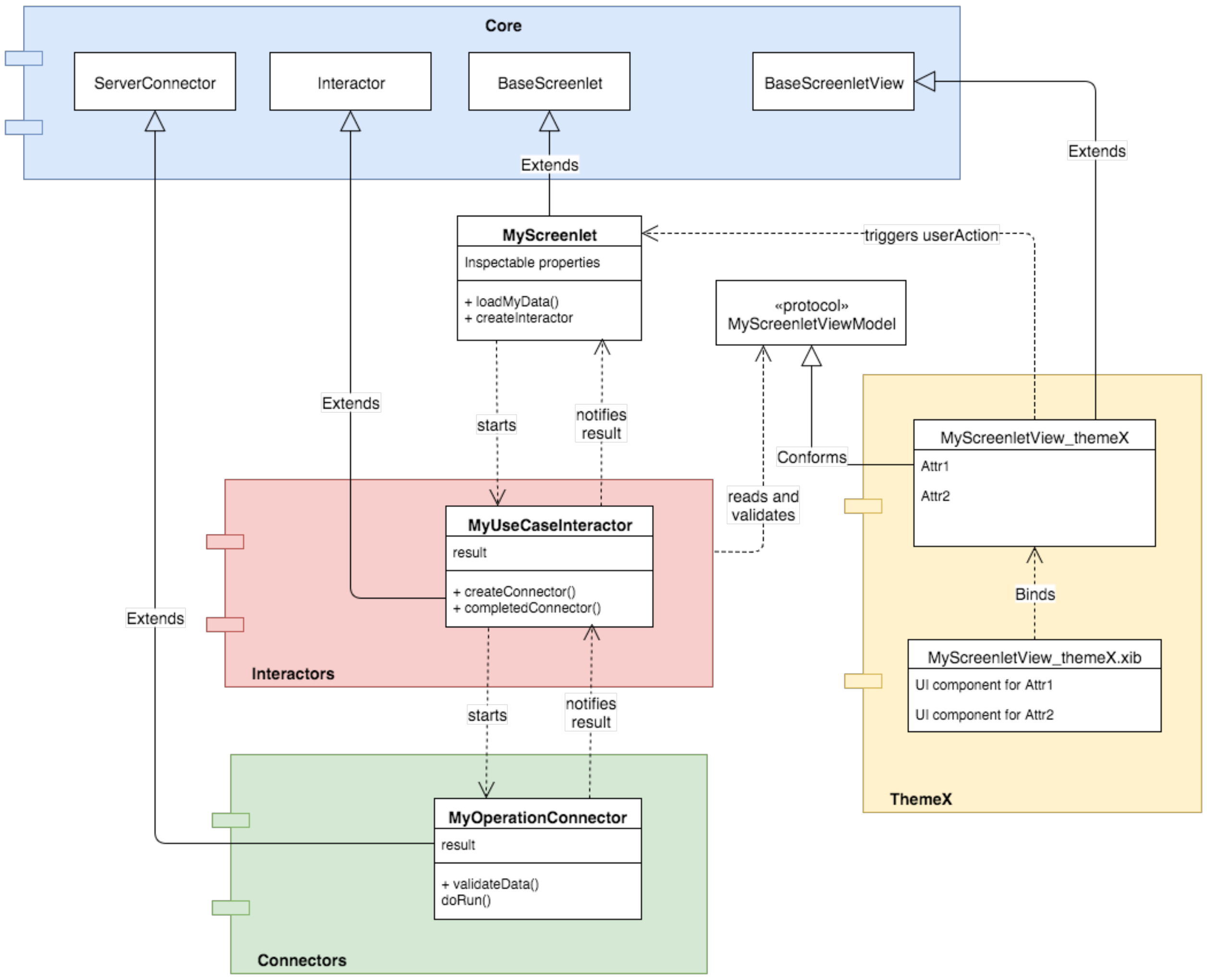


Figure 77.13: This diagram illustrates the iOS Screenlet Layer's relationship to other Screens components.

always created using a factory class. You can therefore take advantage of Inversion of Control. This way, you can implement your own factory class to use to create your own Connector objects. To tell Screens to use your factory class, specify it in the `liferay-server-context.plist` file as described in the tutorial on preparing your iOS project for Screens.

MyScreenletView_themeX: A class that belongs to one specific Theme. In the diagram, this Theme is *ThemeX*. The class renders the Screenlet's UI by using its related XIB file. The View object and XIB file communicate using standard mechanisms like `@IBOutlet` and `@IBAction`. When a user action occurs in the XIB file, it's received by `BaseScreenletView` and then passed to the Screenlet class via the `performAction` method. To identify different events, the component's `restorationIdentifier` property is passed to the `performAction` method.

MyScreenletView_themeX.xib: an XIB file that specifies how to render the Screenlet's View. Its name is very important. By convention, a Screenlet with a view class named *FooScreenletView* and a Theme named *BarTheme* must have an XIB file named `FooScreenletView_barTheme.xib`.

For more details, refer to the tutorial Creating iOS Screenlets. Next, the Theme Layer of Screens for iOS is described.

Theme Layer

The Theme Layer lets developers set a Screenlet's look and feel. The Screenlet property `themeName` determines the Theme to load. This can be set by the Screenlet's *Theme Name* field in Interface Builder. A Theme consists of a view class for Screenlet behavior and an XIB file for the UI. By inheriting one or more of these components from another Theme, the different Theme *types* allow varying levels of control over a Screenlet's UI design and behavior.

There are several different Theme types:

Default Theme: The standard Theme provided by Liferay. It can be used as a template to create other Themes, or as the parent Theme of other Themes. Each Theme for each Screenlet requires a View class. A Default Theme's View class is named `MyScreenletView_default`, where `MyScreenlet` is the Screenlet's name. This class is similar to the standard `ViewController` in iOS; it receives and handles UI events by using the standard `@IBAction` and `@IBOutlet`. The View class usually uses an XIB file to build the UI components. This XIB file is bound to the class.

Child Theme: Presents the same UI components as the parent Theme, but can change the UI components' appearance and position. A Child Theme specifies visual changes in its own XIB file; it can't add or remove any UI components. In the diagram, the Child Theme inherits from the Default Theme. Creating a Child Theme is ideal when you only need to make visual changes to an existing Theme. For example, you can create a Child Theme that sets new positions and sizes for the standard text boxes in Login Screenlet's Default Theme, but without adding or overwriting existing code.

Full: Provides a complete standalone theme. It has no parent Theme and implements unique behavior and appearance for a Screenlet. Its View class must extend Screens's `BaseScreenletView` class and conform to the Screenlet's view model protocol. It must also specify a new UI in an XIB file. Refer to the Default Theme for an example of a Full Theme.

Extended: Inherits the parent Theme's behavior and appearance, but lets you change and add code to both. You can do so by creating a new XIB file and a custom View class that extends the parent Theme's View class. In the diagram, the Extended Theme inherits the Full Theme and extends its Screenlet's View class. Refer to the Flat7 Theme for an example of an Extended Theme.

Themes in Liferay Screens are organized into sets that contain Themes for several Screenlets. Liferay's available Theme sets are listed here:

- *Default:* A mandatory Theme set supplied by Liferay. It's used if the Screenlet's `themeName` isn't specified or is invalid. The Default Theme uses a neutral, flat white and blue design with standard UI components. For example, the Login Screenlet uses standard text boxes for the user name and password fields, but uses the Default Theme's flat white and blue design.

- *Flat7:* A collection of Themes that use a flat black and green design, and UI components with rounded edges. They're Extended Themes.

- *Westeros:* The Theme for the Bank of Westeros sample app.

For more details on Theme creation, see the tutorial Creating iOS Themes.

Awesome! Now you know the nitty gritty details of Liferay Screens for iOS. This information is invaluable when using Screens to develop your apps.

Related Topics

Using Screenlets in iOS Apps

Using Themes in iOS Screenlets

Creating iOS Screenlets

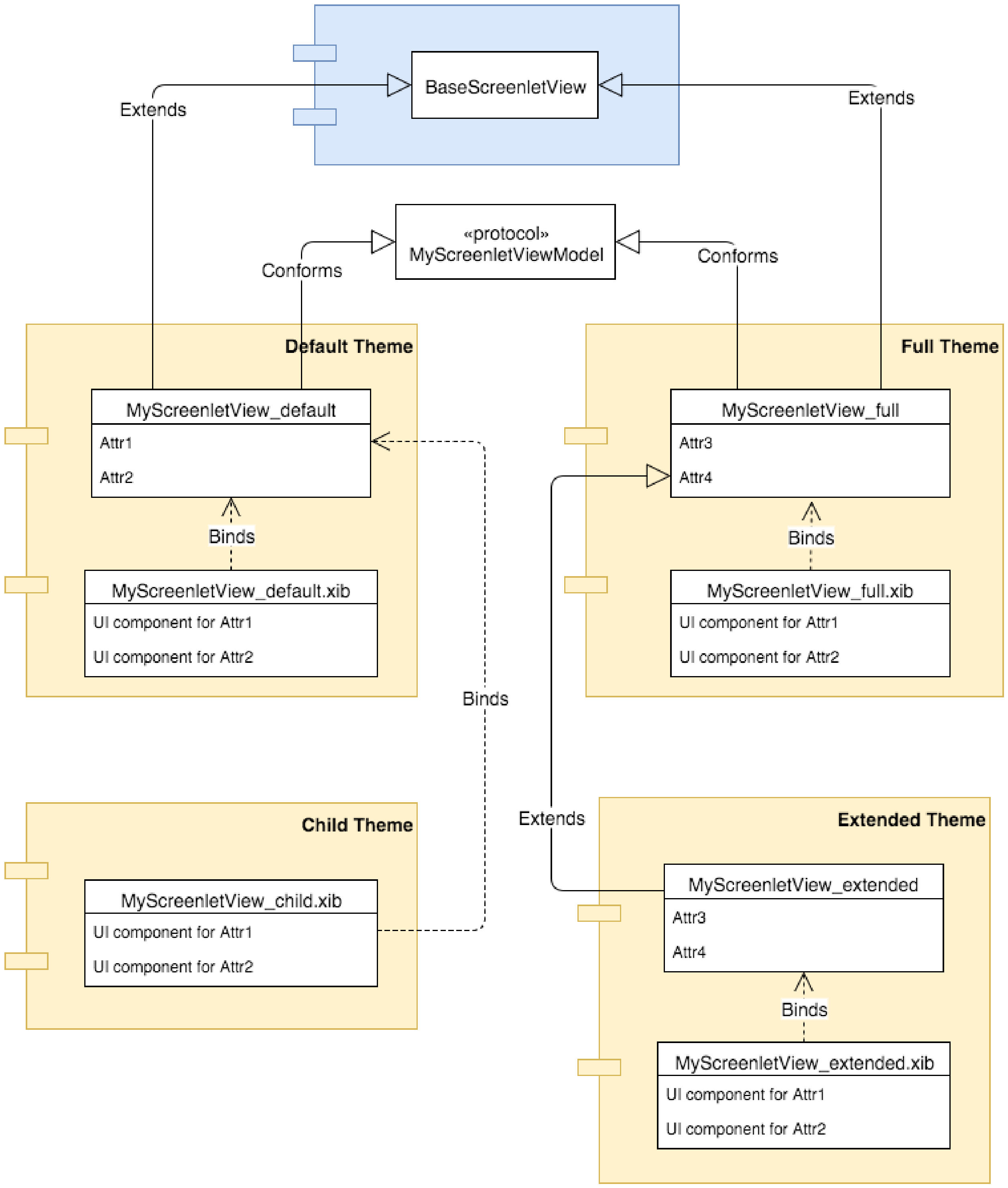


Figure 77.14: The Theme Layer of Liferay Screens for iOS.

77.6 Creating iOS Screenlets

The built-in Screenlets cover common use cases for mobile apps that use Liferay. They authenticate users, interact with Dynamic Data Lists, display assets, and more. What if, however, there's no Screenlet for *your* use case? No problem! You can create your own. Extensibility is a key strength of Liferay Screens.

This tutorial explains how to create your own Screenlets. As an example, it references code from the sample Add Bookmark Screenlet, that saves bookmarks to Liferay's Bookmarks portlet.

In general, you use the following steps to create Screenlets:

1. Plan Your Screenlet: Your Screenlet's features and use cases determine where you'll create it and the Liferay remote services you'll call.

2. Create Your Screenlet's UI (its Theme): Although this tutorial presents all the information you need to create a Theme for your Screenlet, you may first want to learn the steps for creating a Theme. For more information on Themes in general, see the tutorial on using Themes with Screenlets.

3. Create the Screenlet's Interactor. Interactors are Screenlet components that make server calls.

4. Create the Screenlet class. The Screenlet class is the Screenlet's central component. It controls the Screenlet's behavior and is the component the app developer interacts with when inserting a Screenlet.

Before getting started, make sure that you're familiar with the architecture of Liferay Screens. Click here to read the Screens architecture tutorial.

Without further ado, let the Screenlet creation begin!

Planning Your Screenlet

Before creating your Screenlet, you must determine what it needs to do and how you want developers to use it. This determines where you'll create your Screenlet and its functionality.

Where you should create your Screenlet depends on how you plan to use it. If you want to reuse or redistribute it, you should create it in an empty Cocoa Touch Framework project in Xcode. You can then use CocoaPods to publish it. The tutorial Packaging iOS Themes explains how to publish an iOS Screenlet. Even though that tutorial refers to Themes, the steps for preparing Screenlets for publication are the same. If you don't plan to reuse or redistribute your Screenlet, create it in your app's Xcode project.

You must also determine your Screenlet's functionality and what data your Screenlet requires. This determines the actions your Screenlet must support and the Liferay remote services it must call. For example, Add Bookmark Screenlet only needs to respond to one action: adding a bookmark to Liferay's Bookmarks portlet. To add a bookmark, this Screenlet must call the Liferay instance's `add-entry` service for `BookmarksEntry`. If you're running a Liferay instance locally on port 8080, click here to see this service. To add a bookmark, this service requires the following parameters:

- `groupId`: The site ID in the Liferay instance that contains the Bookmarks portlet.

- `folderId`: The folder ID in the Bookmarks portlet that receives the new bookmark.

- `name`: The new bookmark's title.

- `url`: The new bookmark's URL.

- `description`: The new bookmark's description.
- `serviceContext`: A Liferay `ServiceContext` object.

Add Bookmark Screenlet must therefore account for each of these parameters. When saving a bookmark, the Screenlet asks the user to enter the bookmark's URL and name. The user isn't required, however, to enter any other parameters. This is because the app developer sets the `groupId` and `folderId` via the app's code. Also, the Screenlet's code automatically populates the `description` and `serviceContext`.

Great! Now you're ready to create your Screenlet's Theme!

Creating the Screenlet's UI

In Liferay Screens for iOS, a Screenlet's UI is called a Theme. Every Screenlet must have at least one Theme. A Theme has the following components:

1. An XIB file: defines the UI components that the Theme presents to the end user.
2. A View class: renders the UI, handles user interactions, and communicates with the Screenlet class.

First, create a new XIB file and use Interface Builder to construct your Screenlet's UI. In many cases, the Screenlet's actions must be triggered from the Theme. To achieve this, make sure to use a `restorationIdentifier` property to assign a unique ID to each UI component that triggers an action. The user triggers the action by interacting with the UI component. If the action only changes the UI's state (that is, changes the UI component's properties), then you can associate that component's event to an `IBAction` method as usual. Actions using `restorationIdentifier` are intended for use by actions that need an Interactor, such as actions that make server requests or retrieve data from a database.

For example, Add Bookmark Screenlet's UI needs text boxes for entering a bookmark's URL and title. This UI also needs a button to support the Screenlet's action: sending the bookmark to a Liferay instance. The XIB file `AddBookmarkView_default.xib` defines this UI. Because the button triggers the Screenlet's action, it contains `restorationIdentifier="add-bookmark"`.

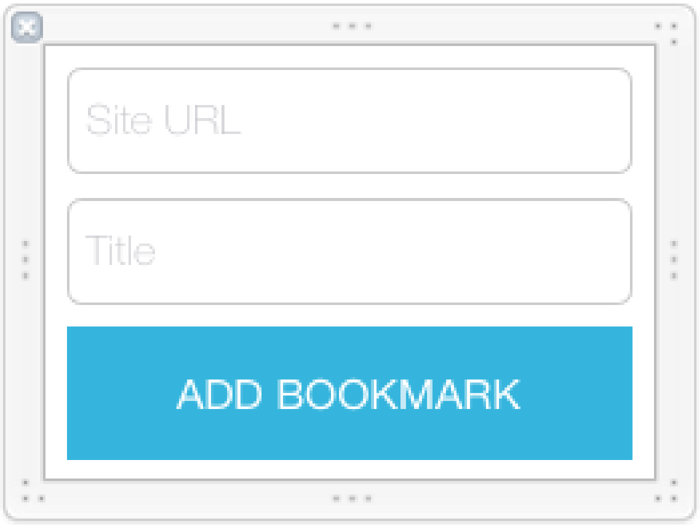


Figure 77.15: Here's the sample Add Bookmark Screenlet's XIB file rendered in Interface Builder.

Note: The Screenlet in this tutorial doesn't support multiple Themes. If you want your Screenlet to support multiple Themes, your View class must also conform a *View Model* protocol that you create. For instructions on this, see the tutorial Supporting Multiple Themes in Your Screenlet.

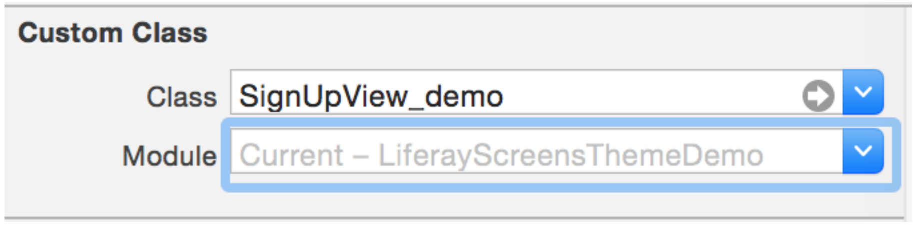
Now you must create your Screenlet's View class. This class controls the UI you just defined. In the `BaseScreenletView` class, Screens provides the default functionality required by all View classes. Your View class must therefore extend `BaseScreenletView` to provide the functionality unique to your Screenlet. To support your UI, use standard `@IBOutlets` and `@IBActions` to connect all your XIB's UI components and events to your View class. You should also implement getters and setters to get values from and set values to the UI components. Your View class should also implement any required animations or front-end logic.

For example, `AddBookmarkView_default` is Add Bookmark Screenlet's View class. This class extends `BaseScreenletView` and contains `@IBOutlet` references to the XIB's text fields. The getters for these references let the Theme retrieve the data the user enters into the corresponding text field:

```
import UIKit
import LiferayScreens

class AddBookmarkView_default: BaseScreenletView {

   @IBOutlet weak var URLTextField: UITextField?
   @IBOutlet weak var titleTextField: UITextField?

   var URL: String? {
       return URLTextField?.text
   }

   var title: String? {
       return titleTextField?.text
   }
}
```

In Interface Builder, you must now specify your View class as your XIB file's custom class. In Add Bookmark Screenlet, for example, `AddBookmarkView_default` is set as the `AddBookmarkView_default.xib` file's custom class in Interface Builder.

If you're using CocoaPods, make sure to explicitly set a valid module for the custom class–the grayed-out *Current* default value only suggests a module.


Custom Class

Class: SignUpView_demo

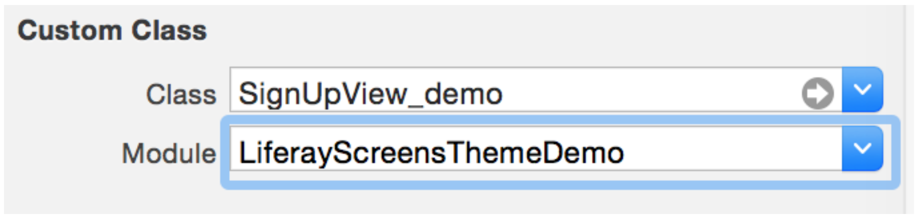
Module: Current – LiferayScreensThemeDemo

Figure 77.16: In this XIB file, the custom class's module is NOT specified.

Next, you'll create your Screenlet's Interactor.

Creating the Interactor

Create an Interactor class for each of your Screenlet's actions. In the Interactor class, Screens provides the default functionality required by all Interactor classes. Your Interactor class must therefore extend Interactor to provide the functionality unique to your Screenlet.


Custom Class

Class SignUpView_demo

Module LiferayScreensThemeDemo

Figure 77.17: The XIB file is bound to the custom class name, with the specified module.

Note: You may wish to make your server call in a Connector instead of an Interactor. Doing so provides an additional abstraction layer for your server call, leaving your Interactor to instantiate your Connector and receive its results. For instructions on this, see the tutorial Create and Use a Connector with Your Screenlet.

Interactors work synchronously, but you can use callbacks (delegates) or Connectors to run their operations in the background. For example, the Liferay Mobile SDK provides the `LRCallback` protocol for this purpose. This is described in the Mobile SDK tutorial on invoking Liferay services asynchronously. Screens bridges this protocol to make it available in Swift. Your Interactor class can conform this protocol to make its server calls asynchronously. To implement an Interactor class:

- Your initializer must receive all required properties and a reference to the Screenlet.
- Override Interactor's `start` method to perform the server operations your Screenlet requires (e.g., invoke a Liferay operation via a Liferay Mobile SDK service).
- Save the server response to an accessible property, if necessary. For example, if the server call returns objects from a Liferay instance, you should store these objects in an accessible property. This way your Screenlet can display those results to the user.
- You must invoke the methods `callOnSuccess` and `callOnFailure` to execute the closures `onSuccess` and `onFailure`, respectively.

For example, the sample Add Bookmark Screenlet's Interactor class `AddBookmarkInteractor` makes the server call that adds a bookmark to a Liferay instance. This class extends the `Interactor` class and conforms the `LRCallback` protocol. The latter ensures that the Interactor's server call runs asynchronously:

```
public class AddBookmarkInteractor: Interactor, LRCallback {...
```

To save the server call's results, `AddBookmarkInteractor` defines the public variable `resultBookmarkInfo`. This class also defines public constants for the bookmark's folder ID, title, and URL. The initializer sets these variables and calls Interactor's constructor with a reference to the base Screenlet class (`BaseScreenlet`):

```
public var resultBookmarkInfo: [String:AnyObject]?
public let folderId: Int64
public let title: String
public let url: String

public init(screenlet: BaseScreenlet, folderId: Int64, title: String, url: String) {
    self.folderId = folderId
    self.title = title
    self.url = url
    super.init(screenlet: screenlet)
}
```

The `AddBookmarkInteractor` class's `start` method makes the server call. To do so, it must first get a `Session`. Since Login Screenlet creates a session automatically upon successful login, the `start` method retrieves this session with SessionContext.createSessionFromCurrentSession(). To make the server call asynchronously, the start method must set a callback to this session. Because AddBookmarkInteractor conforms the LRCallback protocol, setting self as the session's callback accomplishes this. The start method must then create a LRBookmarksEntryService_v7 instance and call this instance's addEntryWithGroupId method. The latter method calls a Liferay instance's add-entry service for BookmarksEntry. The start method therefore provides the groupId, folderId, name, url, description, and serviceContext arguments to addEntryWithGroupId. Note that this example provides a hard-coded string for the description. Also, the serviceContext is nil because the Mobile SDK handles the ServiceContext object for you:

```
override public func start() -> Bool {
    let session = SessionContext.createSessionFromCurrentSession()
    session?.callback = self

    let service = LRBookmarksEntryService_v7(session: session)

    do {
        try service.addEntryWithGroupId(LiferayServerContext.groupId,
                        folderId: folderId,
                        name: title,
                        url: url,
                        description: "Added from Liferay Screens",
                        serviceContext: nil)

        return true
    }
    catch {
        return false
    }
}
```

Finally, the AddBookmarkInteractor class must conform the LRCallback protocol by implementing the onFailure and onSuccess methods. The onFailure method communicates the NSError object that results from a failed server call. It does this by calling the base Interactor class's callOnFailure method with the error. When the server call succeeds, the onSuccess method sets the server call's results (the result argument) to the resultBookmarkInfo variable. The onSuccess method finishes by calling the base Interactor class's callOnSuccess method to communicate the success status throughout the Screenlet:

```
public func onFailure(error: NSError!) {
    self.callOnFailure(error)
}

public func onSuccess(result: AnyObject!) {
    //Save result bookmark info
    resultBookmarkInfo = (result as! [String:AnyObject])

    self.callOnSuccess()
}
```

Next, you'll create the Screenlet class.

Creating the Screenlet Class

The Screenlet class is the central hub of a Screenlet. It contains the Screenlet's properties, a reference to the Screenlet's View class, methods for invoking Interactor operations, and more. When using a Screenlet, app developers primarily interact with its Screenlet class. In other words, if a Screenlet were to become self-aware, it would happen in its Screenlet class (though we're reasonably confident this won't happen).

Screens's BaseScreenlet class is a base Screenlet class implementation. Since BaseScreenlet provides most of a Screenlet class's required functionality, your Screenlet class should extend BaseScreenlet. This lets you focus on your Screenlet's unique functionality. Your Screenlet class must also include any @IBInspectable properties your Screenlet requires and a reference to your Screenlet's View class. To perform your Screenlet's action, your Screenlet class must override BaseScreenlet's createInteractor method. This method should create an instance of your Interactor and then set the Interactor's onSuccess and onFailure closures to define what happens when the server call succeeds or fails, respectively.

For example, the AddBookmarkScreenlet class is the Screenlet class in Add Bookmark Screenlet. This class extends BaseScreenlet and contains an @IBInspectable variable for the bookmark folder's ID (folderId). The AddBookmarkScreenlet class's createInteractor method first gets a reference to the View class (AddBookmarkView_default). It then creates an AddBookmarkInteractor instance with this Screenlet class (self), the folderId, the bookmark's title, and the bookmark's URL. Note that the View class reference contains the bookmark title and URL that the user entered into the UI. The createInteractor method then sets the Interactor's onSuccess closure to print a success message when the server call succeeds. Likewise, the Interactor's onFailure closure is set to print an error message when the server call fails. Note that you're not restricted to only printing messages here: you should set these closures to do whatever is best for your Screenlet. The createInteractor method finishes by returning the Interactor instance. Here's the complete AddBookmarkScreenlet class:

```
import UIKit
import LiferayScreens


public class AddBookmarkScreenlet: BaseScreenlet {

    //MARK: Inspectables

    @IBInspectable var folderId: Int64 = 0

    //MARK: BaseScreenlet

    override public func createInteractor(name name: String?, sender: AnyObject?) -> Interactor? {

        let view = self.screenletView as! AddBookmarkView_default

        let interactor = AddBookmarkInteractor(screenlet: self,
                                         folderId: folderId,
                                         title: view.title!,
                                         url: view.URL!)

        //Called when the Interactor's server call finishes succesfully
        interactor.onSuccess = {
            let bookmarkName = interactor.resultBookmarkInfo!["name"] as! String
            print("Bookmark \"\(bookmarkName)\" saved!")
        }

        //Called when the Interactor's server call fails
        interactor.onFailure = { _ in
            print("An error occurred saving the bookmark")
        }

        return interactor
    }

}
```

For reference, the sample Add Bookmark Screenlet's final code is here on GitHub.

You're done! Your Screenlet is a ready-to-use component that you can add to your storyboard. You can even package it to contribute to the Screens project or distribute it with CocoaPods. Now you know how to create iOS Screenlets!

Related Topics

Supporting Multiple Themes in Your Screenlet
    Adding Screenlet Actions
    Create and Use a Connector with Your Screenlet
    Add a Screenlet Delegate
    Using and Creating Progress Presenters
    Creating and Using Your Screenlet's Model Class
    Using Screenlets in iOS Apps
    Architecture of Liferay Screens for iOS
    Creating iOS Themes
    Creating Android Screenlets

77.7 Supporting Multiple Themes in Your Screenlet

Themes let you present the same Screenlet with a different look and feel. For example, if you have multiple apps that use the same Screenlet, you can use different Themes to match the Screenlet's appearance to each app's style. Each Screenlet that comes with Liferay Screens supports the use of multiple Themes. For your custom Screenlet to support different Themes, however, it must contain a *View Model* protocol. A View Model abstracts the Theme used to display the Screenlet, thus letting developers use other Themes. For example, note that the Screenlet class's `createInteractor` method in the Screenlet creation tutorial accesses the View class (`AddBookmarkView_default`) directly when getting a reference to the View class:

```
let view = self.screenletView as! AddBookmarkView_default
```

This is all fine and well, except it hard codes the Theme defined by `AddBookmarkView_default`! To use a different Theme, you'd have to rewrite this line of code to use that Theme's View class. This isn't very flexible! Instead of making your Screenlet take expensive yoga classes, you can abstract the Theme's View class via a View Model protocol.

This tutorial shows you how to add a View Model to your Screenlet. The Add Bookmark Screenlet created in the Screenlet creation tutorial is used as an example. Note that you can also follow these steps to add a View Model while creating your Screenlet.

Creating and Using a View Model

Follow these steps to add and use a View Model in your Screenlet:

1. Create a View Model protocol that defines your Screenlet's attributes. These attributes are the View class properties your Screenlet class uses. For example, the Screenlet class in Add Bookmark Screenlet uses the View class properties `title` and `URL`. Add Bookmark Screenlet's View Model protocol (`AddBookmarkViewModel`) must therefore define variables for these properties:

```
import UIKit

@objc protocol AddBookmarkViewModel {

    var URL: String? {get}

    var title: String? {get}

}
```

2. Conform your View class to your Screenlet's View Model protocol. Make sure to get/set all the protocol's properties. For example, here's Add Bookmark Screenlet's View Class (`AddBookmarkView_default`) conformed to `AddBookmarkViewModel`:

```
import UIKit
import LiferayScreens

class AddBookmarkView_default: BaseScreenletView, AddBookmarkViewModel {

    @IBOutlet weak var URLTextField: UITextField?
    @IBOutlet weak var titleTextField: UITextField?

    var URL: String? {
        return URLTextField?.text
    }

    var title: String? {
        return titleTextField?.text
    }

}
```

3. Create and use a View Model reference in your Screenlet class. By retrieving data from this reference instead of a direct View class reference, you can use your Screenlet with other Themes. For example, here's the `AddBookmarkScreenlet` class with a `viewModel` property instead of a direct reference to `AddBookmarkView_default`. This class's `createInteractor` method then uses this property to get the `title` and `URL` properties in the `AddBookmarkInteractor` constructor:

```
...
//View Model reference
var viewModel: AddBookmarkViewModel {
    return self.screenletView as! AddBookmarkViewModel
}

override public func createInteractor(name name: String?, sender: AnyObject?) -> Interactor? {

    let interactor = AddBookmarkInteractor(screenlet: self,
                                           folderId: folderId,
                                           title: viewModel.title!,
                                           url: viewModel.URL!)

    // Called when the Interactor finishes succesfully
    interactor.onSuccess = {
        let bookmarkName = interactor.resultBookmarkInfo!["name"] as! String
        print("Bookmark \"\(bookmarkName)\" saved!")
    }

    // Called when the Interactor finishes with an error
    interactor.onFailure = { _ in
        print("An error occurred saving the bookmark")
    }

    return interactor
}
...
```

That's it! Now your Screenlet is ready to use other Themes that you create for it. See the tutorial Creating iOS Themes for instructions on creating a Theme.

Related Topics

Creating iOS Themes
    Creating iOS Screenlets
    Architecture of Liferay Screens for iOS
    Creating iOS List Screenlets

77.8 Adding Screenlet Actions

With multiple Interactors, it's possible for a Screenlet to have multiple actions. You must create an Interactor class for each action. For example, if your Screenlet needs to make two server calls, then you need two Interactors: one for each call. Your Screenlet class's `createInteractor` method must return an instance of each Interactor. Also, recall that each action name is given by the `restorationIdentifier` of the UI components that trigger them. You should set this `restorationIdentifier` to a constant in your Screenlet.

This tutorial walks you through the steps necessary to add an action to your Screenlet, and trigger an action programmatically. As an example, this tutorial uses the advanced version of the sample Add Bookmark Screenlet. This Screenlet is similar to the sample Add Bookmark Screenlet created in the Screenlet creation tutorial. The advanced Add Bookmark Screenlet, however, contains two actions:

1. Add Bookmark: Adds a bookmark to the Bookmarks portlet in a Liferay DXP installation. This is the Screenlet's main action, created in the Screenlet creation tutorial.

2. Get Title: Retrieves the title from a bookmark URL entered by the user. This tutorial shows you how to implement this action.

Note that this tutorial doesn't explain Screenlet creation in general. Before proceeding, make sure you've read the Screenlet creation tutorial. And without any further ado, it's time to implement your Screenlet's action!

Implementing Your Action

Use the following steps to add an action to your your Screenlet:

1. Create a constant in your Screenlet class for each of your Screenlet's actions. For example, here are the constants in Add Bookmark Screenlet's Screenlet class (`AddBookmarkScreenlet`):

   ```
   static let AddBookmarkAction = "add-bookmark"
   static let GetTitleAction = "get-title"
   ```

2. In your Theme's XIB file, add any new UI components that you need to perform the action. For example, Add Boookmark Screenlet's XIB file needs a new button for getting the URL's title:

3. Wire the UI components in your XIB file to your View class. In your View class, you must also register the events you want to react to (e.g., button clicks). The `BaseScreenletView` class contains a set of `userAction` methods that you can call in your View class to perform actions programmatically. For example, it's possible to trigger Add Bookmark Screenlet's `GetTitleAction` automatically whenever the user leaves the `URLTextField`. Since `BaseScreenletView` is the delegate for all `UITextField` objects by default, this is done in the View class (`AddBookmarkView_default`) by implementing the `textFieldDidEndEditing` method to call the `userAction` method with the action name:

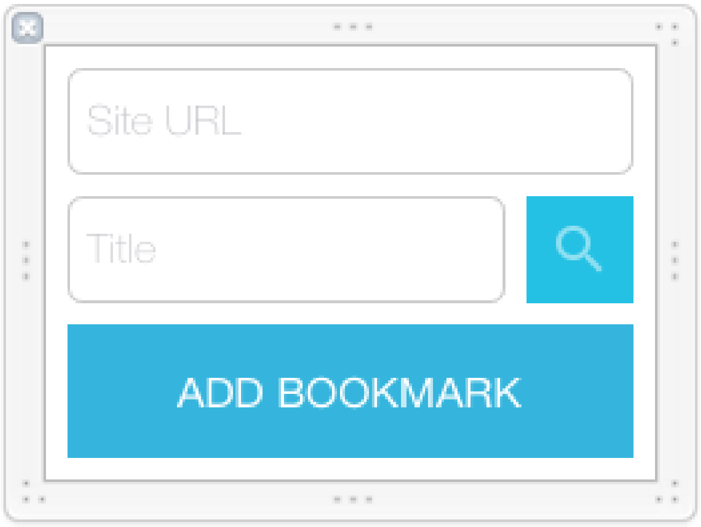


Figure 77.18: The sample Add Bookmark Screenlet's XIB file contains a new button next to the *Title* field for retrieving the URL's title.

```
func textFieldDidEndEditing(textField: UITextField) {
    if textField == URLTextField {
        userAction(name: AddBookmarkScreenlet.GetTitleAction)
    }
}
```

4. Update your View class or View Model protocol to account for the new action. For example, Add Bookmark Screenlet contains a View Model (`AddBookmarkViewModel`) so it can support multiple Themes. This View Model must allow the new action to set its `title` variable:

```
import UIKit

@objc protocol AddBookmarkViewModel {
    var URL: String? {get}
    var title: String? {set get}
}
```

5. If your Screenlet uses a View Model, conform your View class to the updated View Model. For example, the `title` variable in Add Bookmark Screenlet's View class (`AddBookmarkView_default`) must implement the setter from the previous step:

```
var title: String? {
    get {
        return titleTextField?.text
    }
    set {
        self.titleTextField?.text = newValue
    }
}
```

6. Create a new Interactor class for the new action. To do this, use the same steps detailed in the Screenlet creation tutorial. For example, here's the Interactor class for Add Bookmark Screenlet's Get Title action:

```
import UIKit
import LiferayScreens
```

```
public class GetWebTitleInteractor: Interactor {

    public var resultTitle: String?

    var url: String

    //MARK: Initializer

    public init(screenlet: BaseScreenlet, url: String) {
        self.url = url
        super.init(screenlet: screenlet)
    }

    override public func start() -> Bool {
        if let URL = NSURL(string: url) {

            // Use the NSURLSession class to retrieve the HTML
            NSURLSession.sharedSession().dataTaskWithURL(URL) {
                    (data, response, error) in

                if let errorValue = error {
                    self.callOnFailure(errorValue)
                }
                else {
                    if let data = data, html = NSString(data: data, encoding: NSUTF8StringEncoding) {
                        self.resultTitle = self.parseTitle(html)
                    }

                    self.callOnSuccess()
                }
            }.resume()

            return true
        }

        return false
    }

    // Parse the title from a webpage HTML
    private func parseTitle(html: NSString) -> String {
        let range1 = html.rangeOfString("<title>")
        let range2 = html.rangeOfString("</title>")

        let start = range1.location + range1.length

        return html.substringWithRange(NSMakeRange(start, range2.location - start))
    }

}
```

7. Update your Screenlet class's `createInteractor` method so it returns the correct Interactor for each action. For example, the `createInteractor` method in Add Bookmark Screenlet's Screenlet class (`AddBookmarkScreenlet`) contains a `switch` statement that returns an `AddBookmarkInteractor` or `GetWebTitleInteractor` instance when the Add Bookmark or Get Title action is called, respectively. Note that the `createAddBookmarkInteractor()` and `createGetTitleInteractor()` methods create these instances. Although you don't have to create your Interactor instances in separate methods, doing so leads to cleaner code:

```
...
override public func createInteractor(name name: String, sender: AnyObject?)
    -> Interactor? {
        switch name {
        case AddBookmarkScreenlet.AddBookmarkAction:
            return createAddBookmarkInteractor()
        case AddBookmarkScreenlet.GetTitleAction:
            return createGetTitleInteractor()
        default:
            return nil
        }
    }

private func createAddBookmarkInteractor() -> Interactor {
    let interactor = AddBookmarkInteractor(screenlet: self,
                                           folderId: folderId,
                                           title: viewModel.title!,
                                           url: viewModel.URL!)

    // Called when the Interactor finishes succesfully
    interactor.onSuccess = {
        let bookmarkName = interactor.resultBookmarkInfo!["name"] as! String
        print("Bookmark \"\(bookmarkName)\" saved!")
    }

    // Called when the Interactor finishes with an error
    interactor.onFailure = { _ in
        print("An error occurred saving the bookmark")
    }

    return interactor
}

private func createGetTitleInteractor() -> Interactor {
    let interactor = GetWebTitleInteractor(screenlet: self, url: viewModel.URL!)

    // Called when the Interactor finishes succesfully
    interactor.onSuccess = {
        let title = interactor.resultTitle
        self.viewModel.title = title
    }

    // Called when the Interactor finishes with an error
    interactor.onFailure = { _ in
        print("An error occurred retrieving the title")
    }

    return interactor
}
...
```

Great! Now you know how to support multiple actions in your Screenlets.

Related Topics

Creating iOS Screenlets
- Create and Use a Connector with Your Screenlet
- Creating iOS List Screenlets
- Architecture of Liferay Screens for iOS

77.9 Create and Use a Connector with Your Screenlet

In Liferay Screens, a Connector is a class that interacts asynchronously with local and remote data sources and Liferay instances. Recall that callbacks also make asynchronous service calls. So why bother with a Connector? Connectors provide a layer of abstraction by making your service call outside your Interactor.

For example, the Interactor in the Screenlet creation tutorial makes the server call and and processes its results via `LRCallback`. This Screenlet could instead make its server call in a separate Connector class, leaving the Interactor to instantiate the Connector and receive its results. Connectors also let you validate your Screenlet's data. For more information on Connectors, see the tutorial on the architecture of Liferay Screens for iOS.

This tutorial walks you through the steps required to create and use a Connector with your Screenlets, using the advanced version of the sample Add Bookmark Screenlet as an example. This Screenlet contains two actions:

1. Add Bookmark: Adds a bookmark to the Bookmarks portlet in a Liferay DXP installation. This tutorial shows you how to create and use a Connector for this action.

2. Get Title: Retrieves the title from a bookmark URL entered by the user. This tutorial shows you how to use a pre-existing Connector with this action.

Before proceeding, make sure you've read the Screenlet creation tutorial. First, you'll learn how to create your Connector.

Creating Connectors

When you create your Connector class, be sure to follow the naming convention specified in the best practices tutorial.

Use the following steps to implement your Connector class:

1. Create your Connector class by extending the `ServerConnector` class. For example, here's the class declaration for Add Bookmark Screenlet's Connector class, `AddBookmarkLiferayConnector`:

   ```
   public class AddBookmarkLiferayConnector: ServerConnector {
       ...
   }
   ```

2. Add the properties needed to call the Mobile SDK service, then create an initializer that sets those properties. For example, `AddBookmarkLiferayConnector` needs properties for the bookmark's folder ID, title, and URL. It also needs an initializer to set those properties:

   ```
   public let folderId: Int64
   public let title: String
   public let url: String

   public init(folderId: Int64, title: String, url: String) {
       self.folderId = folderId
       self.title = title
       self.url = url
       super.init()
   }
   ```

3. If you want to validate any of your Screenlet's properties, override the `validateData` method to implement validation for those properties. You can use the `ValidationError` class to encapsulate the errors. For example, the following `validateData` implementation in `AddBookmarkLiferayConnector` ensures that `folderId` is greater than `0`, and `title` and `url` contain values. This method also uses `ValidationError` to represent the error:

```
override public func validateData() -> ValidationError? {
    let error = super.validateData()

    if error == nil {

        if folderId <= 0 {
            return ValidationError("Undefined folderId")
        }

        if title.isEmpty {
            return ValidationError("Title cannot be empty")
        }

        if url.isEmpty {
            return ValidationError("URL cannot be empty")
        }
    }

    return error
}
```

4. Override the doRun method to call the Mobile SDK service you need to call. This method should retrieve the result from the service and store it in a public property. Also be sure to handle errors and empty results. For example, the following code defines the resultBookmarkInfo property for storing the service's results retrieved in the doRun method. Note that this method's service call is identical to the one in the AddBookmarkInteractor class's start method in the Screenlet creation tutorial. The doRun method, however, takes the additional step of saving the result to the resultBookmarkInfo property. Also note that this doRun method handles errors as NSError objects:

```
public var resultBookmarkInfo: [String:AnyObject]?

override public func doRun(session session: LRSession) {

    let service = LRBookmarksEntryService_v7(session: session)

    do {
        let result = try service.addEntryWithGroupId(LiferayServerContext.groupId,
                                                     folderId: folderId,
                                                     name: title,
                                                     url: url,
                                                     description: "Added from Liferay Screens",
                                                     serviceContext: nil)

        if let result = result as? [String: AnyObject] {
            resultBookmarkInfo = result
            lastError = nil
        }
        else {
            lastError = NSError.errorWithCause(.InvalidServerResponse)
            resultBookmarkInfo = nil
        }
    }
    catch let error as NSError {
        lastError = error
        resultBookmarkInfo = nil
    }

}
```

Well done! Now you know how to create a Connector class. To see the finished example AddBookmarkLiferayConnector class, click here.

Using Connectors

To use a Connector, your Interactor class must extend the ServerConnectorInteractor class or one of its following subclasses:

- ServerReadConnectorInteractor: Your Interactor class should extend this class when implementing an action that retrieves information from a server or data source.

- ServerWriteConnectorInteractor: Your Interactor class should extend this class when implementing an action that writes information to a server or data source.

When extending ServerConnectorInteractor or one of its subclasses, your Interactor class only needs to override the createConnector and completedConnector methods. These methods create a Connector instance and recover the Connector's result, respectively.

Follow these steps to use a Connector in your Interactor:

1. Set your Interactor class's superclass to ServerConnectorInteractor or one of its subclasses. You should also remove any code that conforms a callback protocol, if it exists. For example, Add Bookmark Screenlet's Interactor class (AddBookmarkInteractor) extends ServerWriteConnectorInteractor because it writes data to a Liferay DXP installation. At this point, your Interactor should contain only the properties and initializer that it requires:

```
public class AddBookmarkInteractor: ServerWriteConnectorInteractor {

    public let folderId: Int64
    public let title: String
    public let url: String

    public var resultBookmark: Bookmark?

    //MARK: Initializer

    public init(screenlet: BaseScreenlet, folderId: Int64, title: String, url: String) {
        self.folderId = folderId
        self.title = title
        self.url = url
        super.init(screenlet: screenlet)
    }
}
```

2. Override the createConnector method to return an instance of your Connector. For example, the createConnector method in AddBookmarkInteractor returns an AddBookmarkLiferayConnector instance created with the folderId, title, and url properties:

```
public override func createConnector() -> ServerConnector? {
    return AddBookmarkLiferayConnector(folderId: folderId, title: title, url: url)
}
```

3. Override the completedConnector method to get the result from the Connector and store it in the appropriate property. For example, the completedConnector method in AddBookmarkInteractor first casts its ServerConnector argument to AddBookmarkLiferayConnector. It then gets the Connector's resultBookmarkInfo property and sets it to the Interactor's resultBookmark property:

```
    override public func completedConnector(c: ServerConnector) {
        if let addCon = (c as? AddBookmarkLiferayConnector),
            bookmarkInfo = addCon.resultBookmarkInfo {
            self.resultBookmark = bookmarkInfo
        }
    }
```

That's it! To see the complete example `AddBookmarkInteractor`, click here.

If your Screenlet uses multiple Interactors, follow the same steps to use Connectors. Also, Screens provides the ready-to-use `HttpConnector` for interacting with non-Liferay URL's. To use this Connector, set your Interactor to use `HttpConnector`. For example, the Add Bookmark Screenlet action that retrieves a URL's title doesn't interact with a Liferay DXP installation; it retrieves the title directly from the URL. Because this action's Interactor class (`GetWebTitleInteractor`) retrieves data, it extends `ServerReadConnectorInteractor`. It also overrides the `createConnector` and `completedConnector` methods to use `HttpConnector`. Here's the complete `GetWebTitleInteractor`:

```
import UIKit
import LiferayScreens

public class GetWebTitleInteractor: ServerReadConnectorInteractor {

    public let url: String?

    // title from the webpage
    public var resultTitle: String?

    //MARK: Initializer

    public init(screenlet: BaseScreenlet, url: String) {
        self.url = url
        super.init(screenlet: screenlet)
    }

    //MARK: ServerConnectorInteractor

    public override func createConnector() -> ServerConnector? {
        if let url = url, URL = NSURL(string: url) {
            return HttpConnector(url: URL)
        }

        return nil
    }

    override public func completedConnector(c: ServerConnector) {
        if let httpCon = (c as? HttpConnector), data = httpCon.resultData,
            html = NSString(data: data, encoding: NSUTF8StringEncoding) {
            self.resultTitle = parseTitle(html)
        }
    }

    //MARK: Private methods

    // Parse the title from the webpage's HTML
    private func parseTitle(html: NSString) -> String {
        let range1 = html.rangeOfString("<title>")
        let range2 = html.rangeOfString("</title>")

        let start = range1.location + range1.length

        return html.substringWithRange(NSMakeRange(start, range2.location - start))
    }

}
```

Awesome! Now you know how to create and use Connectors in your Screenlets.

Related Topics

Creating iOS Screenlets
Adding Screenlet Actions
Architecture of Liferay Screens for iOS
Creating iOS List Screenlets

77.10 Add a Screenlet Delegate

Screenlet delegates let other classes respond to your Screenlet's actions. For example, Login Screenlet's delegate lets the app developer implement methods that respond to login success or failure. Note that the reference documentation for each Screenlet that comes with Liferay Screens lists the Screenlet's delegate methods.

You can also create a delegate for your own Screenlet. This tutorial walks you through the steps required to do this, using code from the advanced version of the sample Add Bookmark Screenlet as an example. All the example code in this tutorial resides in this Screenlet's Screenlet class. Also note that this sample Screenlet has two actions: adding a bookmark to a Liferay instance's Bookmarks portlet, and retrieving a bookmark's title from its URL. This tutorial only details creating a delegate for adding a bookmark.

Follow these steps to add a delegate to your Screenlet:

1. Define a delegate protocol that extends the `BaseScreenletDelegate` class. In this protocol, define success and failure methods so the conforming class can respond to the server call's success and failure, respectively. As parameters, these methods should take a Screenlet instance and the success or failure object. For example, Add Bookmark Screenlet's delegate protocol (`AddBookmarkScreenletDelegate`) defines the following success and failure methods:

```
@objc public protocol AddBookmarkScreenletDelegate: BaseScreenletDelegate {

    optional func screenlet(screenlet: AddBookmarkScreenlet,
                    onBookmarkAdded bookmark: [String: AnyObject])

    optional func screenlet(screenlet: AddBookmarkScreenlet,
                    onAddBookmarkError error: NSError)

}
```

   Both take an `AddBookmarkScreenlet` instance as their first argument. For their second argument, the success method contains the bookmark added to the server, and the failure method contains the `NSError` object. Note that in this example, the methods are optional. This means that the delegate class doesn't have to implement them.

2. In your Screenlet class, add a property for your delegate. This property should return `BaseScreenlet`'s `delegate` property as an instance of your delegate. For example, the `addBookmarkDelegate` property in `AddBookmarkScreenlet` returns the `self.delegate` property as `AddBookmarkScreenletDelegate`:

```
var addBookmarkDelegate: AddBookmarkScreenletDelegate? {
    return self.delegate as? AddBookmarkScreenletDelegate
}
```

3. Also in your Screenlet class, invoke the appropriate delegate methods in your Interactor's closures. For example, the `interactor.onSuccess` closure in `AddBookmarkScreenlet` calls the delegate method that responds when the Screenlet successfully adds a bookmark. The `interactor.onFailure` closure calls the delegate method that responds when the Screenlet fails to add a bookmark. Note that in this example, these closures are in the Screenlet class's Interactor method that adds a bookmark (`createAddBookmarkInteractor`). Be sure to call your delegate methods wherever the appropriate Interactor's closures are in your Screenlet class:

```
 private func createAddBookmarkInteractor() -> Interactor {
     let interactor = AddBookmarkInteractor(screenlet: self,
                                            folderId: folderId,
                                            title: viewModel.title!,
                                            url: viewModel.URL!)

     // Called when the Interactor finishes successfully
     interactor.onSuccess = {
         if let bookmark = interactor.resultBookmark {
             self.addBookmarkDelegate?.screenlet?(self, onBookmarkAdded: bookmark)
         }
     }

     // Called when the Interactor finishes with an error
     interactor.onFailure = { error in
         self.addBookmarkDelegate?.screenlet?(self, onAddBookmarkError: error)
     }

     return interactor
 }
```

Great! Now you know how to add a delegate to your Screenlets.

Related Topics

Creating iOS Screenlets

Adding Screenlet Actions

Creating iOS List Screenlets

Architecture of Liferay Screens for iOS

77.11 Using and Creating Progress Presenters

Many apps display a progress indicator while performing an operation. For example, you've likely seen the spinners in iOS apps that let you know the app is performing some kind of work. For more information, see the iOS Human Interface Guidelines article on Progress Indicators.

You can display these in Screenlets by using classes that conform the `ProgressPresenter` protocol. Liferay Screens includes two such classes:

- `MBProgressHUDPresenter`: Shows a message with a spinner in the middle of the screen. Liferay Screens shows this presenter by default.

- `NetworkActivityIndicatorPresenter`: Shows the progress using the iOS network activity indicator. This presenter doesn't support messages.

This tutorial shows you how to use and create progress presenters, using code from the advanced version of the sample Add Bookmark Screenlet as an example. First, you'll learn how to use progress presenters.

Using Progress Presenters

The BaseScreenletView class contains the default progress presenter functionality. To show a presenter other than the default MBProgressHUDPresenter, your View class must therefore override certain BaseScreenletView functionality. Follow these steps to do this:

1. In your View class, override the BaseScreenletView method createProgressPresenter to return an instance of the desired presenter. For example, to use NetworkActivityIndicatorPresenter in the sample Add Bookmark Screenlet, you must override the createProgressPresenter method in AddBookmarkView_default to return a NetworkActivityIndicatorPresenter instance:

```
override func createProgressPresenter() -> ProgressPresenter {
    return NetworkActivityIndicatorPresenter()
}
```

2. In your View class, override the BaseScreenletView property progressMessages to return the messages you want to use in the presenter. If the presenter doesn't display messages, then return an empty string. The progressMessages property should return the messages as [String : ProgressMessages], where String is the Screenlet's action name. ProgressMessages is a type alias representing a dictionary where the progress type is the key, and the actual message is the value. The three possible progress types correspond to the Screenlet action's status: Working, Failure, or Success. The progressMessages property therefore lets the presenter display the appropriate message for the Screenlet action's current status.

   For example, the following code overrides the progressMessages property in Add Bookmark Screenlet's View class (AddBookmarkView_default). For each Screenlet action (AddBookmarkAction and GetTitleAction), a message (NoProgressMessage) is assigned to the Screenlet operation's Working status. Since NoProgressMessage is an alias for an empty string, this tells the presenter to display no message when the Screenlet attempts to add a bookmark or get a title. Note, however, that the presenter still displays its progress indicator:

```
override var progressMessages: [String : ProgressMessages] {
    return [
        AddBookmarkScreenlet.AddBookmarkAction : [.Working: NoProgressMessage],
        AddBookmarkScreenlet.GetTitleAction : [.Working: NoProgressMessage],
    ]
}
```

   To display a message, replace NoProgressMessage with your message. For example, the following code defines separate messages for Working, Success, and Failure:

```
override var progressMessages: [String : ProgressMessages] {
    return [
        AddBookmarkScreenlet.AddBookmarkAction : [
            .Working: "Saving bookmark...",
            .Success: "Bookmark saved!",
            .Failure: "An error occurred saving the bookmark"
        ],
        AddBookmarkScreenlet.GetTitleAction : [
            .Working: "Getting site title...",
            .Failure: "An error occurred retrieving the title"
        ],
    ]
}
```

Great! Now you know how to use progress presenters. Next, you'll learn how to create your own.

Creating Progress Presenters

Creating your own progress presenter isn't as complicated as you might think. Recall that a presenter in Liferay Screens is a class that conforms the ProgressPresenter protocol. You can create your presenter by conforming this protocol from scratch, or by extending one of Screens's existing presenters that already conform this protocol (MBProgressHUDPresenter or NetworkActivityIndicatorPresenter). In most cases, extending MBProgressHUDPresenter is sufficient.

For example, Add Bookmark Screenlet's AddBookmarkProgressPresenter extends MBProgressHUDPresenter to display a different progress indicator for the Screenlet's get title action. Use the following steps to create a progress presenter that extends from an existing presenter. As an example, these steps extend MBProgressHUDPresenter to add a progress indicator for the get title button:

1. In your View's XIB file, add the activity indicator you want to use. For example, the XIB file in Add Bookmark Screenlet contains an iOS UIActivityIndicatorView over the get title button:

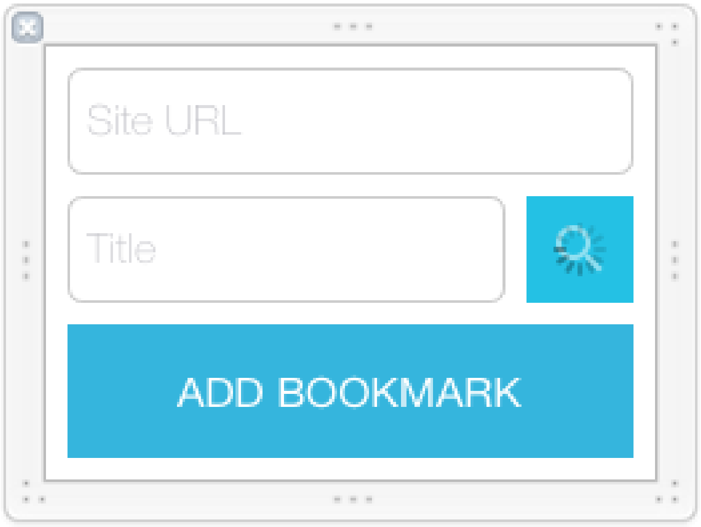


Figure 77.19: The updated Add Bookmark Screenlet's XIB file contains a new activity indicator over the get title button.

2. In your View class, create an outlet for the XIB's new activity indicator. For example, Add Bookmark Screenlet's View class (AddBookmarkView_default) contains an @IBOutlet for the UIActivityIndicatorView from the XIB:

```
@IBOutlet weak var activityIndicatorView: UIActivityIndicatorView?
```

Now you must create your presenter class. You'll do this here by extending an existing presenter class. Use the following steps to do this:

1. Extend the existing presenter class you want to base your presenter on. Your presenter class must contain properties for your presenter's activity indicator and any other UI components. It must also contain an initializer that sets these properties. For example, AddBookmarkProgressPresenter extends MBProgressHUDPresenter and contains properties for the get title button and UIActivityIndicatorView. Its initializer sets these properties:

```
public class AddBookmarkProgressPresenter: MBProgressHUDPresenter {

    let button: UIButton?

    let activityIndicator: UIActivityIndicatorView?

    public init(button: UIButton?, activityIndicator: UIActivityIndicatorView?) {
        self.button = button
        self.activityIndicator = activityIndicator
        super.init()
    }
    ...
```

2. Implement your presenter's behavior by overriding the appropriate methods from the presenter class that you're extending. For example, `AddBookmarkProgressPresenter` overrides `MBProgressHUDPresenter`'s `showHUDInView` and `hideHUDFromView` methods. The overridden `showHUDInView` method hides the button and starts animating the activity indicator. The overridden `hideHUDFromView` method stops this animation and restores the button:

```
public override func showHUDInView(view: UIView, message: String?,
    forInteractor interactor: Interactor) {
        guard interactor is GetWebTitleInteractor else {
            return super.showHUDInView(view, message: message,
                forInteractor: interactor)
        }

        button?.hidden = true
        activityIndicator?.startAnimating()
}

public override func hideHUDFromView(view: UIView?, message: String?,
    forInteractor interactor: Interactor, withError error: NSError?) {
        guard interactor is GetWebTitleInteractor else {
            return super.hideHUDFromView(view, message: message,
                forInteractor: interactor, withError: error)
        }

        activityIndicator?.stopAnimating()
        button?.hidden = false
    }

}
```

Great, that's it! Now you can use your presenter the same way you would any other.

Related Topics

Creating iOS Screenlets
    Creating iOS List Screenlets
    Architecture of Liferay Screens for iOS

77.12 Creating and Using Your Screenlet's Model Class

Liferay Screens typically receives entities from a Liferay instance as `[String:AnyObject]`, where `String` is the entity's attribute and `AnyObject` is the attribute's value. Although you can use these dictionary objects throughout your Screenlet, it's often easier to create a *model class* that converts each into an object that represents the corresponding Liferay entity. This is especially convenient for complex entities composed of many attribute-value pairs. Note that Liferay Screens already provides several model classes for you. Click here to see them.

At this point, you might be saying, "Ugh! I have complex entities and Screens doesn't provide a model class for them! I'm just going to give up and watch football." Fret not! Although we'd never come between you and football, creating and using your own model class is straightforward.

Using the advanced version of the sample Add Bookmark Screenlet as an example, this tutorial shows you how to create and use a model class in your Screenlet. First, you'll create your model class.

Creating Your Model Class

Your model class must contain all the code necessary to transform each `[String:AnyObject]` that comes back from the server into a model object that represents the corresponding Liferay entity. This includes a constant for holding each `[String:AnyObject]`, and initializer that sets this constant, and a public property for each attribute value.

For example, the sample Add Bookmark Screenlet adds a bookmark to a Liferay instance's Bookmarks portlet. Since the Mobile SDK service method that adds the bookmark also returns it as `[String:AnyObject]`, the Screenlet can convert it into an object that represents bookmarks. It does so with its `Bookmark` model class. This class extends `NSObject` and sets the `[String:AnyObject]` to the `attributes` constant via the initializer. This class also defines computed properties that return the attribute values for each bookmark's name and URL:

```
@objc public class Bookmark : NSObject {

    public let attributes: [String:AnyObject]

    public var name: String {
        return attributes["name"] as! String
    }

    override public var description: String {
        return attributes["description"] as! String
    }

    public var url: String {
        return attributes["url"] as! String
    }

    public init(attributes: [String:AnyObject]) {
        self.attributes = attributes
    }

}
```

Next, you'll put your model class to work.

Using Your Model Class

Now that your model class exists, you can use model objects anywhere your Screenlet handles results. Exactly where depends on what Screenlet components your Screenlet uses. For example, Add Bookmark Screenlet's Connector, Interactor, delegate, and Screenlet class all handle the Screenlet's results. The steps here therefore show you how to use model objects in each of these components. Note, however, that your Screenlet may lack a Connector or delegate: these components are optional. Variations on these steps are therefore noted where applicable.

1. Create model objects where the `[String: AnyObject]` results originate. For example, the `[String: AnyObject]` results in Add Bookmark Screenlet originate in the Connector. Therefore, this is

where the Screenlet creates Bookmark objects. The following code in the Screenlet's Connector (AddBookmarkLiferayConnector) does this. The if statement following the service call casts the results to [String: AnyObject], calls the Bookmark initializer with those results, and stores the resulting Bookmark object to the public resultBookmarkInfo variable. Note that this is only the code that makes the service call and creates the Bookmark object. Click here to see the complete AddBookmarkLiferayConnector class:

```
...
// Public property for the results
public var resultBookmarkInfo: Bookmark?

...

override public func doRun(session session: LRSession) {
    let service = LRBookmarksEntryService_v7(session: session)

    do {
        let result = try service.addEntryWithGroupId(LiferayServerContext.groupId,
                                                     folderId: folderId,
                                                     name: title,
                                                     url: url,
                                                     description: "Added from Liferay Screens",
                                                     serviceContext: nil)

        // Creates Bookmark objects from the service call's results
        if let result = result as? [String: AnyObject] {
            resultBookmarkInfo = Bookmark(attributes: result)
            lastError = nil
        }
        ...
    }
    ...
}
```

If your Screenlet doesn't have Connector, then your Interactor's start method makes your server call and handles its results. Otherwise, the process for creating a Bookmark object from [String: AnyObject] is the same.

2. Handle your model objects in your Screenlet's Interactor. The Interactor processes your Screenlet's results, so it must also handle your model objects. If your Screenlet doesn't use a Connector, then you already did this in your Interactor's start method as mentioned at the end of the previous step. If your Screenlet uses a Connector, however, then this happens in your Interactor's completedConnector method. For example, the completedConnector method in Add Bookmark Screenlet's Interactor (AddBookmarkInteractor) retrieves the Bookmark via the Connector's resultBookmarkInfo variable. This method then assigns the Bookmark to the Interactor's public resultBookmark variable. Note that this is only the code that handles Bookmark objects. Click here to see the complete AddBookmarkInteractor class:

```
...
// Public property for the results
public var resultBookmark: Bookmark?

...

// The completedConnector method gets the results from the Connector
override public func completedConnector(c: ServerConnector) {
    if let addCon = (c as? AddBookmarkLiferayConnector),
        bookmark = addCon.resultBookmarkInfo {
            self.resultBookmark = bookmark
        }
}
```

3. If your Screenlet uses a delegate, your delegate protocol must account for your model objects. Skip this step if you don't have a delegate. For example, Add Bookmark Screenlet's delegate (`AddBookmarkScreenletDelegate`) must communicate `Bookmark` objects. The delegate's first method does this via its second argument:

```
@objc public protocol AddBookmarkScreenletDelegate: BaseScreenletDelegate {

    optional func screenlet(screenlet: AddBookmarkScreenlet,
                    onBookmarkAdded bookmark: Bookmark)

    optional func screenlet(screenlet: AddBookmarkScreenlet,
                    onAddBookmarkError error: NSError)

}
```

4. Get the model object from the Interactor in your Screenlet class's `interactor.onSuccess` closure. You can then use the model object however you wish. For example, the `interactor.onSuccess` closure in Add Bookmark Screenlet's Screenlet class (`AddBookmarkScreenlet`) retrieves the `Bookmark` from the Interactor's `resultBookmark` property. It then handles the `Bookmark` via the delegate. Note that in this example, the closure is in the Screenlet class's Interactor method that adds a bookmark (`createAddBookmarkInteractor`). Be sure to get your model object wherever the `interactor.onSuccess` closure is in your Screenlet class. Click here to see the complete `AddBookmarkScreenlet`:

```
...
private func createAddBookmarkInteractor() -> Interactor {
    let interactor = AddBookmarkInteractor(screenlet: self,
                                       folderId: folderId,
                                       title: viewModel.title!,
                                       url: viewModel.URL!)

    // Called when the Interactor finishes successfully
    interactor.onSuccess = {
        if let bookmark = interactor.resultBookmark {
            self.addBookmarkDelegate?.screenlet?(self, onBookmarkAdded: bookmark)
        }
    }

    // Called when the Interactor finishes with an error
    interactor.onFailure = { error in
        self.addBookmarkDelegate?.screenlet?(self, onAddBookmarkError: error)
    }

    return interactor
}
...
```

Awesome! Now you know how to create and use a model class in your Screenlet.

Related Topics

Creating iOS Screenlets

Adding Screenlet Actions

Creating iOS List Screenlets

Architecture of Liferay Screens for iOS

77.13 Creating iOS List Screenlets

It's very common for mobile apps to display lists. Liferay Screens lets you display asset lists and DDL lists in your iOS app by using Asset List Screenlet and DDL List Screenlet, respectively. Screens also includes list Screenlets for displaying lists of other Liferay entities like web content articles, images, and more. The Screenlet reference documentation lists all the Screenlets included with Liferay Screens. If there's not a list Screenlet for the entity you want to display in a list, you must create your own list Screenlet. A list Screenlet can display any entity from a Liferay instance. For example, you can create a list Screenlet that displays standard Liferay entities like `User`, or custom entities from custom Liferay apps.

This tutorial uses code from the sample Bookmark List Screenlet to show you how to create your own list Screenlet. This Screenlet displays a list of bookmarks from Liferay's Bookmarks portlet. You can find this Screenlet's complete code here in GitHub.

Note that because this tutorial focuses on creating a list Screenlet, it doesn't explain general Screenlet concepts and components. Before beginning, you should therefore read the following tutorials:

- Creating iOS Screenlets
- Supporting Multiple Themes in Your Screenlet
- Create and Use a Connector with Your Screenlet
- Add a Screenlet Delegate
- Creating and Using Your Screenlet's Model Class

This tutorial uses the following steps to show you how to create a list Screenlet:

1. Creating the Model class
2. Creating the Theme
3. Creating the Connector
4. Creating the Interactor
5. Creating the Delegate
6. Creating the Screenlet class

First though, you should understand how pagination works with list Screenlets.

Pagination

To ensure that users can scroll smoothly through large lists of items, list Screenlets support fluent pagination. Support for this is built into the list Screenlet framework. You'll see this as you construct your list Screenlet.

Now you're ready to begin!

Creating the Model Class

Recall that a model class transforms each `[String:AnyObject]` entity Screens receives into a model object that represents the corresponding Liferay entity. For instructions on creating your model class, see the tutorial Creating and Using Your Screenlet's Model Class. The example model class in that tutorial is identical to Bookmark List Screenlet's.

Next, you'll create your Screenlet's Theme.

Creating the Theme

Recall that each Screenlet needs a Theme to serve as its UI. A Theme needs an XIB file to define the UI's components and layout. Since a list Screenlet displays a list of entities, its XIB file must contain a Table View. Use these steps to create your Theme's XIB file:

1. In Xcode, create a new XIB file and name it according to these naming conventions. For example, the XIB for Bookmark List Screenlet's Default Theme is `BookmarkListView_default.xib`.

2. In Interface Builder, drag and drop a View from the Object Library to the canvas. Then add a Table View to the View.

3. Resize the Table View to take up the entire View, and set the constraints the Table View needs to maintain this size dynamically. This ensures that the list fills the Screenlet's UI regardless of the iOS device's size or orientation.

For example, Bookmark List Screenlet's XIB file uses a `UITableView` inside a parent View to show the list of bookmarks.

Now you'll create your Theme's View class. Every Theme needs a View class that controls its behavior. Since the XIB file uses a `UITableView` to show a list of guestbooks, your View class must extend the `BaseListTableView` class. Liferay Screens provides this class to serve as the base class for list Screenlets' View classes. Since `BaseListTableView` provides most of the required functionality, extending it lets you focus on the parts of your View class that are unique to your Screenlet. You must also configure the XIB file to use your View class.

Follow these steps to create your Screenlet's View class and configure the XIB file to use it:

1. Create your Theme's View class, and name it according to these naming conventions. Since the XIB uses `UITableView`, your View class must extend `BaseListTableView`. For example, this is Bookmark List Screenlet's View class declaration:

   ```
   public class BookmarkListView_default: BaseListTableView {...
   ```

2. Now you must override the View class methods that fill the table cells' contents. There are two methods for this, depending on the cell type:

   - **Normal cells:** the cells that show the entities. These cells typically use `UILabel`, `UIImage`, or another UI component to show the entity. Override the `doFillLoadedCell` method to fill these cells. For example, Bookmark List Screenlet's View class overrides `doFillLoadedCell` to set each cell's `textLabel` to a bookmark's name:

     ```
     override public func doFillLoadedCell(row row: Int, cell: UITableViewCell,
         object: AnyObject) {

             let bookmark = object as! Bookmark

             cell.textLabel?.text = bookmark.name
     }
     ```

   - **Progress cell:** the cell at the bottom of the list that indicates the list is loading the next page of items. Override the `doFillInProgressCell` method to fill this cell. For example, Bookmark List Screenlet's View class overrides this method to set the cell's `textLabel` to the string `"Loading..."`:

```
    override public func doFillInProgressCell(row row: Int, cell: UITableViewCell) {
        cell.textLabel?.text = "Loading..."
    }
```

3. Return to the Theme's XIB in Interface Builder, and set the View class as the the parent View's custom class. For example, if you were doing this for Bookmark List Screenlet you'd select the Table View's parent View, click the Identity inspector, and enter `BookmarkListView_default` as the custom class.

4. With the Theme's XIB still open in Interface Builder, set the parent View's `tableView` outlet to the Table View. To do this, select the parent View and click the Connections inspector. In the Outlets section, drag and drop from the `tableView`'s circle icon (it turns into a plus icon on mouseover) to the Table View in the XIB. The new outlet then appears in the Connections inspector.

That's it! Now that your Theme is finished, you can create the Connector.

Creating the Connector

Recall that Connectors make a server call. To support pagination, a List Screenlet's Connector class must extend the `PaginationLiferayConnector` class. The Connector class must also contain any properties it needs to make the server call, and an initializer that sets them. To support pagination, the initializer must also contain the following arguments, which you'll pass to the superclass initializer:

- `startRow`: The number representing the page's first row.
- `endRow`: The number representing the page's last row.
- `computeRowCount`: Whether to call the Connector's `doAddRowCountServiceCall` method (you'll learn about this method shortly).

For example, Bookmark List Screenlet must retrieve bookmarks from a Bookmarks portlet folder in a specific site. The Screenlet's Connector class must therefore have properties for the `groupId` (site ID) and `folderId` (Bookmarks folder ID), and an initializer that sets them. The initializer also calls the superclass initializer with the `startRow`, `endRow`, and `computeRowCount` arguments:

```
import UIKit
import LiferayScreens

public class BookmarkListPageLiferayConnector: PaginationLiferayConnector {

    public let groupId: Int64
    public let folderId: Int64

    //MARK: Initializer

    public init(startRow: Int, endRow: Int, computeRowCount: Bool, groupId: Int64,
        folderId: Int64) {

            self.groupId = groupId
            self.folderId = folderId

            super.init(startRow: startRow, endRow: endRow, computeRowCount: computeRowCount)
    }
    ...
```

Next, if you want to validate any of your Screenlet's properties, override the `validateData` method as described in the tutorial on creating Connectors. Note that Bookmark List Screenlet only needs to validate the `folderId`:

```
override public func validateData() -> ValidationError? {
    let error = super.validateData()

    if error == nil {
        if folderId <= 0 {
            return ValidationError("Undefined folderId")
        }
    }

    return error
}
```

Lastly, you must override the following two methods in the Connector class:

- `doAddPageRowsServiceCall`: calls the Liferay Mobile SDK service method that retrieves a page of entities. The `doAddPageRowsServiceCall` method's `startRow` and `endRow` arguments specify the page's first and last entities, respectively. Make the service call as you would in any Screenlet. For example, the `doAddPageRowsServiceCall` method in `BookmarkListPageLiferayConnector` calls the service's `getEntriesWithGroupId` method to retrieve a page of bookmarks from the folder specified by `folderId`:

  ```
  public override func doAddPageRowsServiceCall(session session: LRBatchSession,
      startRow: Int, endRow: Int, obc: LRJSONObjectWrapper?) {
          let service = LRBookmarksEntryService_v7(session: session)

      do {
          try service.getEntriesWithGroupId(groupId,
                                            folderId: folderId,
                                            start: Int32(startRow),
                                            end: Int32(endRow))
      }
      catch  {
          // ignore error: the service method returns nil because
          // the request is sent later, in batch
      }
  }
  ```

  Note that you don't need to do anything in the catch statement because the request is sent later, in batch. The session type `LRBatchSession` handles this for you. You'll receive the request's results elsewhere, once the request completes.

- `doAddRowCountServiceCall`: calls the Liferay Mobile SDK service method that retrieves the total number of entities. This supports pagination. Make the service call as you would in any Screenlet. For example, the `doAddRowCountServiceCall` in `BookmarkListPageLiferayConnector` calls the service's `getEntriesCountWithGroupId` method to retrieve the total number of bookmarks in the folder specified by `folderId`:

  ```
  override public func doAddRowCountServiceCall(session session: LRBatchSession) {
      let service = LRBookmarksEntryService_v7(session: session)

      do {
          try service.getEntriesCountWithGroupId(groupId, folderId: folderId)
      }
      catch  {
          // ignore error: the service method returns nil because
          // the request is sent later, in batch
      }
  }
  ```

Now that you have your Connector class, you're ready to create the Interactor.

Creating the Interactor

Recall that Interactors implement your Screenlet's actions. In list Screenlets, loading entities is usually the only action a user can take. The Interactor class of a list Screenlet that implements fluent pagination must extend the BaseListPageLoadInteractor class. Your Interactor class must also contain any properties the Screenlet needs, and an initializer that sets them. This initializer also needs arguments for the following properties, which it passes to the superclass initializer:

- screenlet: A BaseListScreenlet reference. This ensures the Interactor always has a Screenlet reference.
- page: The page number to retrieve.
- computeRowCount: Whether to call the Connector's doAddRowCountServiceCall method.

For example, Bookmark List Screenlet's Interactor class contains the same groupId and folderId properties as the Connector, and an initializer that sets them. This initializer also passes the screenlet, page, and computeRowCount arguments to the superclass initializer:

```
public class BookmarkListPageLoadInteractor : BaseListPageLoadInteractor {

    private let groupId: Int64
    private let folderId: Int64

    init(screenlet: BaseListScreenlet,
        page: Int,
        computeRowCount: Bool,
        groupId: Int64,
        folderId: Int64) {

            self.groupId = (groupId ≠ 0) ? groupId : LiferayServerContext.groupId
            self.folderId = folderId

            super.init(screenlet: screenlet, page: page, computeRowCount: computeRowCount)
    }
    ...
```

The Interactor class must also initiate the server request by instantiating the Connector, and convert the results into model objects. Override the createListPageConnector method to create and return an instance of your Connector. This method must first get a reference to the Screenlet via the screenlet property. When calling the Connector's initializer, use screenlet.firstRowForPage to convert the page number (page) to the page's start and end indices. You must also pass the initializer any other properties it needs, like groupId. For example, BookmarkListPageLoadInteractor's createListPageConnector method does this to create a BookmarkListPageLiferayConnector instance:

```
public override func createListPageConnector() -> PaginationLiferayConnector {
    let screenlet = self.screenlet as! BaseListScreenlet

    return BookmarkListPageLiferayConnector(
        startRow: screenlet.firstRowForPage(self.page),
        endRow: screenlet.firstRowForPage(self.page + 1),
        computeRowCount: self.computeRowCount,
        groupId: groupId,
        folderId: folderId)
}
```

Next, override the convertResult method to convert each [String:AnyObject] result into a model object. The Screenlet calls this method once for each entity retrieved from the server. For example, BookmarkListPageLoadInteractor's convertResult method converts the [String:AnyObject] result into a Bookmark object:

```
override public func convertResult(_ serverResult: [String:AnyObject]) -> AnyObject {
    return Bookmark(attributes: serverResult)
}
```

You may also want to support offline mode in your Interactor. To do so, the Interactor must override the cacheKey method to return a cache key unique to your Screenlet. For example, BookmarkListPageLoadInteractor's cacheKey method returns a cache key that includes the Screenlet's groupId and folderId settings:

```
override public func cacheKey(_ op: PaginationLiferayConnector) -> String {
    return "\(groupId)-\(folderId)"
}
```

Great! Next, you'll create your Screenlet's delegate.

Creating the Delegate

Recall that a delegate is required if you want other classes to respond to your Screenlet's actions. Create your delegate by following the first step in the tutorial on adding a Screenlet delegate. A list Screenlet's delegate must also define a method for responding to a list item selection. For example, Bookmark List Screenlet's delegate needs the following methods:

- screenlet(_:onBookmarkListResponse:): Returns the Bookmark results when the server call succeeds.
- screenlet(_:onBookmarkListError:): Returns the NSError object when the server call fails.
- screenlet(_:onBookmarkSelected:): Returns the Bookmark when a user selects it in the list.

The BookmarkListScreenletDelegate protocol defines these methods:

```
@objc public protocol BookmarkListScreenletDelegate : BaseScreenletDelegate {

    optional func screenlet(screenlet: BookmarkListScreenlet,
                            onBookmarkListResponse bookmarks: [Bookmark])

    optional func screenlet(screenlet: BookmarkListScreenlet,
                            onBookmarkListError error: NSError)

    optional func screenlet(screenlet: BookmarkListScreenlet,
                            onBookmarkSelected bookmark: Bookmark)

}
```

Nice work! Next, you'll create the Screenlet class.

Creating the Screenlet Class

Now that your list Screenlet's other components exist, you can create the Screenlet class. A list Screenlet's Screenlet class must extend the BaseListScreenlet class and define the configurable properties the Screenlet needs. You should define these as IBInspectable properties. If you want to support offline mode, you should also add an offlinePolicy property.

For example, Bookmark List Screenlet's Screenlet class contains the IBInspectable properties groupId, folderId, and offlinePolicy:

```
public class BookmarkListScreenlet: BaseListScreenlet {

    @IBInspectable public var groupId: Int64 = 0
    @IBInspectable public var folderId: Int64 = 0
    @IBInspectable public var offlinePolicy: String? = CacheStrategyType.RemoteFirst.rawValue

    ...
```

Next, override the createPageLoadInteractor method to create and return the Interactor. If your Screenlet supports offline mode, you should also use offlinePolicy to pass a CacheStrategyType object to the Interactor. For example, the createPageLoadInteractor method in BookmarkListScreenlet creates and returns a BookmarkListPageLoadInteractor instance. This method also sets the Interactor's cacheStrategy property to a CacheStrategyType object created with offlinePolicy:

```
override public func createPageLoadInteractor(
    page page: Int,
    computeRowCount: Bool) -> BaseListPageLoadInteractor {

    let interactor = BookmarkListPageLoadInteractor(screenlet: self,
                                                     page: page,
                                                     computeRowCount: computeRowCount,
                                                     groupId: self.groupId,
                                                     folderId: self.folderId)

    interactor.cacheStrategy = CacheStrategyType(rawValue: self.offlinePolicy ?? "") ?? .RemoteFirst

    return interactor
}
```

Now get a reference to your delegate. The BaseScreenlet class, which BaseListScreenlet extends, already defines the delegate property for the delegate object. Every list Screenlet therefore has this property, and any app developer using the Screenlet can access it. To avoid casting this property to your delegate every time you use it, add a computed property to your Screenlet class that does so. For example, the following bookmarkListDelegate property in BookmarkListScreenlet casts the delegate property to BookmarkListScreenletDelegate:

```
public var bookmarkListDelegate: BookmarkListScreenletDelegate? {
    return delegate as? BookmarkListScreenletDelegate
}
```

Next, override the BaseListScreenlet methods that handle the Screenlet's events. Because these events correspond to the events your delegate methods handle, you'll call your delegate methods in these BaseListScreenlet methods:

- onLoadPageResult: Called when the Screenlet loads a page successfully. Override this method to call your delegate's screenlet(_:onBookmarkListResponse:) method. For example, here's BookmarkListScreenlet's onLoadPageResult method:

  ```
  override public func onLoadPageResult(page page: Int, rows: [AnyObject], rowCount: Int) {
      super.onLoadPageResult(page: page, rows: rows, rowCount: rowCount)

      bookmarkListDelegate?.screenlet?(screenlet: self, onBookmarkListResponse: rows as! [Bookmark])
  }
  ```

- onLoadPageError: Called when the Screenlet fails to load a page. Override this method to call your delegate's screenlet(_:onBookmarkListError:) method. For example, here's BookmarkListScreenlet's onLoadPageError method:

  ```
  override public func onLoadPageError(page page: Int, error: NSError) {
      super.onLoadPageError(page: page, error: error)

      bookmarkListDelegate?.screenlet?(screenlet: self, onBookmarkListError: error)
  }
  ```

- `onSelectedRow`: Called when an item is selected in the list. Override this method to call your delegate's `screenlet(_:onBookmarkSelected:)` method. For example, here's `BookmarkListScreenlet`'s `onSelectedRow` method:

```
override public func onSelectedRow(_ row: AnyObject) {
    bookmarkListDelegate?.screenlet?(screenlet: self, onBookmarkSelected: row as! Bookmark)
}
```

Awesome! You're done! Your list Screenlet, like any other Screenlet, is a ready-to-use component that you can add to your storyboard. You can even package it using the same steps you use to package a Theme, and then contribute it to the Liferay Screens project or distribute it with CocoaPods.

Related Topics

Creating iOS Screenlets
  Supporting Multiple Themes in Your Screenlet
  Create and Use a Connector with Your Screenlet
  Add a Screenlet Delegate
  Creating and Using Your Screenlet's Model Class
  Using Custom Cells with List Screenlets
  Sorting Your List Screenlet
  Creating Complex Lists in Your List Screenlet
  Architecture of Liferay Screens for iOS

77.14 Using Custom Cells with List Screenlets

In most list Screenlets, including those that come with Liferay Screens, the Default Theme uses the default cells in iOS's `UITableView` to show the list. The Theme creation steps in the list Screenlet creation tutorial also instruct you to use these cells. You can, however, use custom cells to tailor the list to your needs. To do this, you must create an extended Theme from a Theme that uses `UITableView`'s default cells. This usually means extending a list Screenlet's Default theme. This tutorial shows you how to create such an extended Theme that contains a custom cell for your list Screenlet. As an example, this tutorial uses code from the sample Bookmark List Screenlet's Custom Theme. You can refer to this Theme's finished code here in GitHub at any time.

Note that besides creating your custom cell, this tutorial follows the same basic steps as the Theme creation tutorial for creating an extended Theme. For example, you must still determine where to create your Theme, and create your Theme's XIB and View class.

First, you'll create your Theme's custom cell.

Creating Your Custom Cell

Once you decide where to create your Theme, you can get started. First, create your custom cell's XIB file and its companion class. Name them according to the naming conventions in the best practices tutorial. After defining your cell's UI in the XIB, create as many outlets and actions as you need in its companion class. Also be sure to assign this class as the XIB's custom class in Interface Builder. Note that if you want to use different layouts for different rows, you must create an XIB file and companion class for each.

For example, the following screenshot shows the XIB file `BookmarkCell_default-custom.xib` for Bookmark List Screenlet's custom cell. This cell must show a bookmark's name and URL, so it contains two labels.

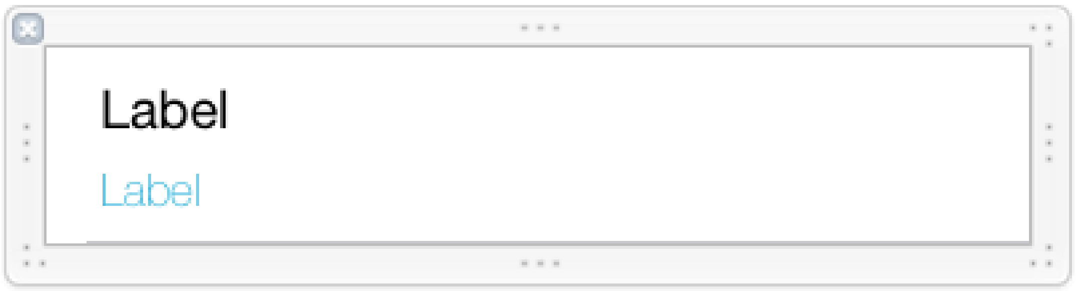


Figure 77.20: The XIB file for Bookmark List Screenlet's custom cell.

This XIB's custom class, `BookmarkCell_default_custom`, contains an outlet for each label. The `bookmark` variable also contains a `didSet` observer that sets the bookmark's name and URL to the respective label:

```
import UIKit

class BookmarkCell_default_custom: UITableViewCell {

    @IBOutlet weak var nameLabel: UILabel?
    @IBOutlet weak var urlLabel: UILabel?

    var bookmark: Bookmark? {
        didSet {
            nameLabel?.text = bookmark?.name
            urlLabel?.text = bookmark?.url
        }
    }

}
```

Great! Now you have your custom cell. Next, you'll create the rest of your Theme.

Creating Your Theme's XIB and View Class

Now you're ready to create your Theme's XIB file and View class. Create your XIB by copying the parent Theme's XIB and making any changes you need. You may not need to make any changes besides the file name and custom class name. For example, the custom cell is the only difference between Bookmark List Screenlet's Custom and Default Themes. These Themes' XIB files (`BookmarkListView_default-custom.xib` and `BookmarkListView_default.xib`) are therefore identical besides their name and custom class; the size and position of their UI components are the same.

Now create your View class by extending the parent Theme's View class. You should also add a string constant to serve as the cell ID. In a moment, you'll use this constant to register your custom cell. For example, the View class in Bookmark List Screenlet's Custom Theme (`BookmarkListView_default_custom`) extends the Default Theme's View class (`BookmarkListView_default`) and defines the string constant `BookmarkCellId`:

```
public class BookmarkListView_default_custom: BookmarkListView_default {

    let BookmarkCellId = "bookmarkCell"
    …
```

Next, override the `doRegisterCellNibs` method to register your custom cell. In this method, create a `UINib` instance for your cell and then register it with the `UITableView` instance (`tableView`) inherited from the `BaseListTableView` class. When registering the nib file, you must use the string constant you created earlier as the `forCellReuseIdentifier`. For example, here's the `doRegisterCellNibs` method in `BookmarkListView_default-custom`:

```
  public override func doRegisterCellNibs() {
      let nib = UINib(nibName: "BookmarkCell_default-custom", bundle: NSBundle.mainBundle())

      tableView?.registerNib(nib, forCellReuseIdentifier: BookmarkCellId)
}
```

Also in your View class, override the `doGetCellId` method to return the cell ID for each row. All you need to do in this method is return the string constant you created earlier. For example, the `doGetCellId` method in `BookmarkListView_default-custom` returns the `BookmarkCellId` constant:

```
override public func doGetCellId(row row: Int, object: AnyObject?) -> String {
    return BookmarkCellId
}
```

Now override the `doFillLoadedCell` method to fill the cell with data. Note that this method isn't called for in-progress cells; it's only called for cells that display data. Also note that this method's `object` argument contains the data as `AnyObject`. You must cast this to your desired type and then set it to the appropriate cell variable. For example, the `doFillLoadedCell` method in `BookmarkListView_default-custom` casts the `object` argument to `Bookmark` and then sets it to the cell's `bookmark` variable:

```
override public func doFillLoadedCell(row row: Int, cell: UITableViewCell, object:AnyObject) {
    if let bookmarkCell = cell as? BookmarkCell_default_custom, bookmark = object as? Bookmark {
        bookmarkCell.bookmark = bookmark
    }
}
```

The typical iOS `UITableViewDelegate` protocol and `UITableViewDataSource` protocol methods are also available in your View class. You can override any of them if you need to (check first to make sure they're not already overridden). For example, `BookmarkListView_default-custom` implements the following method to use a different cell height for each row:

```
public func tableView(tableView: UITableView, heightForRowAtIndexPath indexPath: NSIndexPath) -> CGFloat {
    return 80
}
```

When you finish, set your View class as your XIB file's custom class.

Awesome! You're done! Now you know how to implement your own custom cells for use in list Screenlets.

Related Topics

Creating iOS List Screenlets

Creating iOS Themes

Sorting Your List Screenlet

Creating Complex Lists in Your List Screenlet

iOS Best Practices

77.15 Sorting Your List Screenlet

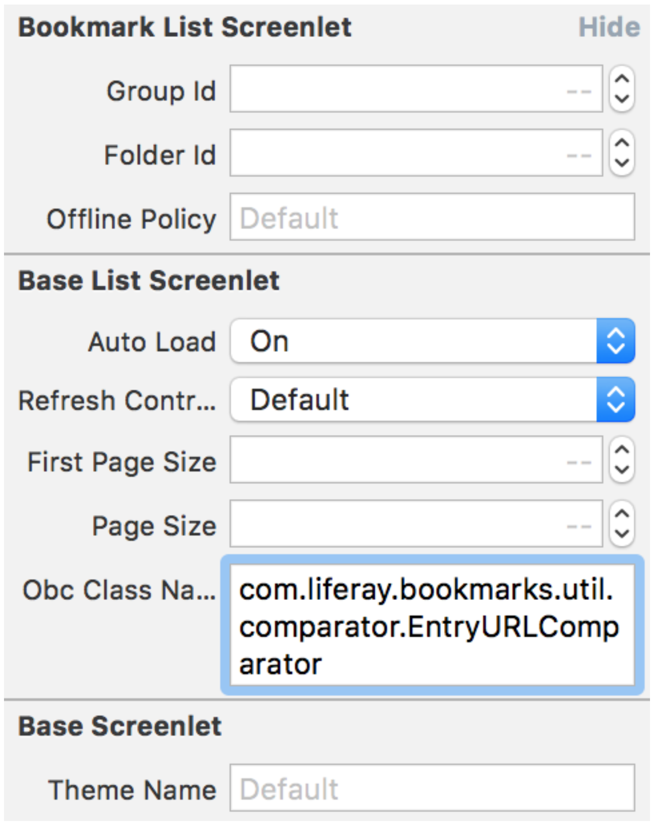
To sort your list Screenlet, you must point it to a *comparator class* in your portal. A comparator class implements the logic that sorts your entities. You can create your own comparator class or use those that already exist in your portal. Once your list is sorted, you can split it into sections. This tutorial shows you how to sort your list Screenlet with a comparator and create sections for your sorted list.

Note: To create a new comparator, you must create a class that extends the portal's `OrderByComparator` class with your entity as a type argument. Then you must override the methods that implement the sort. For example, the portal's `EntryURLComparator` class sorts bookmarks in Liferay's Bookmarks portlet by URL.

First, you'll learn how to use a comparator to sort your list Screenlet.

Using a Comparator

To use a comparator, you must set the list Screenlet's `obcClassName` property to the comparator's fully qualified class name. Do this in Interface Builder when inserting the Screenlet in an app, just as you would set any other Screenlet property. For example, to set the sample Bookmark List Screenlet to sort its list of bookmarks by URL, you must set *Obc Class Name* to *com.liferay.bookmarks.util.comparator.EntryURLComparator* in Interface Builder:



Bookmark List Screenlet	Hide
Group Id	--
Folder Id	--
Offline Policy	Default
Base List Screenlet	
Auto Load	On
Refresh Contr...	Default
First Page Size	--
Page Size	--
Obc Class Na...	com.liferay.bookmarks.util.comparator.EntryURLComparator
Base Screenlet	
Theme Name	Default

Figure 77.21: To use a comparator, set the *Obc Class Name* property in Interface Builder to the comparator's fully qualified class name.

That's it! Note that although all list Screenlets inherit the `obcClassName` property from the `BaseListScreenlet` class, the list Screenlet must also make its service call with this property. See the Screenlet reference documentation to see which list Screenlets included with Liferay Screens support the `obcClassName` property. Also, Liferay DXP's comparator classes can change between versions. If you're using one of these comparators, make sure you specify the one that matches your Liferay DXP version.

Create Sections for Your List

Dividing lists into sections that contain like elements is common in iOS apps. To do this in list Screenlets, first use a comparator to sort the list by the criteria you'll use to create the sections. Then override the

BookmarkListPageLoadInteractor class's sectionForRowObject method in your list Screenlet's Interactor. This method is called for each item in the list and should return the information necessary to place the item in a section. For example, the sample Bookmark List Screenlet's Interactor overrides the sectionForRowObject method to group results by hostname:

```
public override func sectionForRowObject(object: AnyObject) -> String? {
    guard let bookmark = object as? Bookmark else {
        return nil
    }

    let host = NSURL(string: bookmark.url)?.host?.lowercaseString

    return host?.stringByReplacingOccurrencesOfString("www.", withString: "")
}
```

Note that this only produces predictable results when Bookmark List Screenlet is sorted by EntryURLComparator as detailed in the preceding section.

And that's all there is to it! Now you know how to sort and section your list Screenlet's list.

Related Topics

Creating iOS List Screenlets
    Using Custom Cells with List Screenlets
    Creating Complex Lists in Your List Screenlet
    iOS Best Practices

77.16 Creating Complex Lists in Your List Screenlet

Most list Screenlets' Themes use iOS's UITableView to display simple lists. Although UITableView is great for this, it's not so great for complex lists like grids or stacks. To create complex lists, you should use iOS's UICollectionView in your list Screenlet's Theme.

This tutorial shows you how to create such a Theme, using the sample Bookmark List Screenlet's Collection Theme as an example. First, you'll create the list's cell.

Creating the Cell

You'll create your list's cell with the same sequence of steps used to create any list Screenlet's cell. Note, however, that how you perform these steps is a bit different:

1. Define your cell's UI in a new XIB file. Because this cell is part of a Theme that uses UICollectionView, you can shape it however you want. For example, here's the BookmarkCell_default-collection.xib file for the cell in Bookmark List Screenlet's Collection Theme. It's a simple square that displays the bookmark's URL and the URL's first letter.

2. Create your XIB file's class by extending UICollectionViewCell. Create as many outlets and actions as you need for your UI components and write the logic required for your cell's UI to function. For example, BookmarkCell_default_collection is the XIB file's class in Bookmark List Screenlet's Custom Theme. This class extends UICollectionViewCell and contains outlets for the URL (urlLabel) and the URL's first letter (centerLabel). The bookmark variable's didSet observer sets the bookmark's name and URL to the respective label. Also note that the overridden prepareForReuse method resets the labels for reuse:

Figure 77.22: The XIB file for the cell in Bookmark List Screenlet's custom View.

```
import UIKit
import LiferayScreens

public class BookmarkCell_default_collection: UICollectionViewCell {

    //MARK: Outlets

    @IBOutlet weak var centerLabel: UILabel?
    @IBOutlet weak var urlLabel: UILabel?

    //MARK: Public properties

    public var bookmark: Bookmark? {
        didSet {
            if let bookmark = bookmark, url = NSURL(string: bookmark.url),
                firstLetter = url.host?.remove("www.").characters.first {

                    self.centerLabel?.text = String(firstLetter).uppercaseString
                    self.urlLabel?.text = bookmark.url.remove("http://").remove("https://").remove("www.")
            }
        }
    }

    //MARK: UICollectionViewCell

    override public func prepareForReuse() {
        super.prepareForReuse()

        centerLabel?.text = "..."
        urlLabel?.text = "..."
    }
}
```

Now that your cell exists, you can create the rest of your Theme.

Creating the Theme's XIB and View Class

You'll create the rest of your Theme with the same sequence of steps used to create any list Screenlet's Theme. Like creating the cell, how you perform these steps is a bit different because your Theme uses `UICollectionView` instead of `UITableView`.

First, define your Theme's UI in a new XIB file. Add a `UICollectionView` instead of a `UITableView` to this file. For example, the `BookmarkListView_default-collection.xib` file for Bookmark List Screenlet's Custom Theme contains a collection view.

Next, create the View class. Instead of extending `BaseListTableView`, this class must extend Screens's `BaseListCollectionView` class. The `BaseListCollectionView` class implements most of the code necessary to use `UICollectionView` in your Screenlet. By extending it, you can focus on the code unique to your

Screenlet. Your View class should also contain a string constant to serve as the cell ID. You'll use this constant when you register your cell. For example, the View class in Bookmark List Screenlet's Collection Theme (`BookmarkListView_default_collection`) extends `BaseListCollectionView` and defines the string constant `BookmarkCellId`:

```
public class BookmarkListView_default_collection : BaseListCollectionView {

    let BookmarkCellId = "bookmarkCell"
    ...
```

In Interface Builder, set this new class as the XIB's Custom Class.

Next, override the `doRegisterCellNibs` method to register the cell you created in the previous section. In this method, create a `UINib` instance for your cell and then register it with the `UICollectionView` instance (`collectionView`) inherited from `BaseListCollectionView`. When registering the nib file, you must use the string constant you created earlier as the `forCellReuseIdentifier`. For example, here's the `doRegisterCellNibs` method in `BookmarkListView_default_collection`:

```
public override func doRegisterCellNibs() {
    let cellNib = UINib(nibName: "BookmarkCell_default-collection", bundle: nil)
    collectionView?.registerNib(cellNib, forCellWithReuseIdentifier: BookmarkCellId)
}
```

Also in your View class, override the `doGetCellId` method to return the ID you registered the cell with. For example, the `doGetCellId` method in `BookmarkListView_default_collection` returns the string constant `BookmarkCellId`:

```
public override func doGetCellId(indexPath indexPath: NSIndexPath, object: AnyObject?) -> String {
    return BookmarkCellId
}
```

Next, override the `doFillLoadedCell` method to fill the cell with data. This method's `object` argument contains the data as `AnyObject`. You must cast this to your desired type and then set it to the appropriate cell variable. For example, the `doFillLoadedCell` method in `BookmarkListView_default_collection` casts the `object` argument to `Bookmark` and then sets it to the cell's `bookmark` variable:

```
public override func doFillLoadedCell(
        indexPath indexPath: NSIndexPath,
        cell: UICollectionViewCell,
        object: AnyObject) {

    if let cell = cell as? BookmarkCell_default_collection, bookmark = object as? Bookmark {
        cell.bookmark = bookmark
    }
}
```

Next, you'll create the layout.

Creating the Layout

The layout object is a key part of `UICollectionView`. This object controls the position of the UI elements, their size, and more. To customize the layout object, override the `doCreateLayout` method in your View class. For example, the `doCreateLayout` method in Bookmark List Screenlet's View class (`BookmarkListView_default_collection`) returns a `UICollectionViewFlowLayout` for the layout object. This is a basic layout that gives you a simple way to customize things like item size, spacing between items, scroll direction, and more:

```
public override func doCreateLayout() -> UICollectionViewLayout {
    let layout = UICollectionViewFlowLayout()
    layout.itemSize = CGSize(width: 110, height: 110)
    layout.minimumLineSpacing = 10
    layout.minimumInteritemSpacing = 10

    return layout
}
```

Great! You're done! You can now use your new Theme the same way you would any other.

If you want to package your Theme to contribute it to the Liferay Screens project or distribute it with CocoaPods, see the tutorial on packaging Themes.

Related Topics

Creating iOS List Screenlets

Creating iOS Themes

Sorting Your List Screenlet

Using Custom Cells with List Screenlets

iOS Best Practices

77.17 Creating iOS Themes

By creating your own Themes, you can customize your mobile app's design and functionality. You can create them from scratch or use an existing Theme as a foundation. Themes include a View class for implementing Screenlet behavior and an XIB file for specifying the UI. The three Liferay Screens Theme types support different levels of customization and parent Theme inheritance. Here's what each Theme type offers:

Child Theme: presents the same UI components as its parent Theme, but lets you change their appearance and position.

Extended Theme: inherits its parent Theme's functionality and appearance, but lets you add to and modify both.

Full Theme: provides a complete standalone View for a Screenlet. A full Theme is ideal for implementing functionality and appearance completely different from a Screenlet's current Theme.

This tutorial explains how to create all three types. To understand Theme concepts and components, you might want to examine the architecture of Liferay Screens for iOS. The tutorial Creating iOS Screenlets can help you create any Screenlet classes your Theme requires. Now get ready to create some Themes!

Determining Your Theme's Location

After determining the type of Theme to create, you need to decide where to create it. If you want to reuse or redistribute it, you should create it in an empty Cocoa Touch Framework project in Xcode. The packaging tutorial explains how to package and publish with CocoaPods. If you're not planning to reuse or redistribute your Theme, you can create it directly inside your app project.

The rest of this tutorial explains how to create each type of Theme. First, you'll learn how to create a Child Theme.

Creating a Child Theme

In a Child Theme, you leverage a parent Theme's behavior and UI components, but you can modify the appearance and position of the UI components. Note that you can't add or remove any components and the parent Theme must be a Full Theme. The Child Theme presents visual changes with its own XIB file and inherits the parent's View class.

For example, the Child Theme in Figure 1 presents the same UI components as the Login Screenlet's Default Theme, but enlarges them for viewing on devices with larger screens.

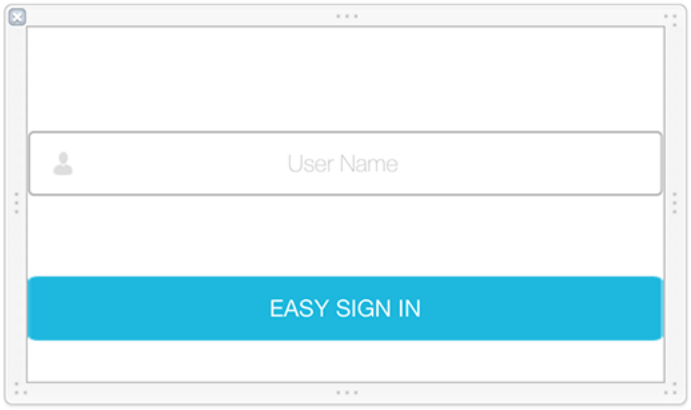


Figure 77.23: The UI components are enlarged in the example Child Theme's XIB file.

You can follow these steps to create a Child Theme:

1. In Xcode, create a new XIB file that's named after the Screenlet's View class and your Theme. By convention, an XIB file for a Screenlet with a View class named *FooScreenletView* and a Theme named *BarTheme* must be named `FooScreenletView_barTheme.xib`. You can use content from the parent Theme's XIB file as a foundation for your new XIB file. In your new XIB, you can change the UI components' visual properties (e.g., their position and size). You mustn't change, however, the XIB file's custom class, outlet connection, or `restorationIdentifier`–these must match those of the parent's XIB file.

```
The XIB file name serves as the Theme's Xcode name. For example, the Theme
in Figure 1 inherits from the Login Screenlet's Default Theme, which uses
the View class `LoginView_default`. The new child Theme is named *Large*
because it's purpose is to enlarge the Screenlet's UI components. In Xcode,
it's assigned the Theme Name *large*. The XIB file is named
`LoginView_large.xib`, after the Login Screenlet's View class and the
Theme's Xcode name.
```

You can optionally package your Theme and/or start using it. Fantastic! Next, you'll learn how to create an Extended Theme.

Extended Theme

An Extended Theme inherits another Theme's UI components and behavior, but lets you add to or alter it by extending the parent Theme's View class and creating a new XIB file. An Extended Theme's parent must be a Full Theme. The Flat7 Theme is an Extended Theme.

These steps explain how to create an Extended Theme:

1. In Xcode, create a new XIB file named after the Screenlet's View class and your Theme. By convention, an XIB file for a Screenlet with a View class named *FooScreenletView* and a Theme named *BarTheme* must be named `FooScreenletView_barTheme.xib`. You can use the XIB file of your parent Theme as a template. Build your UI changes in your new XIB file with Interface Builder.

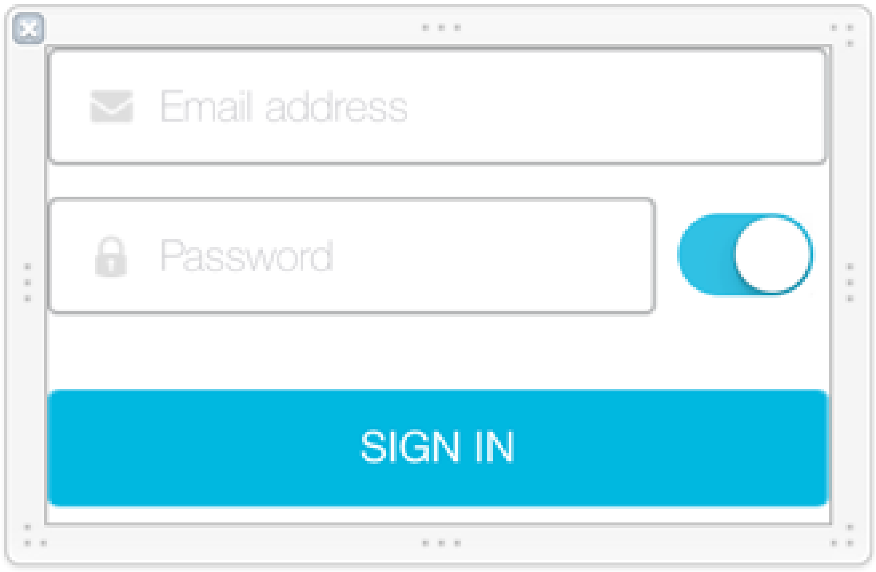


Figure 77.24: This example Extended Theme's XIB file extends the Login Portlet's UI and behavior with a switch that lets the user show or hide the password field value.

2. Create a new View class that extends the parent Theme's View class. You should name this class after the XIB file you just created. You can add or override functionality of the parent Theme's View class.

3. Set your new View class as the custom class for your Theme's XIB file. If you added `@IBOutlet` or `@IBAction` actions, bind them to your class.

Well done! You can optionally package your Theme and/or start using it. Now you know how to create and use an Extended Theme. Next, you'll learn how to create a Full Theme.

Full Theme

A Full Theme implements unique behavior and appearance for a Screenlet, without using a parent Theme. Its View class must inherit Screens's `BaseScreenletView` and conform to the Screenlet's View Model protocol. It must also specify a new UI in an XIB file. As you create a Full Theme, you can refer to the tutorial Creating iOS Screenlets to learn how to create these classes.

Follow these steps to create a Full Theme:

1. Create a new XIB file and use Interface Builder to build your UI. By convention, an XIB file for a Screenlet with a View class named *FooScreenletView* and a Theme named *BarTheme* must be named `FooScreenletView_barTheme.xib`. You can use the XIB file from the Screenlet's default Theme as a template.

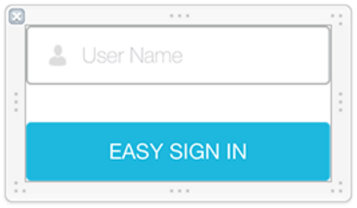


Figure 77.25: This Full Theme for the Login Screenlet, includes a text field for entering the user name, uses the UDID for the password, and adds a *Sign In* button with the same `restorationIdentifier` as the Default Theme.

2. Create a new View class for your Theme named after the XIB file you just created. As a template, you can use the View class of your Screenlet's Default Theme. Your new View class must inherit `BaseScreenletView` and conform to the Screenlet's `*ScreenletViewModel` protocol, implementing the corresponding getters and setters. It should also add all the `@IBOutlet` properties or `@IBAction` methods you need to bind your UI components.

3. Set your Theme's new View class as your XIB file's custom class and bind any `@IBOutlet` and `@IBAction` actions to your class.

Super! You can optionally package your Theme and/or start using it. Now you know how to create a Full Theme. Note that a Full Theme can serve as a parent to a Child and Extended Theme.

You've mastered Themes!

Related Topics

Packaging iOS Themes

Using Themes in iOS Screenlets

Architecture of Liferay Screens for iOS

Creating iOS Screenlets

77.18 Packaging iOS Themes

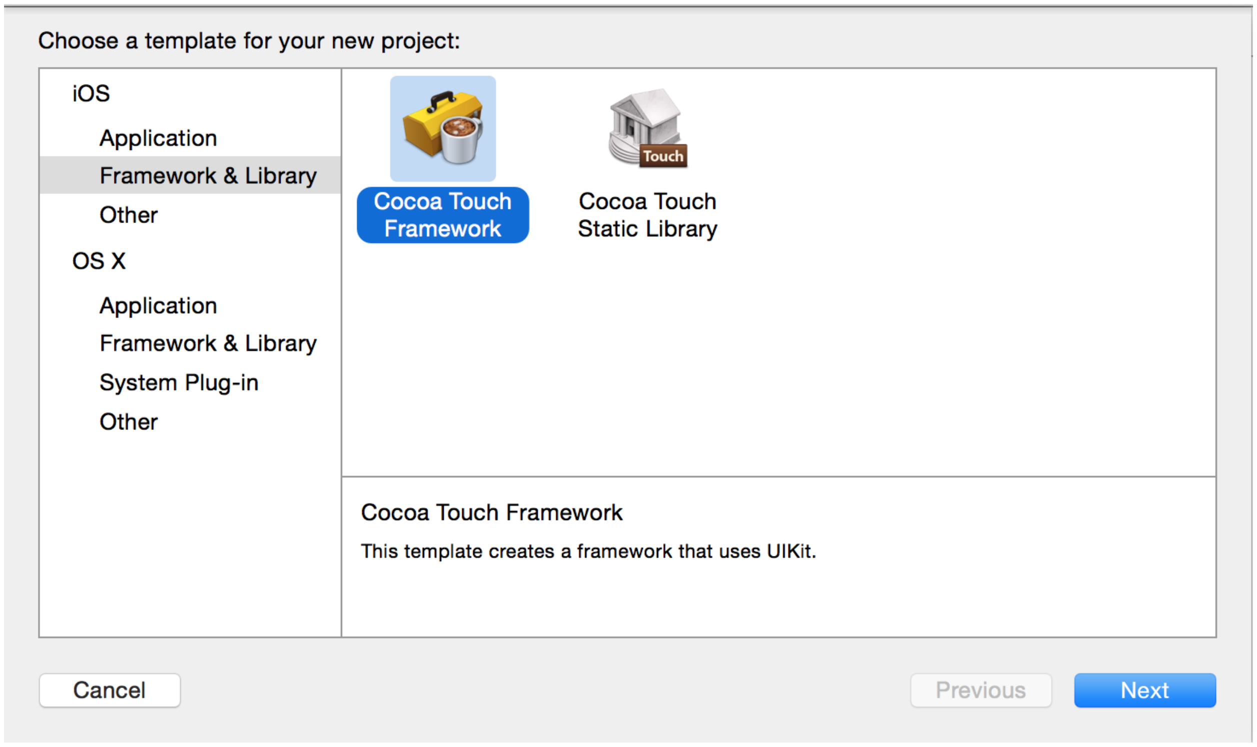
Once you've created a Theme, you can package it up to install and use with its Screenlet. Your Theme is a code library that you can package using CocoaPods.

Follow the steps below to package your Theme for use with CocoaPods. (Note that it's important that you use the same names and identifiers described in these steps):

1. Create an empty *Cocoa Touch Framework* Xcode project.

2. Name your project `LiferayScreensThemeName`, replacing `Name` with your Theme's name. You can specify any name, but it's a best practice to use your Theme's Xcode name, capitalizing its first letter. The entire name becomes the Theme's CocoaPods name.

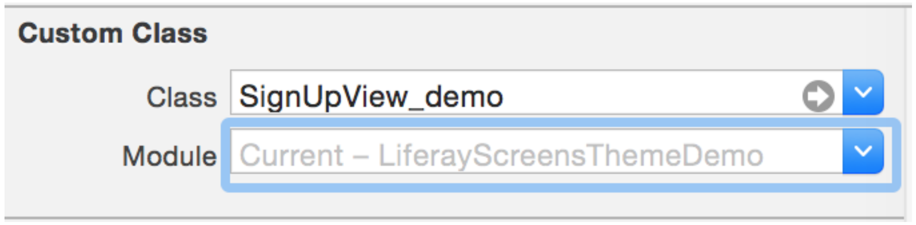
3. Configure Liferay Screens for CocoaPods, using the steps described in Preparing iOS Projects for Liferay Screens.



Cocoa Touch Framework

This template creates a framework that uses UIKit.

Figure 77.26: Choose *Cocoa Touch Framework* when creating a project for your Theme.



Custom Class

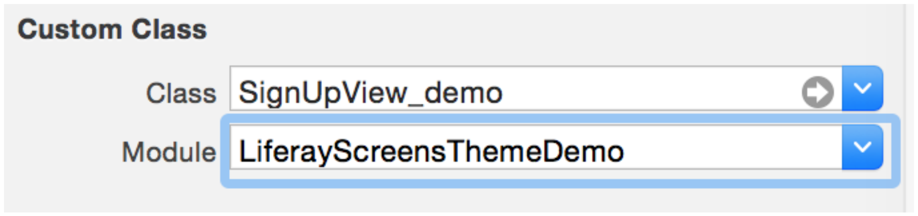
Class: SignUpView_demo

Module: Current – LiferayScreensThemeDemo

Figure 77.27: This XIB file's custom class's module is NOT specified.

4. Prepare your Theme's classes and resources by making sure your classes compile successfully in Xcode and by explicitly specifying a valid module for the custom class–the grayed-out *Current* default value only suggests a module.

   In the following screenshot, the setting for the custom class is correct:



Custom Class

Class: SignUpView_demo

Module: LiferayScreensThemeDemo

Figure 77.28: The XIB file is bound to the custom class name, with the specified module.

5. In your project's root folder, add a file named `LiferayScreensTheme-Name.podspec` (change `Name` to your

Theme's CocoaPods name–the value you used to replace `Name` in step 2). Note: you must start your the `.podspec` file's name and the project's name with `LiferayScreens`.

Add the following content to the file:

```
Pod::Spec.new do |s|
    s.name = 'LiferayScreensThemeName'
    s.version = '1.0'
    s.summary = 'Your theme description'
    s.source = {
        :git => 'https://your_repository_url.git',
        :tag => 'v1.0'
    }

    s.platform = :ios, '8.0'
    s.requires_arc = true

    s.source_files = 'Your/Relative/Folder/**/*.{h,m,swift}'
    s.resources = 'Your/Relative/Folder/**/*.{xib,png,plist,lproj}'

    s.dependency 'LiferayScreens'
end
```

Make the following substitutions in the `.podspec` file:

- Replace `Name` in `LiferayScreensThemeName`, with your Theme's CocoaPods name–the value you used to replace `Name` in step 2.
- Replace `your_repository_url` with your repository's URL.
- Replace `Your/Relative/Folder/` with the path to your source and resource files.

6. Commit your changes and push your project's branch to your Git repository.

Your Theme is now available for other developers to pull from your Git repository. You can, alternatively, publish your Theme as a public Pod. For instructions, see the chapter *Deploying a library* in the official CocoaPods guide.

Developers can now use your Theme by adding the following line to their app's `Podfile`; they must, of course, change `Name` to the Theme's CocoaPods name and `your_repository_url` to your repository's URL:

```
pod 'LiferayScreensThemeName', :git => 'https://your_repository_url.git'
```

Nice work! Now you know how to package and distribute Screenlet Themes with CocoaPods.

Related Topics

Using Themes in iOS Screenlets

Architecture of Liferay Screens for iOS

Creating iOS Themes

Creating iOS Screenlets

Preparing Android Projects for Liferay Screens

77.19 Accessing the Liferay Session in iOS

A session is a conversation state between the client and server. It typically consists of multiple requests and responses between the two. To facilitate this communication, the session must have the server IP address, and a user's login credentials. Liferay Screens uses a Liferay Session to access and query the JSON web services provided by Liferay Portal. When you log in using a Liferay Session, the portal returns the user's information (name, email, user ID, etc...). Screens stores this information and the active Liferay Session in Screens's `SessionContext` class.

The `SessionContext` class is very powerful and lets you use Screens in many different scenarios. For example, you can use `SessionContext` to request information with the JSON WS API provided by Liferay, or with the Liferay Mobile SDK. You can also use `SessionContext` to create anonymous sessions, or log in a user without showing Login Screenlet.

This tutorial explains some common `SessionContext` use cases, and also describes the class's most important methods.

Getting the current session

The current session is established after the user successfully logs in with Login Screenlet. Use `SessionContext.currentContext` to retrieve the session. Note this will return `nil` if the user didn't sign in with Login Screenlet. You can also use the `SessionContext` property `isLoggedIn` to determine if a session exists. This returns `false` if there's no current session.

Creating a Liferay Session

When working with Liferay Screens, you may wish to call the remote JSON web services provided by the Liferay Mobile SDK. Every operation with the Liferay Mobile SDK needs a Liferay Session to provide the server address, user credentials, and any other required parameters. Login Screenlet creates a session when a user successfully logs in. You can retrieve this session with the `SessionContext` method `createRequestSession()`. Typically, you call this method through the `currentContext` object. For example:

```
SessionContext.currentContext?.createRequestSession()
```

You can then use the session to make the Mobile SDK service call. If you need to check first to see if a user has logged in, you can use the `SessionContext` property `isLoggedIn`.

Great! Now you know how to retrieve an existing session in your app. But what if you're not using Login Screenlet? There won't be an existing session to retrieve. No sweat! You can still use `SessionContext` to create one manually. The next section shows you how to do this.

Creating a Session Manually

If you don't use Login Screenlet, then `SessionContext` doesn't have a session for you to retrieve. In this case, you must create one manually. You can do this with the `SessionContext` method `loginWithBasic`. The method takes a username, password, and user attributes as parameters, and creates a session with those credentials. The following code uses `loginWithBasic` to create a session:

```
Session session = SessionContext.loginWithBasic(username: USERNAME, password: PASSWORD, userAttributes: [:]);
```

For the `userAttributes` parameter, you must provide some attributes associated with the logged in user, such as their `userId`. For a complete list of attributes, see the user model interface.

Super! Now you know how to create a session manually. The next section shows you how to implement auto-login, and save or restore a session.

Implementing Auto-login and Saving or Restoring a Session

Although Login Screenlet is awesome, your users may not want to enter their credentials every time they open your app. It's very common for apps to only require a single login. To implement this in your app, see this video.

In short, you need to set `saveCredentials` to `true` in Login Screenlet. The next login then uses the saved credentials. To make sure this also works when the app restarts, you must retrieve the stored credentials by using the `SessionContext` method `loadStoredCredentials`. The following Swift code shows a typical implementation of this:

```
if SessionContext.loadStoredCredentials() {
    // user auto-logged in
    // consider doing a relogin here (see next section)
}
else {
    // send user to login screen with the login screenlet
}
```

Awesome! Now you know how to implement auto-login in your Liferay Screens apps. For more information on available `SessionContext` methods, see the Methods section at the end of this tutorial. Next, you'll learn how to implement relogin for cases where a user's credentials change on the server while they're logged in.

Implementing Relogin

A session, whether created via Login Screenlet or auto-login, contains basic user data that verifies the user in the Liferay instance. If that data changes in the server, then your session is outdated, which may cause your app to behave inconsistently. Also, if a user is deleted, deactivated, or otherwise changes their credentials in the server, the auto-login feature won't deny access because it doesn't perform server transactions: it only retrieves an existing session from local storage. This isn't an optimal situation!

For such scenarios, you can use the relogin feature. This feature is implemented in a method that determines if the current session is still valid. If the session is still valid, the user's data is updated with the most recent data from the server. If the session isn't valid, the user is logged out and must then log in again to create a new session.

To this feature, call the `SessionContext.currentContext` method `relogin`:

```
SessionContext.currentContext?.relogin(closure)
```

Note that this operation is done asynchronously in a background thread. The `closure` argument is a function that eventually receives the new user attributes. In case of error, the closure is called with `nil` attributes and the user is logged out of the session. The typical Swift code for a full relogin is as follows. Note a trailing closure is used:

```
SessionContext.currentContext?.relogin { userAttributes in
    if userAttributes == nil {
        // couldn't retrieve the user attributes: user invalidated or password changed?
    }
    else {
        // full re-login made. Everything is updated
    }
}
```

Great! Now you know how to implement relogin in your app. You've also seen how handy `SessionContext` can be. It can do even more! The next section lists some additional `SessionContext` methods, and some more detail on the ones used in this tutorial.

Methods

Method	Return Type	Explanation
`logout()`	`void`	Clears the stored user attributes and session.
`relogin(closure)`	`void`	Refreshes user data from the server. This recreates `currentContext` if successful, or calls `logout()` on failure. When the server data is received, the closure is called with received user's attributes. If an error occurs, the closure is called with `nil`.
`loginWithBasic(username, password, userAttributes)`	`LRSession`	Creates a Liferay Session using the default server, and the supplied username, password, and user information.
`loginWithOAuth(authentication, userAttributes)`	`LRSession`	Creates a Liferay Session using the default server and the supplied OAuth tokens. This is intended to be used together with the Liferay iOS OAuth library.
`createRequestSession()`	`LRSession`	Creates a Liferay Session based on the current session's server and user credentials. This Liferay Session is intended to be used for only a single request (don't reuse it).
`createEphemeralBasicSession(username, password)`	`LRSession`	Creates a Liferay Session based on the provided username and password. Note that this session isn't stored anywhere. This is the method used to create a session for anonymous access. Anonymous access is used by the Sign Up and Forgot Password Screenlets.
`userAttribute(key: String)`	`AnyObject`	Returns a `User` object with the server attributes of the logged-in user. This includes the user's email, user ID, name, and portrait ID.
`storeCredentials()`	`Bool`	Stores the current session.
`removeStoredCredentials()`	`Bool`	Clears the session and user information from storage.
`loadStoredCredentials()`	`Bool`	Loads the session and user information from storage. They're then used, respectively, as the current session and user.

Properties

Property	Type	Explanation
`currentContext`	`SessionContext`	The current session established through Login Screenlet, or the `loginWithBasic` or `loginWithOAuth` methods.
`isLoggedIn`	`Bool`	Returns `true` if `SessionContext` contains a Liferay Session.
`basicAuthUsername`	`String`	The username used to establish the current session (if any).
`basicAuthPassword`	`String`	The password used to establish the current session (if any).
`userId`	`Number`	The user identifier used to establish the current session (if any).

For more information, see the `SessionContext` source code in GitHub.

Related Topics

Login Screenlet for iOS
Using Screenlets in iOS Apps

77.20 Adding Custom Interactors to iOS Screenlets

Interactors are Screenlet components that implement server communication for a specific use case. For example, the Login Screenlet's interactor calls the Liferay Mobile SDK service that authenticates a user to the portal. Similarly, the interactor for the Add Bookmark Screenlet calls the Liferay Mobile SDK service that adds a bookmark to the Bookmarks portlet.

That's all fine and well, but what if you want to customize a Screenlet's server call? What if you want to use a different back-end with a Screenlet? No problem! You can implement a custom interactor for the Screenlet. You can plug in a different interactor that makes its server call by using custom logic or network code. To do this, you must implement the current interactor's interface and then pass it to the Screenlet you want to override. You should do this inside your app's code.

In this tutorial, you'll see an example interactor that overrides the Login Screenlet to always log in the same user, without a password.

Implementing a Custom Interactor

1. Implement your custom interactor. You must inherit `ServerConnectorInteractor`, as shown here:

```
class LoginCustomInteractor: ServerConnectorInteractor {

    override func createConnector() -> ServerConnector? {

        ...

        return connector
    }

}
```

2. Implement the optional protocol that receives a `customInteractorForAction`, and return your own interactor:

```
func screenlet(screenlet: BaseScreenlet,
        customInteractorForAction: String,
        withSender: AnyObject?) -> Interactor? {

    return LoginCustomInteractor()
}
```

Great! Now you know how to implement custom interactors for iOS Screenlets.

Related Topics

Architecture of Liferay Screens for iOS

Creating iOS Screenlets

77.21 Rendering Web Content in Your iOS App

Liferay Screens provides several ways to render web content in your app. For historical reasons, web content articles are `JournalArticle` entities in Liferay. Using Web Content Display Screenlet is a simple and powerful way to display HTML from a `JournalArticle` in your app. To fit your needs, this Screenlet supports several use cases. This tutorial describes them.

Retrieving Basic Web Content

The simplest use case for Web Content Display Screenlet is to retrieve a web content article's HTML and render it in a `UIWebView`. To do this, provide the web content article's ID via the *Article Id* attribute in Interface Builder. The Screenlet takes care of the rest. This includes rendering the content to fit mobile devices, performing any required caching, and more.

To render the content *exactly* as it appears on your mobile site, however, you must provide the CSS inline or use a template. The HTML returned isn't aware of a Liferay instance's global CSS.

You can also modify the rendered HTML with a delegate, as explained in the Web Content Display Screenlet reference documentation.

As you can see, this is all fairly straightforward. What could go wrong? Famous last words. A common mistake is to use the default site ID (groupId) instead of the one for the site that contains your web content articles. To continue using a default groupId in your app, but use a different one for Web Content Display Screenlet, assign the Screenlet's *Group Id* property in Interface Builder.

Using Templates

Web Content Display Screenlet can also use templates to render web content articles. For example, your Liferay instance may have a custom template specifically designed to display content on mobile devices. To use a template, set the template's ID as the Screenlet's templateId property (*Template Id* in Interface Builder).

Recall that structured web content in Liferay can have many templates. You can create your own template if there's not one suitable for displaying web content in your app.

Rendering Structured Web Content

To render structured web content in Web Content Display Screenlet, you must create a custom theme capable of doing so. Also, you must create a custom theme for each structure you want to display in your app. In this case, you may find it convenient to create each theme inside a single parent theme and use compound naming to indicate this relationship. For example, if you have structures in your Liferay instance called *book*, *employee*, and *meeting*, you must create a custom theme for each. If you create these themes as children of another custom theme called *mytheme*, you could name them *mytheme.book*, *mytheme.employee*, and *mytheme.meeting*.

Regardless of where you create your themes or what you name them, use the following steps to create them:

1. Create a theme to render your web content. If you've already created your own theme, you can skip this step.

2. In your theme, create a new class called WebContentDisplayView_themeName, extending from BaseScreenletView. This class will hold the outlets and actions associated with the web content's UI.

3. Create the UI in the WebContentDisplayView_themeName.xib file. This file should have a UIView that contains the components you need to render the web content's structure fields. For example, if your structured web content contains latitude and longitude fields, you can use a MKMapView component to render the map point.

4. Once your components are ready, change the root view's class to WebContentDisplayView_themeName (the class you created in the first step), and create the outlets and actions you need to manage your UI components.

5. Conform the WebContentDisplayViewModel protocol in the WebContentDisplayView_themeName class. This protocol requires you to add the htmlContent and recordContent properties. The htmlContent property is intended for HTML web content; this isn't your theme's use case. Your theme must display structured web content; use the recordContent property for this content. In this property, set the structure field's value as the corresponding outlet's value. For example:

```
public var htmlContent: String? {
    get {
        return nil
    }
    set {
        // not used for structured Web Contents
    }
}

public var recordContent: DDLRecord? {
    didSet {
        // set the outlets with record's values
        set.myOutlet.myProperty = recordContent?["my_field_name"]?.currentValueAsLabel
    }
}
```

Next, you'll learn how to display a list of web content articles in your app.

Displaying a List of Web Content Articles

The preceding examples show you how to use Web Content Display Screenlet to display a single web content article's contents in your app. But what if you want to display a list of articles instead? No problem! You can do this by using Web Content List Screenlet, or Asset List Screenlet.

First, you'll learn how to use Web Content List Screenlet.

Using Web Content List Screenlet

Web Content List Screenlet lets you retrieve and display a list of web content articles from a web content folder. Follow these steps to use the Screenlet:

- Insert Web Content List Screenlet in your View Controller.
- Configure the *Group Id* and *Folder Id* properties in Interface Builder. The folder ID is the ID of the web content folder you want to display articles from. To use the root folder, use `0` for the Folder Id.
- To receive events related to the list, conform `WebContentListScreenletDelegate`. The events contain the `WebContent` objects.

For more information on the Screenlet and its supported functionality, see the Web Content List Screenlet reference documentation.

Using Asset List Screenlet

Asset List Screenlet is similar to Web Content Display Screenlet in that it can display a list of items from a Liferay instance. Asset List Screenlet, however, displays a list of assets. Since web content is an asset, you can use Asset List Screenlet to show a list of web content articles. Consider the following when doing this:

- In the delegate, `screenlet:onAssetListResponse` gets an array of `Asset` objects that represent `WebContent` objects. Since `WebContent` is a child of `Asset`, you can cast the `Asset` objects to `WebContent`. Each `WebContent` object has the `html`, `structure`, or `structuredRecord` properties.
- To render Asset List Screenlet with `WebContent` objects, you must create your own theme. Create a class in your theme that extends `AssetListView_default`, and override the `doFillLoadedCell` method. In this method, cast the `object` parameter as `WebContent` and then retrieve field values from the web content's `structuredRecord` property. If you want custom cells, you can also override the `doRegisterCellNibs` and `doCreateCell` methods. See the Asset List Screenlet reference documentation for more details on customizing your asset list.

Related Topics

Using Screenlets in iOS Apps
  Using Themes in iOS Screenlets
  Creating iOS Themes
  Web Content Display Screenlet for iOS
  Web Content List Screenlet for iOS
  Asset List Screenlet for iOS

77.22 Rendering Web Pages in Your iOS App

The Rendering Web Content tutorial shows you how to display web content from a Liferay DXP site in your iOS app. Displaying content is great, but what if you want to display an entire page? No problem! Web Screenlet lets you display any web page. You can even customize the page by injecting local or remote JavaScript and CSS files. When combined with Liferay DXP's server-side customization features (e.g., Application Display Templates), Web Screenlet gives you almost limitless possibilities for displaying web pages in your iOS apps.

In this tutorial, you'll learn how to use Web Screenlet to display web pages in your iOS app.

Inserting Web Screenlet in Your App

Inserting Web Screenlet in your app is the same as inserting any Screenlet in your app:

1. In Interface Builder, insert a new view (`UIView`) in a new view controller. This new view should be nested under the view controller's existing view.

2. With the new view selected, open the Identity inspector and set the view's Custom Class to `WebScreenlet`.

3. Set any constraints that you want for the Screenlet in the scene.

The exact steps for configuring Web Screenlet are unique to Web Screenlet. First, you'll conform your view controller to Web Screenlet's delegate protocol.

Conforming to Web Screenlet's Delegate Protocol

To use any Screenlet, you must conform the class of the view controller that contains it to the Screenlet's delegate protocol. Web Screenlet's delegate protocol is `WebScreenletDelegate`. Follow these steps to conform your view controller to `WebScreenletDelegate`:

1. Import `LiferayScreens` and set your view controller to adopt the `WebScreenletDelegate` protocol:

```
import UIKit
import LiferayScreens

class ViewController: UIViewController, WebScreenletDelegate {...
```

2. Implement the `WebScreenletDelegate` method `onWebLoad(_:url:)`. This method is called when the Screenlet loads the page successfully. How you implement it depends on what (if anything) you want to happen upon page load. Its arguments are the `WebScreenlet` instance and the page URL. This example prints a message to the console indicating that the page was loaded:

```
func onWebLoad(_ screenlet: WebScreenlet, url: String) {
    // Called when the page is loaded
    print("\(url) was just loaded")
}
```

3. Implement the `WebScreenletDelegate` method `screenlet(_:onError:)`. This method is called when an error occurs loading the page, and therefore includes the `NSError` object. This lets you log or print the error. For example, this implementation prints a message containing the error's description:

```
func screenlet(_ screenlet: WebScreenlet, onError error: NSError) {
    print("Failed to load the page: \(error.localizedDescription)")
}
```

4. Implement the `WebScreenletDelegate` method `screenlet(_:onScriptMessageNamespace:onScriptMessage:)`. This method is called when the Screenlet's `WKWebView` sends a message. This method's arguments include the message's namespace and the message. How you implement this method depends on what you want to happen when the message is sent. For example, you could perform a segue and include the message as the segue's `sender`:

```
func screenlet(_ screenlet: WebScreenlet,
    onScriptMessageNamespace namespace: String,
    onScriptMessage message: String) {

    performSegue(withIdentifier: "detail", sender: message)
}
```

5. Get a reference to the Web Screenlet on your storyboard by using Interface Builder to create an outlet to it in your view controller. It's a best practice to name a Screenlet outlet after the Screenlet it references, or simply `screenlet`. Here's an example Web Screenlet outlet:

```
@IBOutlet weak var webScreenlet: WebScreenlet?
```

6. In the view controller's `viewDidLoad()` method, use the Web Screenlet reference you just created to set the view controller as the Screenlet's delegate. To do this, add the following line of code just below the `super.viewDidLoad()` call:

```
self.webScreenlet?.delegate = self
```

Next, you'll use the same Web Screenlet reference to set the Screenlet's parameters.

Setting Web Screenlet's Parameters

Web Screenlet has `WebScreenletConfiguration` and `WebScreenletConfigurationBuilder` objects that supply the parameters the Screenlet needs to work. These parameters include the URL of the page to load and the location of any JavaScript or CSS files that customize the page. You'll set most of these parameters via `WebScreenletConfigurationBuilder`'s methods.

Note: For a full list of `WebScreenletConfigurationBuilder`'s methods, and a description of each, see the table in the Configuration section of Web Screenlet's reference doc.

To set Web Screenlet's parameters, follow these steps in the `viewDidLoad()` method of a view controller that uses Web Screenlet:

1. Use `WebScreenletConfigurationBuilder(<url>)`, where `<url>` is the web page's URL string, to create a `WebScreenletConfigurationBuilder` object. If the page requires Liferay DXP authentication, then the user must be logged in via Login Screenlet or a `SessionContext` method, and you must provide a relative URL to the `WebScreenletConfigurationBuilder` constructor. For example, if such a page's full URL is `http://your.liferay.instance/web/guest/blog`, then the constructor's argument is `/web/guest/blog`. For any other page that doesn't require Liferay DXP authentication, you must supply the full URL to the constructor.

2. Call the `WebScreenletConfigurationBuilder` methods to set the parameters that you need.

```
**Note:** If the URL you supplied to the `WebScreenletConfigurationBuilder`
constructor is to a page that doesn't require Liferay DXP authentication, then
you must call the `WebScreenletConfigurationBuilder` method
`set(webType: .other)`. The default `WebType` is `.liferayAuthenticated`,
which is required to load Liferay DXP pages that require authentication. If
you need to set `.liferayAuthenticated` manually, call
`set(webType: .liferayAuthenticated)`.
```

---

3. Call the `WebScreenletConfigurationBuilder` instance's `load()` method, which returns a `WebScreenletConfiguration` object.

4. Set the `WebScreenletConfiguration` object to the Web Screenlet instance's `configuration` property.

5. Call the Web Screenlet instance's `load()` method.

Here's an example snippet of these steps in the `viewDidLoad()` method of a view controller in which the Web Screenlet instance is `webScreenlet`, and the `WebScreenletConfiguration` object is `webScreenletConfiguration`:

```
override func viewDidLoad() {
    super.viewDidLoad()

    self.webScreenlet?.delegate = self

    let webScreenletConfiguration =
        WebScreenletConfigurationBuilder(url: "/web/westeros-hybrid/companynews")
            .addCss(localFile: "blogs")
            .addJs(localFile: "blogs")
            .load()
    webScreenlet.configuration = webScreenletConfiguration
    webScreenlet.load()
}
```

The relative URL `/web/westeros-hybrid/companynews` supplied to the `WebScreenletConfigurationBuilder` constructor, and the lack of a `set(webType: .other)` call, indicates that this Web Screenlet instance loads a Liferay DXP page that requires authentication. The `addCss` and `addJs` calls add local CSS and JavaScript files, respectively. Both files are named `blogs`.

Great! Now you know how to use Web Screenlet in your iOS apps.

Related Topics

Web Screenlet for iOS

  Using Web Screenlet with Cordova in Your iOS App

  Using Screenlets in iOS Apps

  Rendering Web Content in Your iOS App

77.23 Using Web Screenlet with Cordova in Your iOS App

By using Cordova plugins in Web Screenlet, you can extend the functionality of the web page that the Screenlet renders. This lets you tailor that page to your app's needs. You'll get started by installing Cordova.

Installing and Configuring Cordova Automatically

Follow these steps to automatically create an empty Android project configured to use Cordova. Note that you must have git, Node.js and npm, and CocoaPods installed.

1. Install `screens-cli`:

   ```
   npm install -g screens-cli
   ```

2. Create the file `.plugins.screens` in the folder you want to create your project in. In this file, add all the Cordova plugins you want to use in your app. For example, you can add plugins from Cordova or GitHub:

   ```
   https://github.com/apache/cordova-plugin-wkwebview-engine.git
   cordova-plugin-call-number
   cordova-plugin-camera
   ```

   Note that the `WKWebView` Engine plugin is mandatory in iOS.

3. In the folder containing your `.plugins.screens` file, run `screens-cli` to create your project:

   ```
   screens-cli ios <project-name>
   ```

   This creates your project in the folder `platforms/ios/<project-name>`.

4. Run the following in `platforms/ios/<project-name>`:

   ```
   pod install
   ```

5. Open the `<project-name>.xcworkspace` file with Xcode.

Installing and Configuring Cordova Manually

Follow these steps to install and configure Cordova:

1. Follow the Cordova getting started guide to install Cordova, create a Cordova project, and add the iOS platform to your Cordova project.

2. Install the Cordova `WKWebView` engine:

   ```
   cordova plugin add cordova-plugin-wkwebview-engine
   ```

3. Install any other Cordova plugins you want to use in your app. You can use `cordova plugin` to view the currently installed plugins.

4. Copy the following files and folders from your Cordova project to your iOS project's root folder:

   - `platforms/ios/<your-cordova-project>/config.xml`
   - `platforms/ios/<your-cordova-project>/Plugins`
   - `platforms/ios/www`

5. In the `config.xml` file you just copied to your iOS project's root folder, add `<allow-navigationhref="*" />` below `<access origin="*" />`.

Using Cordova in Web Screenlet

Now that you've installed and configured Cordova in your iOS project, you're ready to use it with Web Screenlet. Follow these steps to do so:

1. Insert and configure Web Screenlet in your app.

2. When you set Web Screenlet's parameters via the `WebScreenletConfigurationBuilder` object, call the `enableCordova()` method. For example, this code adds a local JavaScript file via `addJs` and then calls `enableCordova()` before loading the configuration and the Screenlet:

   ```
   let configuration = WebScreenletConfigurationBuilder(url: "url")
       .addJs(localFile: "call")
       .enableCordova()
       .load()

   webScreenlet?.configuration = configuration
   webScreenlet?.load();
   ```

That's it! Note, however, that you may also need to invoke Cordova from a JavaScript file, depending on what you're doing. For example, to use the Cordova plugin `cordova-plugin-call-number` to call a number, then you must add a JavaScript file with the following code:

```
function callNumber() {
    //This line triggers the Cordova plugin and makes a call
    window.plugins.CallNumber.callNumber(null, function(){ alert("Calling failed.") }, "900000000", true);
}

setTimeout(callNumber, 3000);
```

If you run the app containing this code and wait three seconds, the plugin activates and calls the number in the JavaScript file.

Great! Now you know how to use Web Screenlet with Cordova.

Related Topics

Rendering Web Pages in Your iOS App
    Web Screenlet for iOS

77.24 iOS Best Practices

When developing iOS projects with Liferay Screens, there are a few best practices that you should follow to ensure your code is as clean and bug-free as possible. This tutorial lists these. Note that this tutorial doesn't cover Swift coding conventions for contributing to the Liferay Screens project on GitHub. Click here to see these.

Naming Conventions

Using the naming conventions described here leads to consistency and a better understanding of the Screens library. This makes working with your Screenlets much simpler.

Screenlet Folder

Your Screenlet folder's name should indicate your Screenlet's functionality. For example, Login Screenlet's folder is named `LoginScreenlet`.

If you have multiple Screenlets that operate on the same entity, you can place them inside a folder named for that entity. For example, Asset Display Screenlet and Asset List Screenlet both work with Liferay assets. They're therefore in the Screens library's `Asset` folder.

Screenlets

Naming Screenlets properly is very important; they're the main focus of Liferay Screens. Your Screenlet should be named with its principal action first, followed by *Screenlet*. Its Screenlet class should also follow this pattern. For example, Login Screenlet's principal action is to log users into a Liferay instance. Its Screenlet class is `LoginScreenlet`.

View Models

You should place View Models in your Screenlet's root folder and name them after your Screenlet. For example, Forgot Password Screenlet's View Model is in the `ForgotPasswordScreenlet` folder and is named `ForgotPasswordViewModel`.

Interactors

You should place your Screenlet's Interactors in a folder named `Interactors` in your Screenlet's root folder. You should name each Interactor with its action first, followed by *Interactor*. For example, Rating Screenlet has three Interactors in its `Interactors` folder:

- `DeleteRatingInteractor`: Deletes an asset's ratings
- `LoadRatingsInteractor`: Loads an asset's ratings
- `UpdateRatingInteractor`: Updates an asset's ratings

Connectors

Name your Connectors with the same naming conventions as Interactors, replacing *Interactor* with *Connector*. If your Connector calls a Liferay service, precede *Connector* with *Liferay*. For example, the Connector `CommentAddLiferayConnector` adds comments to an asset in a Liferay instance. A Connector that retrieves a webpage's title from any URL would be called `GetWebsiteTitleConnector`.

Themes

Place your Screenlet's Themes in a folder named *Themes* in your Screenlet's root folder. If you're creating a group of similarly styled Themes for multiple Screenlets, however, then you can place them in a separate *Themes* folder outside of your Screenlets' root folders. This is what the Screens Library does for its Default and Flat7 Themes. The `Default` and `Flat7` folders each contain similarly styled Themes for several Screenlets. Also note that each Screenlet's Theme is in its own folder. For example, Forgot Password Screenlet's Default Theme is in the folder `Themes/Default/Auth/ForgotPasswordScreenlet`. Note that the `Auth` folder is the Screenlet's module. Creating your Screenlets and Themes in modules isn't required.

Recall that a Theme consists of an XIB file and a View class. Name these after your Screenlet, but with *View* instead of *Screenlet*. The filenames should also be suffixed with `_yourThemeName`. For example, the XIB file and View class for Forgot Password Screenlet's Default theme are `ForgotPasswordView_default.xib` and `ForgotPasswordView_default.swift`, respectively.

Avoid Hardcoded Elements

Using constants instead of hard coded elements is a simple way to avoid bugs. Constants reduce the likelihood that you'll make a typo when referring to common elements. They also gather these elements in a single location. For example, when you add an action to your Screenlet, each Screenlet action used as a `restorationIdentifier` in the View class is defined as a constant in the Screenlet class. The Screenlet class's `createInteractor` method then uses the constants to distinguish between the actions. If you instead typed each action manually in both places, a typo could break your Screenlet and would be difficult to track down. Defining the actions in one place via constants avoids this potentially maddening complication.

Screenlet attributes, like those listed in each Screenlet's reference documentation, are another good example of this. Although you can set these directly in Interface Builder, it's better to set them via constants in a `plist` file. This puts all your Screenlets' attributes in a single location that is also subject to version control. For instructions on setting attributes in a `plist` file, see the Configuring Communication with Liferay section of the tutorial on preparing iOS projects for Liferay Screens.

To retrieve these values in your code, you can use the following `LiferayServerContext` methods:

- `propertyForKey`: Get a property as an `AnyObject`
- `numberPropertyForKey`: Get a property as an `NSNumber`.
- `longPropertyForKey`: Get a property as an `Int64`.
- `intPropertyForKey`: Get a property as an `Int`.
- `booleanPropertyForKey`: Get a property as a `Bool`.
- `datePropertyForKey`: Get a property as an `NSDate`.
- `stringPropertyForKey`: Get a property as a `String`.

For example, the following code retrieves the `galleryFolderId` value and sets it to Image Gallery Screenlet's `folderId` attribute:

```
@IBOutlet weak var imageGalleryScreenlet: ImageGalleryScreenlet? {
    didSet {
        imageGalleryScreenlet?.delegate = self
        imageGalleryScreenlet?.presentingViewController = self

        imageGalleryScreenlet?.repositoryId = LiferayServerContext.groupId
        imageGalleryScreenlet?.folderId = LiferayServerContext.longPropertyForKey("galleryFolderId")
    }
}
```

Stay in Your Layer

When accessing variables that belong to other Screenlet components, you should avoid those outside your current Screenlet layer. This achieves better decoupling between the layers, which tends to reduce bugs and simplify maintenance. For an explanation of the layers in Liferay Screens, see the architecture tutorial. For example, you shouldn't directly access View variables from an Interactor. This Interactor's `start` method gets a View instance and accesses its `title` variable:

```
public class MyInteractor: Interactor {
    override func start() -> Bool {
        if let view = self.screenlet.screenletView as? MyView {
            let title = view.title
            ...
        }
    }
}
```

Instead, you should pass the variable to the Interactor's initializer. The Interactor now contains its own `title` variable, set in its initializer:

```
public class MyInteractor: Interactor {

    public let title: String

    //MARK: Initializer

    public init(screenlet: BaseScreenlet, title: String) {
        self.title = title
        super.init(screenlet: screenlet)
    }
}
```

The Screenlet class's `createInteractor` method calls this initializer when creating an instance of the Interactor. Also note that the Screenlet's View Model is used to retrieve the View's `title`. As explained in the tutorial Supporting Multiple Themes in Your Screenlet, a View Model serves as an abstraction layer for your View, which lets you use different Themes with a Screenlet:

```
public class MyScreenlet: BaseScreenlet {
    ...
    override public func createInteractor(name name: String, sender: AnyObject?) -> Interactor? {
        let interactor = MyInteractor(self, title: viewModel.title)
        ...
    }
    ...
}
```

There are, however, a few places where you can break this rule (otherwise it wouldn't be possible for layers to interact):

- The Screenlet class's `createInteractor` method. To get the user's input, this method must access the View's computed properties.

- The Interactor's `onSuccess` closure in the Screenlet class. Here, you must retrieve the Interactor's result object.
- When using a Connector, the Interactor's `completedConnector` method. In this method, you must retrieve the Connector's result object.
- The Screenlet class's View Model references. This is required for the Screenlet to communicate with the View.

Related Topics

Creating iOS Screenlets
    Creating iOS List Screenlets
    Creating iOS Themes
    Supporting Multiple Themes in Your Screenlet
    Adding Screenlet Actions
    Create and Use a Connector with Your Screenlet
    Architecture of Liferay Screens for iOS

Chapter 78

Using Xamarin with Liferay Screens

Liferay Screens for Android and iOS lets you use *Screenlets* to develop native mobile apps on each platform. Screenlets are complete visual components that you insert in your app to leverage Liferay DXP's content and services. As of Liferay Screens 3.0, you can use Screenlets with Xamarin to develop hybrid mobile apps for Android and iOS.

The tutorials in this section show you how to develop hybrid mobile apps using Liferay Screens and Xamarin. You'll start by preparing your Xamarin project for Screens. You'll then learn how to use Screenlets in Xamarin, customize their appearance, and more.

Note: These tutorials assume that you know how to use Xamarin. If you need assistance with Xamarin, see its documentation.

78.1 Preparing Xamarin Projects for Liferay Screens

To use Liferay Screens with Xamarin, you must install Screens in your Xamarin project. You must then configure your project to communicate with your Liferay DXP instance. Note that Liferay Screens for Xamarin is released as a NuGet package hosted in NuGet.org.

Note: After installation, you must configure Liferay Screens to communicate with your Liferay DXP instance. The last section in this tutorial shows you how to do this.

Requirements and Example Projects

Liferay Screens for Xamarin includes the bindings necessary to use all Screenlets included with Screens. The following software is required:

- Visual Studio
- Android SDK 4.1 (API Level 16) or above
- Liferay CE Portal 7.0/7.1, or Liferay DXP 7.0
- Liferay Screens NuGet package

Also note that if you get confused or stuck while using Screens for Xamarin, the official Liferay Screens repository contains two sample Xamarin projects that you can reference:

- **Showcase-Android:** An example app for Xamarin.Android containing all the currently available Screenlets.

- **Showcase-iOS:** An example app for Xamarin.iOS containing all the currently available Screenlets.

Securing JSON Web Services

Each Screenlet in Liferay Screens calls one or more of Liferay DXP's JSON web services, which are enabled by default. The Screenlet reference documentation for Android and iOS lists the web services that each Screenlet calls. To use a Screenlet, its web services must be enabled in the portal. It's possible, however, to disable the web services needed by Screenlets you're not using. For instructions on this, see the tutorial Configuring JSON Web Services. You can also use Service Access Policies for more fine-grained control over accessible services.

Install Liferay Screens in Xamarin Solutions

Follow these steps to install Liferay Screens in your Xamarin project:

1. Open your project in Visual Studio.

2. Right click your project's *Packages* folder and then select *Add packages…*.

3. Look for *LiferayScreens* and install the latest version.

4. Accept the license agreements for any dependencies. These are necessary to use Liferay Screens in Xamarin.

5. Check your configuration one of these ways:

    - Rebuild and execute your project, and import a Liferay Screens path (e.g., `Com.Liferay.Mobile.Screens.Auth.Logir`
    - In your project, select *References → From Packages* and look for the `LiferayScreens*` file. Open that file in the Assembly Browser. You should then see all the available Liferay Screens files.

Next, you'll set up communication with Liferay DXP.

Configuring Communication with Liferay DXP

Before using Liferay Screens, you must configure your project to communicate with your Liferay DXP instance. To do this, you must provide your project with the following information:

- Your Liferay DXP instance's ID.
- The ID of the Liferay DXP site your app needs to communicate with.
- Your Liferay DXP instance's version.
- Any other information required by specific Screenlets.

Fortunately, this is straightforward. Do the following in your Xamarin projects:

- For Xamarin.Android, create a new file called `server_context.xml` in the `Resources/values` folder. Add the following code to this file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <!-- Change these values for your portal installation -->
    <string name="liferay_server">http://10.0.2.2:8080</string>

    <integer name="liferay_company_id">20116</integer>
    <integer name="liferay_group_id">20143</integer>

    <integer name="liferay_portal_version">70</integer>

</resources>
```

- For Xamarin.iOS, create a new file called `liferay-server-context.plist` in the `Resources` folder. Add the following code to this file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>server</key>
    <string>http://localhost:8080</string>
    <key>version</key>
    <integer>70</integer>
    <key>companyId</key>
    <real>20116</real>
    <key>groupId</key>
    <real>20143</real>
</dict>
</plist>
```

Make sure to change these values to match those of your Liferay DXP instance. The server address `http://10.0.2.2:8080` is suitable for testing with Android Studio's emulator, because it corresponds to `localhost:8080` through the emulator. If you're using the Genymotion emulator, you should, however, use `192.168.56.1` instead of `localhost`.

The `liferay_company_id` and `companyId` values are your Liferay DXP instance's ID. You can find this in your Liferay DXP instance at *Control Panel* → *Configuration* → *Virtual Instances*. The instance's ID is in the *Instance ID* column.

The `liferay_group_id` and `groupId` values are the ID of the site your app needs to communicate with. To find this value, first go to the site in your Liferay DXP instance that you want your app to communicate with. In the *Site Administration* menu, select *Configuration* → *Site Settings*. The site ID is listed at the top of the *General* tab.

The `liferay_portal_version` and `version` value `70` tells Screens that it's communicating with a Liferay CE Portal 7.0 or Liferay DXP 7.0 instance. Here are the supported values and the portal versions they correspond to:

- `71`: Liferay CE Portal 7.1 or Liferay DXP 7.1
- `70`: Liferay CE Portal 7.0 or Liferay DXP 7.0
- `62`: Liferay Portal 6.2 CE/EE

You can also configure Screenlet properties in `server_context.xml` and `liferay-server-context.plist`. The example `server_context.xml` properties listed below, `liferay_recordset_id` and `liferay_recordset_fields`, enable DDL Form Screenlet and DDL List Screenlet to interact with a Liferay DXP instance's DDLs:

```
<!-- Change these values for your portal installation -->

<integer name="liferay_recordset_id">20935</integer>
<string name="liferay_recordset_fields">Title</string>
```

For additional examples of these files, see the Showcase-Android and Showcase-iOS sample projects.

Super! Your Xamarin projects are ready for Liferay Screens.

Related Topics

Using Screenlets in Xamarin Apps

Using Views in Xamarin.Android

Using Themes in Xamarin.iOS

Creating Xamarin Views and Themes

Liferay Screens for Xamarin Troubleshooting and FAQs

78.2 Using Screenlets in Xamarin Apps

You can start using Screenlets once you've prepared your Xamarin project to use Liferay Screens. The Screenlet reference documentation describes the available Screenlets:

- Screenlets in Liferay Screens for Android
- Screenlets in Liferay Screens for iOS

Using Screenlets is very straightforward. This tutorial shows you how to insert and configure Screenlets in your Xamarin app. You'll be a Screenlet master in no time!

Xamarin.iOS

Follow these steps to insert Screenlets in your Xamarin.iOS app:

1. Insert a view (`UIView`) in your storyboard (in Visual Studio's iOS Designer or Xcode's Interface Builder). Note that if you're editing an XIB file, you must insert the view inside the XIB's parent view.

2. Set the view's class to the class of the Screenlet you want to use. For example, Login Screenlet's class is `LoginScreenlet`. If you're using Xamarin Designer for iOS in Visual Studio, you must also give the view a name so you can refer to it in your view controller's code.

   For example, the following video shows the first two steps for inserting Login Screenlet in a Xamarin Designer for iOS storyboard.

3. Configure the Screenlet's behavior in your app by implementing the Screenlet's delegate in your view controller. To configure your app to listen for events the Screenlet triggers, implement the Screenlet's delegate methods and register the view controller as the delegate. Make sure to annotate each delegate method with `[Export(...)]`. This ensures the method can be called from Objective-C, which is required for it to work in Screens. You should also set any Screenlet attributes you need. Each Liferay Screenlet's reference documentation lists its available attributes and delegate methods.

```
**Note:** In Liferay Screens for Xamarin, Screenlet delegates are prefixed
with an `I`. For example, Login Screenlet's delegate in native code is
`LoginScreenletDelegate`, while in Xamarin it's `ILoginScreenletDelegate`.
```

```
For example, here's a view controller that implements Login Screenlet's
delegate, `ILoginScreenletDelegate`. Note that the `ViewDidLoad()` method
sets the Screenlet's `ThemeName` attribute (`ThemeName` is available for all
Screenlets via `BaseScreenlet` inheritance) and registers the view
controller as the delegate. This view controller also implements the
`OnLoginResponseUserAttributes` method, which is called upon successful
login. Also note this method's `[Export(...)]` annotation:

    public partial class ViewController : UIViewController, ILoginScreenletDelegate
    {
        protected ViewController(IntPtr handle) : base(handle) {}

        public override void ViewDidLoad()
        {
            base.ViewDidLoad();

            // Set the Screenlet's attributes
            this.loginScreenlet.ThemeName = "demo";

            // Registers this view controller as the delegate
            this.loginScreenlet.Delegate = this;
        }

        ...

        // Delegate methods

        [Export("screenlet:onLoginResponseUserAttributes:")]
        public virtual void OnLoginResponseUserAttributes(BaseScreenlet screenlet,
            NSDictionary<NSString, NSObject> attributes)
        {
            ...
        }
    }
```

See the Showcase-iOS app for more examples of view controllers that use Liferay's Screenlets.

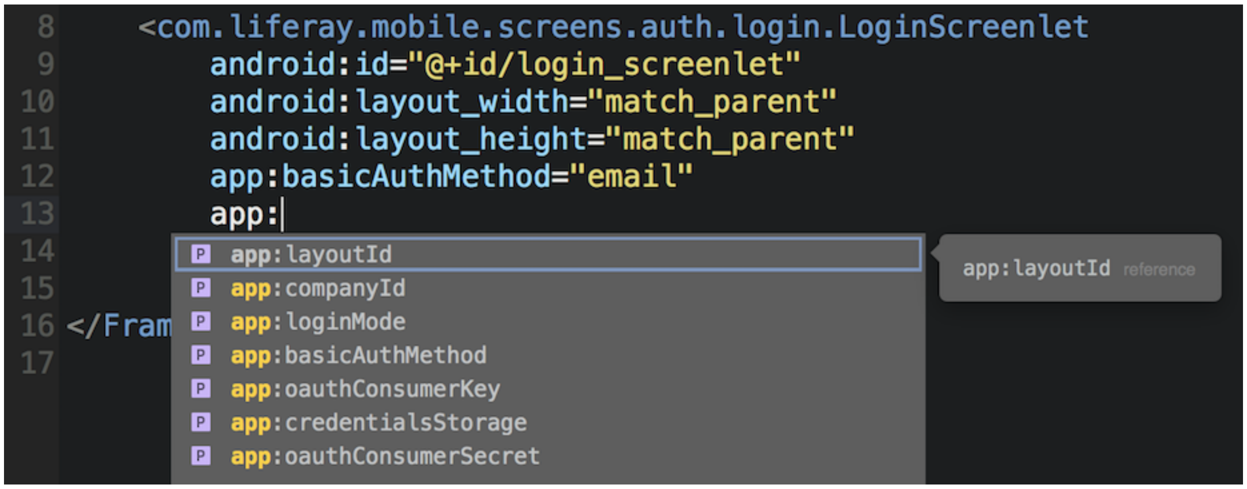
Xamarin.Android

Follow these steps to insert Screenlets in your Xamarin.Android app:

1. Open your app's layout AXML file and insert the Screenlet's XML in your activity or fragment layout. For example, here's Login Screenlet's XML in an activity's `FrameLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <com.liferay.mobile.screens.auth.login.LoginScreenlet
        android:id="@+id/login_screenlet"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:basicAuthMethod="email"/>
</FrameLayout>
```

2. Set the Screenlet's attributes. If it's a Liferay Screenlet, refer to the Screenlet reference documentation to learn the Screenlet's required and supported attributes. This screenshot shows Login Screenlet's attributes being set:



```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:basicAuthMethod="email"
    app:
</Fram
```

Figure 78.1: You can set a Screenlet's attributes via the app's layout AXML file.

3. To configure your app to listen for events the Screenlet triggers, implement the Screenlet's listener interface in your activity or fragment class. Refer to the Screenlet's reference documentation to learn its listener interface. Then register that activity or fragment as the Screenlet's listener.

```
**Note:** In Liferay Screens for Xamarin, Screenlet listeners are prefixed
with an `I`. For example, Login Screenlet's listener in native code is
`LoginListener`, while in Xamarin it's `ILoginListener`.
```

```
For example, the following activity class implements Login Screenlet's
`ILoginListener` interface, and registers itself as the Screenlet's
listener via `loginScreenlet.Listener = this`. Note that the listener
methods `OnLoginSuccess` and `OnLoginFailure` are called when login succeeds
and fails, respectively. In this case, these methods print simple toast
messages:

    [Activity]
    public class LoginActivity : Activity, ILoginListener
    {
        LoginScreenlet loginScreenlet;

        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);
            SetContentView(Resource.Layout.LoginView);

            loginScreenlet = (LoginScreenlet) FindViewById(Resource.Id.login_screenlet);
            loginScreenlet.Listener = this;
        }

        // ILoginListener

        public void OnLoginSuccess(User p0)
        {
            Toast.MakeText(this, "Login success: " + p0.Id, ToastLength.Short).Show();
        }

        public void OnLoginFailure(Java.Lang.Exception p0)
        {
            Android.Util.Log.Debug("LoginScreenlet", $"Login failed: {p0.Message}");
        }

    }
```

See the Showcase-Android app for more examples of activities that use Liferay's Screenlets.

Related Topics

Preparing Xamarin Projects for Liferay Screens
Using Views in Xamarin.Android
Using Themes in Xamarin.iOS
Creating Xamarin Views and Themes
Liferay Screens for Xamarin Troubleshooting and FAQs

78.3 Using Views in Xamarin.Android

You can use a Liferay Screens *View* to set a Screenlet's look and feel independent of the Screenlet's core functionality. Liferay's Screenlets come with several Views, and more are being developed by Liferay and the community. The Screenlet reference documentation lists the Views available for each Screenlet included with Screens. This tutorial shows you how to use Views in Xamarin.Android.

Views and View Sets

The concepts and components that comprise Views and View Sets in Liferay Screens for Xamarin are the same as they are in Liferay Screens for Android. For a brief description of these components, see the section on Views and View Sets in the general tutorial on using Views. For a detailed description of the View layer in Liferay Screens, see the tutorial Architecture of Liferay Screens for Android.

Using Views

Follow these steps to use a View in Xamarin.Android:

1. Copy the layout of the View you want to use from the Liferay Screens repository to your app's `res/layout` folder. Alternatively, you can create a new layout. The following links list the View layouts available in each View Set:

    - Default
    - Material
    - Westeros

    For example, this is Login Screenlet's Material View, `login_material.xml`:

    ```
    <com.liferay.mobile.screens.viewsets.material.auth.login.LoginView
        xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:liferay="http://schemas.android.com/apk/res-auto"
        android:paddingLeft="40dp"
        android:paddingRight="40dp"
        style="@style/default_screenlet">

    <LinearLayout
        android:id="@+id/basic_authentication_login"
    ```

```
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <LinearLayout style="@style/material_row">

        <ImageView
            android:id="@+id/drawable_login"
            android:contentDescription="@string/user_login_icon"
            android:src="@drawable/material_email"
            style="@style/material_icon"/>

        <EditText
            android:id="@+id/liferay_login"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_marginTop="8dp"
            android:inputType="text"
            android:labelFor="@+id/liferay_login"/>

    </LinearLayout>

    <LinearLayout style="@style/material_row">

        <ImageView
            android:id="@+id/drawable_password"
            android:contentDescription="@string/password_icon"
            android:src="@drawable/material_https"
            style="@style/material_icon"/>

        <EditText
            android:id="@+id/liferay_password"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_marginTop="8dp"
            android:hint="@string/password"
            android:inputType="textPassword"/>

    </LinearLayout>

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="32dp">

        <Button
            android:id="@+id/liferay_login_button"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_margin="10dp"
            android:text="@string/sign_in"/>

        <com.liferay.mobile.screens.base.ModalProgressBar
            android:id="@+id/liferay_progress"
            android:layout_width="wrap_content"
            android:layout_height="match_parent"
            android:layout_gravity="center_vertical|left"
            android:layout_margin="10dp"
            android:theme="@style/white_theme"
            android:visibility="invisible"
            liferay:actionViewId="@id/liferay_login_button"/>
    </FrameLayout>
</LinearLayout>

<Button
    android:id="@+id/oauth_authentication_login"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/authorize_application"
    android:visibility="gone"/>

</com.liferay.mobile.screens.viewsets.material.auth.login.LoginView>
```

2. When you insert the Screenlet's XML in the layout of the activity or fragment you want the Screenlet to appear in, set the `liferay:layoutId` attribute to the View's layout. For example, here's Login Screenlet's XML with `liferay:layoutId` set to `@layout/login_material`, which specifies Login Screenlet's Material View from the previous step:

```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    liferay:layoutId="@layout/login_material"
    />
```

3. If the View you want to use is part of a View Set (e.g., the Material View is part of the Material View Set), your app or activity's theme must also inherit the theme that defines that View Set's styles. For example, the following code in an app's `Resources/values/Styles.xml` tells `AppTheme.NoActionBar` to use the Material View Set as its parent theme:

```
<resources>
    <style name="AppTheme.NoActionBar" parent="material_theme">
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>

        <item name="windowActionBar">false</item>
        <item name="windowNoTitle">true</item>
    </style>
    ...
</resources>
```

   To use the Default or Westeros View Set, inherit `default_theme` or `westeros_theme`, respectively.

Awesome! Now you know how to use Views to spruce up your Xamarin.Android Screenlets.

Related Topics

Preparing Xamarin Projects for Liferay Screens
Using Screenlets in Xamarin Apps
Using Themes in Xamarin.iOS
Creating Xamarin Views and Themes
Liferay Screens for Xamarin Troubleshooting and FAQs

78.4 Using Themes in Xamarin.iOS

Themes in Xamarin.iOS are analogous to *Views* in Xamarin.Android. Like Views, Themes let you set a Screenlet's look and feel independent of the Screenlet's core functionality. Liferay's Screenlets come with several Themes, and more are being developed by Liferay and the community. The Screenlet reference documentation lists the Themes available for each Screenlet included with Screens. This tutorial shows you how to use Themes in Xamarin.iOS.

Installing and Using Themes

Follow these steps to install and use a Theme:

1. If the Theme is packaged as a NuGet dependency, you can install it in your project via NuGet. To do so, right-click your project's *Packages* folder and then select *Add packages…*. Then search for the Theme and install it. If the Theme isn't available in NuGet, you can drag and drop the Theme's folder directly into your project.

2. To use the installed Theme, set its name to the Screenlet instance's `ThemeName` property in your view controller that implements the Screenlet's delegate. All Screenlets inherit this property from `BaseScreenlet`. For example, this code sets Login Screenlet's `ThemeName` property to the Material Theme:

   ```
   loginScreenlet.ThemeName = "material"
   ```

   If you don't set this property or enter an invalid or missing Theme, the Screenlet uses its Default Theme. Each Screenlet's available Themes are listed in the *Themes* section of the Screenlet's reference documentation.

Great, that's it! Now you know how to use Themes to dress up Screenlets in your Xamarin.iOS apps.

Related Topics

Preparing Xamarin Projects for Liferay Screens
Using Screenlets in Xamarin Apps
Using Views in Xamarin.Android
Creating Xamarin Views and Themes
Liferay Screens for Xamarin Troubleshooting and FAQs

78.5 Creating Xamarin Views and Themes

Recall that Views in Xamarin.Android and Themes in Xamarin.iOS are analogous components that let you customize a Screenlet's look and feel. You can use the Views and Themes provided by Liferay Screens, or write your own. Writing your own lets you tailor a Screenlet's UI to your exact specifications. This tutorial shows you how to do this.

You can create Views and Themes from scratch, or use an existing one as a foundation. Views and Themes include a View class for implementing the Screenlet UI's behavior, a Screenlet class for notifying listeners/delegates and invoking Interactors, and an AXML or XIB file for defining the UI.

There are also different types of Views and Themes. These are discussed in the tutorials on creating Views and Themes in native code. You should read those tutorials before creating Views in Xamarin.Android or Themes in Xamarin.iOS.

First, you'll determine where to create your View or Theme.

Determining the Location of Your View or Theme

If you plan to reuse or redistribute your View or Theme, create it in a new Xamarin project as a multiplatform library for code sharing. Otherwise, create it in your app's project.

Creating a Xamarin.Android View

Creating Views for Xamarin.Android is very similar to doing so in native code. You can create the following View types:

- **Themed View:** Creating a Themed View in Xamarin.Android is identical to doing so in native code. In Xamarin.Android, however, only the Default View Set is available to extend.
- **Child View:** Creating a Child View in Xamarin.Android is identical to doing so in native code.
- **Extended View:** Creating an Extended View in Xamarin.Android differs from doing so in native code. The next section shows you how.

Extended View

To create an Extended View in Xamarin.Android, follow the steps for creating an Extended View in native code, but make sure your custom View class in the second step is the appropriate C# class. For example, here's the View class from the native code tutorial, converted to C#:

```
using System;
using Android.Content;
using Android.Util;
using Com.Liferay.Mobile.Screens.Viewsets.Defaultviews.Auth.Login;

namespace ShowcaseAndroid.CustomViews
{
    public class LoginCheckPasswordView : LoginView
    {
        public LoginCheckPasswordView(Context context) : base(context) { }

        public LoginCheckPasswordView(Context context, IAttributeSet attributes) : base(context, attributes) {}

        public LoginCheckPasswordView(Context context, IAttributeSet attributes, int defaultStyle) : base(context, attributes, defaultStyle) {}

        public override void OnClick(Android.Views.View view)
        {
            // compute password strength
            if (PasswordIsStrong) {
                base.OnClick(view);
            }
            else {
                // Present user message
            }
        }
    }
}
```

Awesome! Now you know how to create Extended Views in Xamarin.Android.

Creating a Xamarin.iOS Theme

Creating Themes for Xamarin.iOS is very similar to doing so in native code. You can create the following Theme types in Xamarin.iOS:

- **Child Theme:** presents the same UI components as its parent Theme, but lets you change their appearance and position.
- **Extended Theme:** inherits its parent Theme's functionality and appearance, but lets you add to and modify both.

First, you'll learn how to create a Child Theme in Xamarin.iOS.

Child Theme

Child Themes leverage a parent Theme's behavior and UI components, letting you modify the appearance and position of those components. Note that you can't add or remove components, and the parent Theme must be a Full Theme. The Child Theme presents visual changes with its own XIB file and inherits the parent's View class.

Follow these steps to create a Child Theme in Xamarin.iOS:

1. In Visual Studio, create a new XIB file named after the Screenlet's View class and your Theme. By convention, an XIB file for a Screenlet with a View class named `LoginView` and a Theme named `demo` should be named `LoginView_demo`. You can use content from the parent Theme's XIB file as a foundation for your new XIB file. In your new XIB, you can change the UI components' visual properties (e.g., their position and size). You mustn't change, however, the XIB file's custom class, outlet connection, or `restorationIdentifier`. These must match those of the parent XIB file.

2. In the View Controller, set the Screenlet's `ThemeName` property to the Theme's name. For example, this sets Login Screenlet's `ThemeName` property to the `demo` Theme from the first step:

   ```
   this.loginScreenlet.ThemeName = "demo";
   ```

   This causes Liferay Screens to look for the file `LoginView_demo` in all apps' bundles. If that file doesn't exist, Screens uses the Default Theme instead (`LoginView_default`).

You can see an example of `LoginView_demo` in the Showcase-iOS demo app. Fantastic! Next, you'll learn how to create an Extended Theme.

Extended Theme

An Extended Theme inherits another Theme's UI components and behavior, but lets you add to or alter both. For example, you can extend the parent Theme's View class to change the parent Theme's behavior. You can also create a new XIB file that contains new or modified UI components. An Extended Theme's parent must be a Full Theme.

Follow these steps to create an Extended Theme:

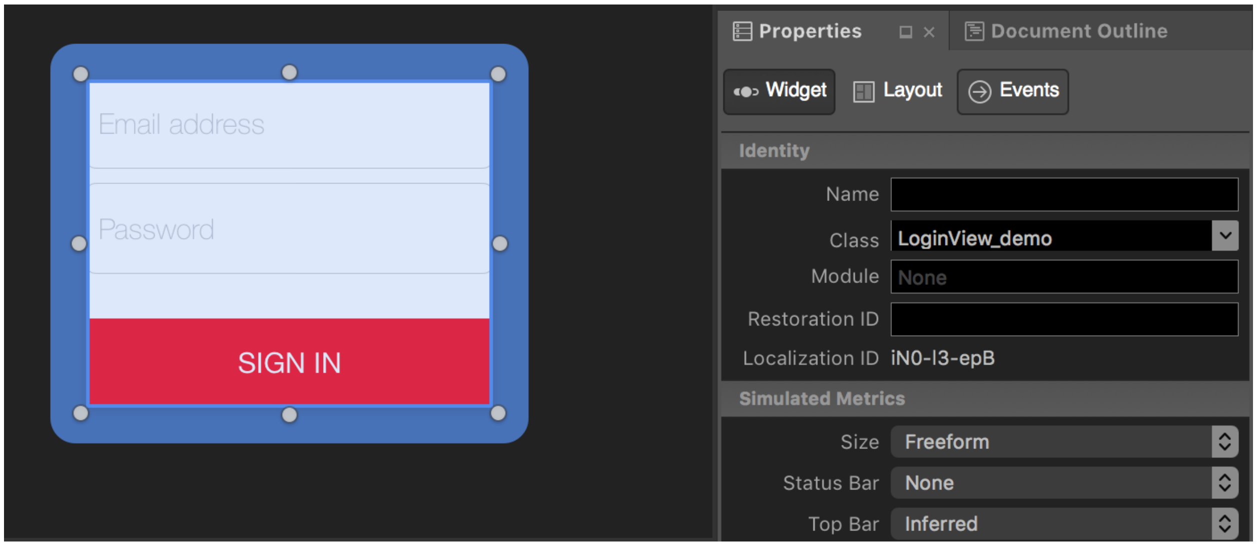
1. In Visual Studio, create a new XIB file named after the Screenlet's View class and your Theme. By convention, an XIB file for a Screenlet with a View class named `LoginView` and a Theme named `demo` should be named `LoginView_demo`. You can use the parent Theme's XIB file as a template. Make your Theme's UI changes by editing your XIB file in Visual Studio's iOS Designer or Xcode's Interface Builder.

2. Create a new View class that extends the parent Theme's View class. You should name this class after the XIB file you just created. You can add to or override functionality of the parent Theme's View class. Here's an example that extends the View class of Login Screenlet's default Theme (`LoginView_default`). Note that it changes the login button's background color and disables the progress presenter:

   ```
   using LiferayScreens;
   using System;

   namespace ShowcaseiOS
   {
       public partial class LoginView_demo : LoginView_default
       {
   ```

```
        public LoginView_demo (IntPtr handle) : base (handle) { }

        public override void OnCreated()
        {
            // You can change the login button color from code
            this.LoginButton.BackgroundColor = UIKit.UIColor.DarkGray;
        }

        // If you don't want a progress presenter, create an empty one
        public override IProgressPresenter CreateProgressPresenter()
        {
            return new NoneProgressPresenter();
        }
    }
}
```

3. Set your new View class as the custom class for your Theme's XIB file:



Name:
Class: LoginView_demo
Module: None
Restoration ID:
Localization ID: iN0-l3-epB
Size: Freeform
Status Bar: None
Top Bar: Inferred

Figure 78.2: Set new View class in XIB Theme file.

Well done! Now you know how to create an Extended Theme.

Related Topics

Creating Android Views (native code)

Creating iOS Themes (native code)

Preparing Xamarin Projects for Liferay Screens

Using Screenlets in Xamarin Apps

Using Views in Xamarin.Android

Using Themes in Xamarin.iOS

Liferay Screens for Xamarin Troubleshooting and FAQs

78.6 Liferay Screens for Xamarin Troubleshooting and FAQs

Even though Liferay developed Liferay Screens for Xamarin with great care, you may still run into some common issues. This tutorial lists tips and solutions for these issues, as well as answers to common questions about Screens for Xamarin.

General Troubleshooting

Before exploring specific issues, you should first make sure that you've installed the correct versions of Visual Studio and the Mono .NET framework. Each Screenlet's reference documentation (available for Android and iOS) lists these versions.

It may also help to investigate the sample Xamarin.Android and Xamarin.iOS apps developed by Liferay. Both are good examples of how to use Screenlets, Views (Android), and Themes (iOS):

- Showcase-Android
- Showcase-iOS

If you get stuck at any point, you can post your question on our forum. We're happy to assist you!

Common Issues

1. Build issues:

   Running *Clean* in Visual Studio may not be enough. Close Visual Studio, remove all the `bin` and `obj` folders that weren't removed by the clean, then rebuild your project.

2. `NSUnknownKeyException` error in Xamarin.iOS:

   This error occurs when Liferay Screens for iOS has a wrong module name in an XIB file. You must solve this in Xcode, removing the module name in the XIB file's *Custom Class* assignment in Interface Builder.

3. `The selector is already registered` error in Xamarin.iOS:

   This error occurs because one or more methods share the same name. To fix this, the binding file must be updated. Please file a ticket in our Jira or post the issue on our forum.

4. Xamarin.iOS crashes unexpectedly without any error messages in the console:

   Check the log file. On Mac OS, do this via the Console. On Windows, use the Event Viewer. In the app, you must click *User Reports* and then look for your app's name. Note that there may be more than one log file.

5. The app doesn't call delegate methods in Xamarin.iOS:

   When you implement the delegate methods in your view controller, make sure to annotate them with `[Export(...)]`. You must also set the view controller to the Screenlet instance's `Delegate` property. Here's an example of such a view controller that implements Login Screenlet's delegate, `ILoginScreenletDelegate`:

   ```
   public partial class ViewController : UIViewController, ILoginScreenletDelegate
   {
       protected ViewController(IntPtr handle) : base(handle) {}

       public override void ViewDidLoad()
   ```

```
    {
        base.ViewDidLoad();

        this.loginScreenlet.Delegate = this;
    }

    [Export("screenlet:onLoginResponseUserAttributes:")]
    public virtual void OnLoginResponseUserAttributes(BaseScreenlet screenlet,
        NSDictionary<NSString, NSObject> attributes)
    {
        ...
    }

    ...
}
```

DataType Mapping

For a better understanding of Xamarin code and example apps, see this list to compare type mapping between platforms. You must write Xamarin apps in C#, which has some differences compared to native code:

- Delegate (iOS) or listener (Android) classes:

  These classes are important because they listen for a Screenlet's events. In Liferay Screens for Xamarin, Screenlet delegates and listeners are prefixed with an I. For example, Login Screenlet's delegate in native code is `LoginScreenletDelegate`, while in Xamarin it's `ILoginScreenletDelegate`. Similarly, Login Screenlet's listener in native code is `LoginListener`, while in Xamarin it's `ILoginListener`. Use a similar naming scheme when you define a class/interface pair where the class is a standard implementation of the interface.

- Getter and setter methods:

  To get or set a value in native code, you use its getter and setter methods. In Liferay Screens for Xamarin, you should convert such methods to properties. If you have only one of these methods, you can call the method itself. For example:

```
// If you implemented a setter and a getter, call the property
loginScreenlet.Listener = this;

// Otherwise, call the method
loginScreenlet.getListener();
```

- Pascal case convention:

  C# code is usually written in Pascal case. However, you should use Camel case for protected instance fields or parameters.

Language Equivalents between Swift and C

- Protocols in Swift are analogous to interfaces in C#:

```
// Swift
protocol DoThings {
    func MyMethod() -> String
}


// C#
interface DoThings
```

```
{
    string MyMethod();
}
```

- Initializers in Swift are analogous to constructors in C#:

  ```
  // Swift
  class MyClass {
      var myVar : String = ""

      init(myVar : String) {
          self.myVar = myVar
      }
  }

  var testing = MyClass(myVar: "Test")


  // C#
  class MyClass {
      protected string myVar = "";

      public MyClass() {}

      public MyClass(string myVar) {
          this.myVar = myVar;
      }
  }

  var testing = new MyClass(myVar: "Test");
  ```

To learn more about language equivalents between Swift and C#, see this quick reference.

Language Equivalents between Java and C

To extend or implement a class or interface, Java requires that you use the `extends` or `implements` keywords. C# doesn't require this:

```
// Java
class Bird extends Vertebrate implements Actions {
    ...
}


// C#
class Bird : Vertebrate, Actions {
    ...
}
```

To learn more about language equivalents between Java and C#, see the C# for Java developers cheat sheet.

FAQs

1. Do I have to use Visual Studio?

   No, but we strongly recommend it. If you wish, however, you can use Xamarin Studio or Visual Studio Code instead.

2. What's the meaning of `[Export(...)]` above delegate method names?

   In short, this attribute makes properties and methods available in Objective-C. Xamarin's documentation explains this attribute in detail.

Related Topics

Preparing Xamarin Projects for Liferay Screens
    Using Screenlets in Xamarin Apps
    Using Views in Xamarin.Android
    Creating Xamarin Views and Themes
    Using Themes in Xamarin.iOS

Chapter 79

Mobile SDK

Want to wield Liferay's power in your mobile apps? Thanks to Liferay's Mobile SDK, you can do just that. The Liferay Mobile SDK provides a way to streamline consuming Liferay core web services, Liferay utilities, and custom app web services. It's a low-level layer that wraps Liferay JSON web services, making them easy to call in native mobile apps. It takes care of authentication, makes HTTP requests (synchronously or asynchronously), parses JSON results, and handles server-side exceptions so you can concentrate on *using* the services in your app. The Liferay Mobile SDK bridges the gap between your native app and Liferay services. The official project page gives you access to the SDK releases, provides the latest SDK news, and has forums for you to engage in mobile app development discussions. The Liferay Mobile SDK is available as separate downloads for Android and iOS.

There are two different types of Mobile SDKs that you need to add to your app's project, depending on the remote services you need to call. Liferay's prebuilt Mobile SDK includes the classes required to construct remote service calls in general. It also contains the classes required to call the specific remote services of Liferay's *core* portlets. Core portlets are included with every Liferay installation (these are also referred to as out-of-the-box or built-in portlets). However, you need to build an additional Mobile SDK if you want to leverage your custom portlet's remote services. Once built, this Mobile SDK contains *only* the classes required to call those services. Therefore, you must install it in your app alongside Liferay's prebuilt Mobile SDK to leverage your custom portlet's remote services.

Note that Liferay also provides Liferay Screens for constructing mobile apps that connect to Liferay. Screens uses components called *screenlets* to leverage and abstract the Mobile SDK's low-level service calls. However, if there's not a screenlet for your use case, or you need more control over the service call, then you may want to use the Mobile SDK directly. You should read the Screens tutorials in addition to the Mobile SDK tutorials here to decide which better fits your needs.

This section's tutorials cover using the Mobile SDK in Android and iOS app development. The following tutorials introduce these topics and are followed by in-depth tutorials on each:

- Creating Android Apps that Use the Mobile SDK
- Creating iOS Apps that Use the Mobile SDK

In addition, the following tutorial covers building Mobile SDKs to support your custom portlet services:

- Building Mobile SDKs

Fasten your seatbelt–it's time to go mobile with Liferay's Mobile SDK!

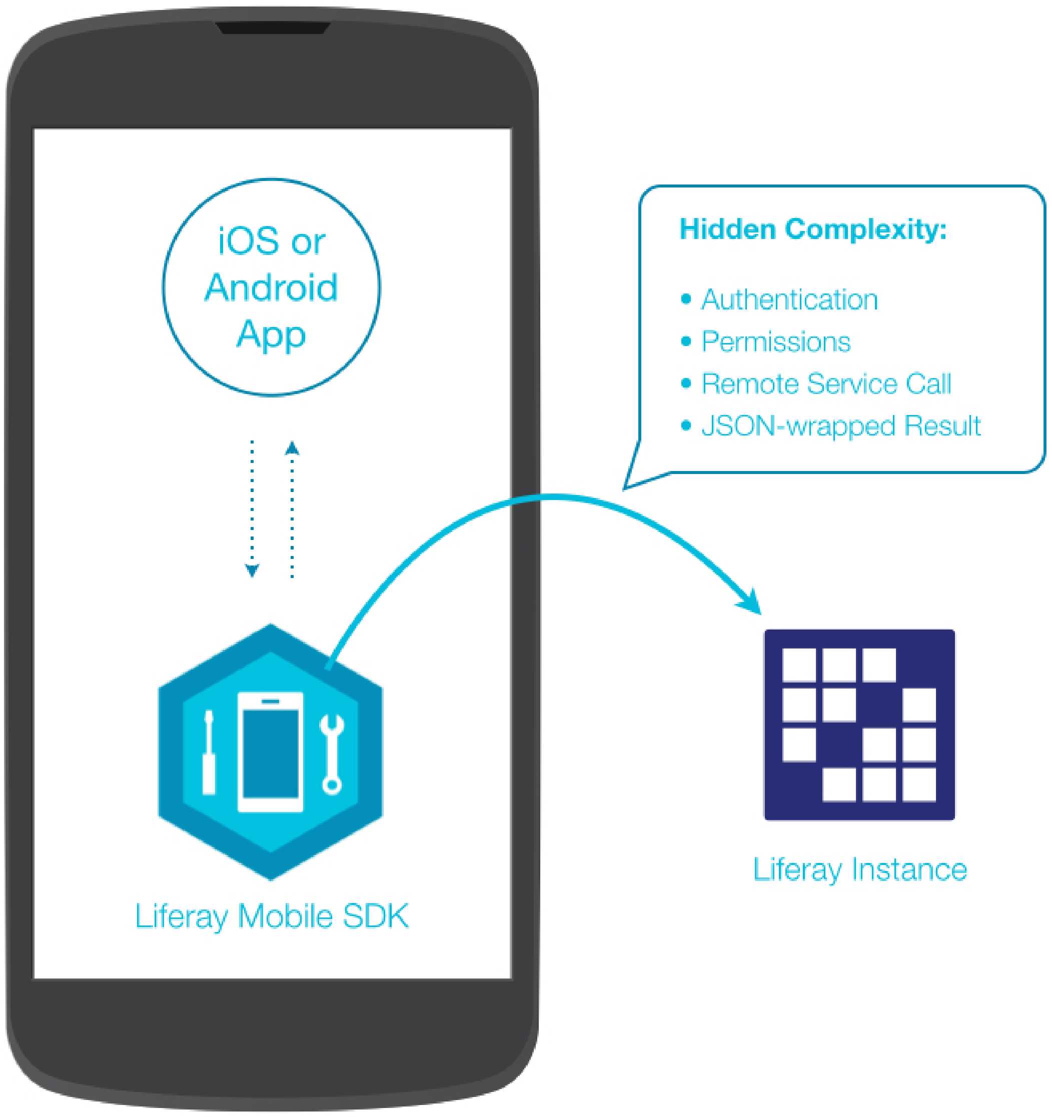


Figure 79.1: Liferay's Mobile SDK enables your native app to communicate with Liferay.

Related Topics:
Android Apps with Liferay Screens
iOS Apps with Liferay Screens
Building Mobile SDKs

79.1 Creating Android Apps that Use the Mobile SDK

The Liferay Mobile SDK provides a way to streamline the consumption of Liferay DXP's core web services and utilities, as well as those of custom apps. It wraps Liferay DXP's JSON web services, making them easy to call in native mobile apps. It handles authentication, makes HTTP requests (synchronously or asynchronously), parses JSON results, and handles server-side exceptions so you can concentrate on *using* the services in your app.

The Liferay Mobile SDK comes with the Liferay Android SDK. The official project page gives you access to the SDK releases, provides the latest SDK news, and has forums for you to engage in mobile app development discussions. Once you configure the Mobile SDK in your app, you can invoke Liferay DXP services in it.

The Android Mobile SDK app development tutorials cover these topics:

- Making Liferay and Custom Portlet Services Available in Your Android App
- Invoking Liferay Services in Your Android App
- Invoking Services Asynchronously from Your Android App
- Sending Your Android App's Requests Using Batch Processing

A great way to start is by setting up the Mobile SDK your Android project. This makes Liferay DXP's services available in your app.

Related Topics

Invoking Liferay Services in Your Android App
Creating iOS Apps that Use the Mobile SDK
Building Mobile SDKs

79.2 Making Liferay and Custom Portlet Services Available in Your Android App

You must install the correct Mobile SDKs in your Android project to call the remote services you need in your app. You should first install Liferay's prebuilt Mobile SDK. This is required for any app that leverages Liferay. To call your custom portlet's services, you also need to install the Mobile SDK that you built for it. For instructions on building a Mobile SDK for your custom portlet, see the tutorial Building Mobile SDKs.

This tutorial shows you how to install Liferay's prebuilt Mobile SDK, and any custom built Mobile SDKs. First, you'll learn how to use Gradle or Maven to install Liferay's prebuilt Mobile SDK. You'll then learn how to install a Mobile SDK manually, which is required for installing any custom built Mobile SDKs. Now go forth and fear no remote service!

Adding the SDK to Your Gradle Project

If your Android project is using Gradle as the build system, you can add Liferay's prebuilt Mobile SDK as a dependency to your project. All versions are available at the JCenter and Maven Central repositories. Both repositories are listed here, but you only need to have one in your app:

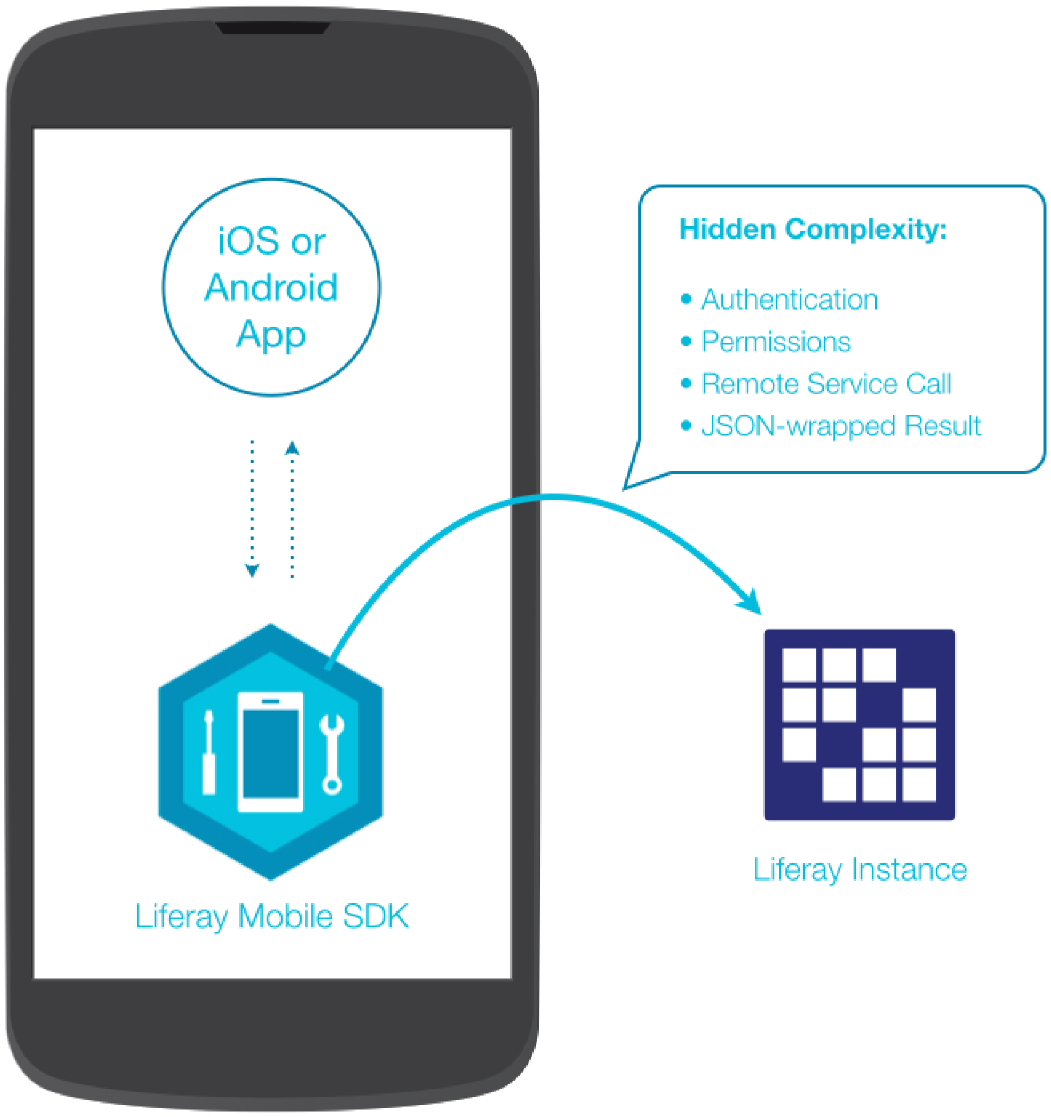


Figure 79.2: Liferay's Mobile SDK enables your native app to communicate with Liferay DXP.

```
repositories {
  jcenter()
  mavenCentral()
}

dependencies {
  compile group: 'com.liferay.mobile', name: 'liferay-android-sdk', version: '7.0.+'
}
```

If you get errors such as `Duplicate files copied in APK META-INF/NOTICE` when building with Gradle, add this to your `build.gradle` file:

```
android {
    ...
    packagingOptions {
        exclude 'META-INF/LICENSE'
        exclude 'META-INF/NOTICE'
    }
    ...
}
```

That's all there is to it! When your project syncs with your Gradle files, Liferay's prebuilt Mobile SDK downloads to your project. The instructions for doing this with Maven are shown next.

Adding the SDK to Your Maven Project

You can also add the Liferay's prebuilt Mobile SDK as a dependency to your project using Maven. To do so, add the following code to your `pom.xml` file:

```
<dependency>
    <groupId>com.liferay.mobile</groupId>
    <artifactId>liferay-android-sdk</artifactId>
    <version>LATEST</version>
</dependency>
```

Awesome! However, what if you're not using Gradle or Maven? What if you want to install a custom built Mobile SDK? No problem! The next section shows you how to install a Mobile SDK manually.

Manually Adding the SDK to Your Android Project

Use the following steps to manually set up a Mobile SDK in your Android project:

1. To install Liferay's prebuilt Mobile SDK, first download the latest version of `liferay-android-sdk-[version].jar`. If you built your own Mobile SDK, find its JAR file on your machine. This is detailed in the Building Mobile SDKs tutorial.

2. Copy the JAR into your Android project's `/libs` folder.

3. If you're manually installing Liferay's prebuilt Mobile SDK, you also need to download and copy these dependencies to your Android Project's `/libs` folder: `httpclient-android-4.3.3.jar` and `httpmime-4.3.3.jar`.

4. Start using it!

Great! Now you know how to manually install a Mobile SDK in your Android apps.

Making Custom Portlet Services Available in Your Android App

If you want to invoke remote web services for your custom portlet, then you need to generate its client libraries by building an Android Mobile SDK yourself. Building an SDK is covered in the tutorial Building Mobile SDKs. Once you build an SDK to a JAR file, you can install it using the manual installation steps above (make sure to use the JAR file you built instead of Liferay's prebuilt JAR file). Note that because your custom built SDKs contain *only* the client libraries for calling your custom portlet services, you must install them alongside Liferay's prebuilt SDK. Liferay's prebuilt SDK contains additional classes that are required to construct any remote service call.

Super! Now that the remote services you need are available in your app, you're ready to call them.

Related Topics

Invoking Liferay Services in Your Android App

Creating iOS Apps that Use the Mobile SDK

Building Mobile SDKs

79.3 Invoking Liferay Services in Your Android App

Once the appropriate Mobile SDKs are set up in your Android project, you can access and invoke Liferay DXP services in your app. This tutorial takes you through the steps you must follow to invoke these services:

1. Create a session.
2. Import the Liferay DXP services you need to call.
3. Create a service object and call the service methods.

Since some service calls require special treatment, this tutorial also shows you how to handle them. But first, you'll learn about securing Liferay DXP's JSON web services in the portal.

Securing JSON Web Services

The Liferay Mobile SDK calls Liferay DXP's JSON web services, which are enabled by default. The web services you call via the Mobile SDK must remain enabled for those calls to work. It's possible, however, to disable the web services that you don't need to call. For instructions on this, see the tutorial Configuring JSON Web Services. You can also use Service Access Policies for more fine-grained control over accessible services.

Step 1: Create a Session

A session is a conversion state between the client and server, that consists of multiple requests and responses between the two. You need a session to pass requests between your app and the Mobile SDK. In most cases, sessions need to be created with user authentication. The imports and code required to create a session are shown here:

```
import com.liferay.mobile.android.auth.basic.BasicAuthentication;
import com.liferay.mobile.android.service.Session;
import com.liferay.mobile.android.service.SessionImpl;
...
Session session = new SessionImpl("http://10.0.2.2:8080",
    new BasicAuthentication("test@example.com", "test"));
```

The arguments to `SessionImpl` are used to create the session. The first parameter is the URL of the Liferay instance you're connecting to. If you're running your app on Android Studio's emulator, `http://10.0.2.2:8080` is equivalent to `http://localhost:8080`. Be sure to replace this with the correct address for your server.

Warning: Be careful when using administrator credentials on a production Liferay instance, as you'll have permission to call any service. Make sure not to modify data by accident. Of course, the default administrator credentials should be disabled on a production Liferay instance.

The second parameter creates a new `BasicAuthentication` object containing the user's credentials. Depending on the authentication method used by your Liferay instance, you need to provide the user's email address, screen name, or user ID. You also need to provide the user's password. The `BasicAuthentication` object tells the session to use Basic Authentication to authenticate each service call. The Mobile SDK also supports OAuth authentication, as long as the OAuth Provider portlet is deployed to your Liferay instance. To learn how to use OAuth authentication with the Mobile SDK, see the OAuth sample app. Also, note that the OAuth Provider portlet is only available to customers with an active Liferay subscription.

If you're building a sign in view for your app, you can use the `SignIn` utility class to check if the credentials given by the user are valid.

```
import com.liferay.mobile.android.auth.SignIn;
...
SignIn.signIn(session, new JSONObjectAsyncTaskCallback() {

    @Override
    public void onSuccess(JSONObject userJSONObject) {
        System.out.println("Successful sign-in, user details: " + userJSONObject)
    }

    @Override
    public void onFailure(Exception e) {
        e.printStackTrace();
    }

});
```

Note that the Mobile SDK doesn't keep a persistent connection or session with the server. Each request is sent with the user's credentials (except when using OAuth). However, the `SignIn` class provides a way to return user information after a successful sign-in.

Next, you're shown how to create an unauthenticated session in the limited cases where this is possible.

Creating an Unauthenticated Session

In some cases, it's possible to create a `Session` instance without user credentials. However, most Liferay remote methods don't accept unauthenticated remote calls. Making a call with an unauthenticated session generates an `Authentication access required` exception in most cases.

Unauthenticated service calls only work if the remote method in the Liferay instance or your plugin has the `@AccessControlled` annotation. This is shown here for the hypothetical class `FooServiceImpl` and its method `bar`:

```
import com.liferay.portal.security.ac.AccessControlled;
...
public class FooServiceImpl extends FooServiceBaseImpl {
...
    @AccessControlled(guestAccessEnabled = true)
    public void bar() { ... }
...
```

To make such a call, you need to use the constructor that accepts the server URL only:

```
Session session = new SessionImpl("http://10.0.2.2:8080");
```

Fantastic! Now that you have a session, you can use it to call Liferay's services.

Step 2: Import the Liferay Services You Need

First, you should determine the Liferay services you need to call. You can find the available services at `http://localhost:8080/api/jsonws`. Be sure to replace `http://localhost:8080` in this URL with your server's address.

Add the imports for the services you need to call. For example, if you're building a blogs app, you can import `BlogsEntryService`:

```
import com.liferay.mobile.android.v62.blogsentry.BlogsEntryService;
```

Note that the Liferay version (`.v62`) is used in the package namespace. Since the Mobile SDK is built for a specific Liferay version, service classes for different Liferay versions are separated by their package names. In this example, the Mobile SDK classes use the `.v62` package, which means this Mobile SDK is compatible with Liferay 6.2. Mobile SDK classes compatible with Liferay 7.0 use the `v7` package. This namespacing lets your app support multiple Liferay versions.

Step 3: Create a Service Object and Call its Service Methods

Once you have a session and the required imports, you're ready to make the service call. This is done by creating a service object for the service you want to call, and then calling its service methods. For example, if you're creating a blogs app, you need to use `BlogsEntryService` to get all the blogs entries from a site. This is demonstrated by the following code:

```
BlogsEntryService service = new BlogsEntryService(session);

JSONArray jsonArray = service.getGroupEntries(10184, 0, 0, -1, -1);
```

This fetches all blog entries from the *Guest* site. In this example, the *Guest* site's `groupId` is `10184`. Note that many service methods require `groupId` as a parameter. You can get the user's groups by calling the `getUserSites()` method from `GroupService`.

Service method return types can be `void`, `String`, `JSONArray`, or `JSONObject`. Primitive type wrappers can be `Boolean`, `Integer`, `Long`, or `Double`.

This `BlogsEntryService` call is a basic example of a synchronous service call; the method only returns after the request finishes. However, Android doesn't allow network communication from an app's main UI thread. Service calls issued from the main UI thread need need to be asynchronous. For instructions on doing this, see the tutorial Invoking Services Asynchronously from Your Android App.

Great! Now you're familiar with the basics of accessing Liferay services through the Mobile SDK. However, there are some special cases you may run into when making service calls from your app. These are discussed in the following sections.

Non-Primitive Arguments

There are some special cases in which a service method's arguments aren't primitives. In these cases, you should use JSONObjectWrapper. For example:

```
JSONObjectWrapper wrapper = new JSONObjectWrapper(new JSONObject());
```

You must pass a JSON containing the object properties and their values. On the server side, your object is instantiated and setters for each property are called with the values from the JSON you passed.

There are other cases in which service methods require interfaces or abstract classes as arguments. Since it's impossible for the SDK to guess which implementation you want to use, you must initialize JSONObjectWrapper with the class name. For example:

```
JSONObjectWrapper wrapper = new JSONObjectWrapper(className, new JSONObject());
```

The server looks for the class name in its classpath and instantiates the object for you. It then calls setters, as in the previous example. The abstract class OrderByComparator is a good example of this. This is discussed next.

OrderByComparator

On the server side, OrderByComparator is an abstract class. You must therefore pass the name of a class that implements it. For example:

```
String className = "com.liferay.portlet.bookmarks.util.comparator.EntryNameComparator";

JSONObjectWrapper orderByComparator = new JSONObjectWrapper(className, new JSONObject());
```

If the service you're calling accepts null for a comparator argument, pass null to the service call.

You may want to set the ascending property for a comparator. Unfortunately, as of Liferay 6.2, most Liferay OrderByComparator implementations don't have a setter for this property and it isn't possible to set from the Mobile SDK. Future Liferay versions may address this. However, you may have a custom OrderByComparator that has a setter for ascending. In this case, you can use the following code:

```
String className = "com.example.MyOrderByComparator";

JSONObject jsonObject = new JSONObject();
jsonObject.put("ascending", true);

JSONObjectWrapper orderByComparator = new JSONObjectWrapper(className, jsonObject);
```

For more examples, see the test case OrderByComparatorTest.java.

ServiceContext

Another non-primitive argument is ServiceContext. It requires special attention because most Liferay service methods require it. However, you aren't required to pass it to the SDK; you can pass null instead. The server then creates a ServiceContext instance for you, using default values.

If you need to set properties for ServiceContext, you can do so by adding them to a new JSONObject and then passing it as the ServiceContext argument:

```
JSONObject jsonObject = new JSONObject();
jsonObject.put("addGroupPermissions", true);
jsonObject.put("addGuestPermissions", true);

JSONObjectWrapper serviceContext = new JSONObjectWrapper(jsonObject);
```

For more examples, see the test case ServiceContextTest.java.

Binaries

Some Liferay services require argument types such as byte arrays (`byte[]`) and Files (`java.io.File`).

The Mobile SDK converts byte arrays to strings before sending the POST request. For example, `"hello".getBytes("UTF-8")` becomes a JSON array such as `"[104,101,108,108,111]"`. The Mobile SDK does this for you so you don't have worry about it; you only need to pass the byte array to the method.

However, you need to be careful when using such methods. This is because you're allocating memory for the whole byte array, which may cause memory issues if the content is large.

Other Liferay service methods require `java.io.File` as an argument. In these cases, the Mobile SDK requires `InputStreamBody` instead. To accomodate this, you need to create an `InputStream` and pass it to the `InputStreamBody` constructor, along with the file's mime type and name. For example:

```
InputStream is = context.getAssets().open("file.png");
InputStreamBody file = new InputStreamBody(is, "image/png", "file.png");
```

The Mobile SDK sends a multipart form request to the Liferay instance. On the server side, a `File` instance is created and sent to the service method you're calling.

It's also possible to cancel or monitor service calls that upload data to Liferay. Every service that uploads data returns an `AsyncTask` instance. You can use it to cancel the upload if needed. If want to listen for upload progress to create a progress bar, you can create an `UploadProgressAsyncTaskCallback` callback and set it to the current `Session` object. Its `onProgress` method is called for each byte chunk sent. It passes the total number of uploaded bytes so far. For example:

```
session.setCallback(

    new UploadProgressAsyncTaskCallback<JSONObject>() {

        (...)

        public void onProgress(int totalBytes) {
            // This method will be called for each byte chunk sent.
            // The totalBytes argument will contain the total number
            // of uploaded bytes, from 0 to the total size of the
            // request.
        }

        (...)

    }
);
```

For more examples on this subject, see the `addFileEntry*` methods in `DLAppServiceTest.java`

As you can see, the Mobile SDK does a great deal of work for you even when special service method arguments are required.

Related Topics

Invoking Services Asynchronously from Your Android App

Building Mobile SDKs

Creating iOS Apps that Use the Mobile SDK

79.4 Invoking Services Asynchronously from Your Android App

Android doesn't allow synchronous HTTP requests to be made from the main UI thread. You can use Android's `AsyncTask` to make synchronous requests from threads other than the main UI thread. If you don't want to use

AsyncTask, you can make asynchronous requests through the Mobile SDK. To do so, you need to implement and instantiate a callback class, and then set it to the session. When the Mobile SDK makes your service calls for that session, it then makes them asynchronously. To make synchronous calls again, set null as the session's callback.

With the following steps, this tutorial shows you how to implement asynchronous requests in your Android app:

1. Implement and instantiate your callback class.
2. Set the callback on the session.
3. Call Liferay services.

Now go ahead and get started!

Implementing and Instantiating Your Callback Class

Before implementing and instantiating your callback class, you should add the required imports. The imports you add depend on the return type of the service method you're calling. For example, if you need to call the service method getGroupEntries to retrieve blog entries from a site's Blogs portlet, you need to import the Mobile SDK's AsyncTaskCallback and JSONArrayAsyncTaskCallback:

```
import com.liferay.mobile.android.task.callback.AsyncTaskCallback;
import com.liferay.mobile.android.task.callback.typed.JSONArrayAsyncTaskCallback;
```

This is because the getGroupEntries returns a JSONArray. There are multiple AsyncTaskCallback implementations, one for each method return type:

- JSONObjectAsyncTaskCallback
- JSONArrayAsyncTackCallback
- StringAsyncTaskCallback
- BooleanAsyncTaskCallback
- IntegerAsyncTaskCallback
- LongAsyncTaskCallback
- DoubleAsyncTaskCallback

It's also possible to use a generic AsyncTaskCallback implementation called GenericAsyncTaskCallback. To do so, you must implement a transform method and handle JSON parsing yourself.

If you still don't want to use any of these callbacks, you can implement AsyncTaskCallback directly. However, you should be careful when doing so. You should always get the first element of the JSONArray passed as a parameter to the onPostExecute(JSONArray jsonArray) method (for example, jsonArray.get(0)).

Next, implement and instantiate your callback class. When implementing your callback class, you need to implement its onFailure and onSuccess methods. These methods respectivley determine what your app does when the request fails or succeeds. The onFailure() method is called if an exception occurs during the request. This could be triggered by a connection exception (e.g., a request timeout) or a ServerException. If a ServerException occurs, it's because something went wrong on the server side. For example, if you pass a groupId that doesn't exist, the Liferay instance complains about it, and the Mobile SDK wraps the error message with ServerException.

The onSuccess method is called on the main UI thread after the request finishes. Since the request is asynchronous, the service call immediately returns a null object. The service delivers the service's real return value to the callback's onSuccess() method, instead.

Example code is shown here for AsyncTaskCallback and JSONArrayAsyncTaskCallback:

```
AsyncTaskCallback callback = new JSONArrayAsyncTaskCallback() {

    public void onFailure(Exception exception) {
        // Implement exception handling code
    }

    public void onSuccess(JSONArray result) {
        // Called after request has finished successfully
    }

};
```

Now that you have your callback class, you can set it to the session.

Setting the Callback to the Session

Once you've implemented and instantiated your callback class, you're ready to set it to the session. If you haven't created a session yet, do so now. The tutorial Invoking Liferay Services in Your Android App shows you how to create a session. Now you're ready to set the callback to the session. For example, this is done here for `AsyncTaskCallback`:

```
session.setCallback(callback);
```

Pretty simple! Now you're ready to make the service call.

Making the Service Call

Last but certainly not least, make the service call. This is done the same as calling any other service: create a service object from the session and use it to make the service call. This is also described in the tutorial Invoking Liferay Services in Your Android App. An example service call that gets all the blog entries from a site's Blogs portlet is shown here:

```
service.getGroupEntries(10184, 0, 0, -1, -1);
```

The example code from the above sections is shown together here:

```
import com.liferay.mobile.android.task.callback.AsyncTaskCallback;
import com.liferay.mobile.android.task.callback.typed.JSONArrayAsyncTaskCallback;

...

AsyncTaskCallback callback = new JSONArrayAsyncTaskCallback() {

    public void onFailure(Exception exception) {
        // Implement exception handling code
    }

    public void onSuccess(JSONArray result) {
        // Called after request has finished successfully
    }

};

// create a session first
session.setCallback(callback);

// create a service object first
service.getGroupEntries(10184, 0, 0, -1, -1);
```

Great! Now you know how to invoke services asynchronously from your Android app.

Related Topics

Creating iOS Apps that Use the Mobile SDK
Building Mobile SDKs

79.5 Sending Your Android App's Requests Using Batch Processing

The Mobile SDK also allows sending requests in batch. This can be much more efficient than sending separate requests. For example, suppose you want to delete ten blog entries in a site's Blogs portlet at the same time. Instead of making a request for each deletion, you can create a batch of calls and send them all together.

This tutorial shows you how to implement batch processing for your Android app. It's assumed that you already know how to invoke Liferay services from your Android app. If you don't, see the tutorial Invoking Liferay Services in Your Android App. Now get ready to whip up a fresh batch of service calls!

Implementing Batch Processing

Making service calls in batch only requires two extra steps over making them one at a time:

- Create a batch session with `BatchSessionImpl`.
- Make the batch service calls with the `invoke` method of `BatchSessionImpl`.

The rest of the steps are the same as making other service calls. You still need a service object, and you still need to call its service methods. As an example, here's a code snippet from an app that deletes a Blogs portlet's blog entries synchronously in batch:

```
import com.liferay.mobile.android.service.BatchSessionImpl;

BatchSessionImpl batch = new BatchSessionImpl(session);
BlogsEntryService service = new BlogsEntryService(batch);

service.deleteEntry(1);
service.deleteEntry(2);
service.deleteEntry(3);

JSONArray jsonArray = batch.invoke();
```

So what's going on here? After the import, `BatchSessionImpl` is used with a pre-existing session to create a batch session. Note that the `BatchSessionImpl` constructor takes either credentials or a session. Passing a session to the constructor is useful when you already have a `Session` object and want to reuse the same credentials. After creating the service object, several `deleteEntry` service calls are created. Since the service object is created with a batch session, these calls aren't made immediately; they return `null` instead. The calls aren't made until issued in batch by calling the `invoke()` method on the batch session object. It returns a `JSONArray` containing the results for each service call. Since this example contains three `deleteEntry` calls, the `jsonArray` contains three objects. The results are ordered the same as the service calls.

Great! But what if you want to make batch calls asynchronously? No problem! Set the callback as a `BatchAsyncTaskCallback` instance:

```
import com.liferay.mobile.android.task.callback.BatchAsyncTaskCallback;

batch.setCallback(new BatchAsyncTaskCallback() {

    public void onFailure(Exception exception) {
    }
    public void onSuccess(JSONArray results) {
        // The result is always a JSONArray
    }

});
```

This is similar to the procedure for making asynchronous calls as described in the tutorial Invoking Services Asynchronously from Your Android App. Awesome! Now you know how to make efficient service calls in batch!

Related Topics

Invoking Liferay Services in Your Android App
Invoking Services Asynchronously from Your Android App
Creating iOS Apps that Use the Mobile SDK

79.6 Creating iOS Apps that Use the Mobile SDK

The Liferay Mobile SDK provides a way to streamline the consumption of Liferay DXP's core web services and utilities, as well as those of custom apps. It wraps Liferay DXP's JSON web services, making them easy to call in native mobile apps. It handles authentication, makes HTTP requests (synchronously or asynchronously), parses JSON results, and handles server-side exceptions so you can concentrate on *using* the services in your app.

The Liferay Mobile SDK comes with the Liferay iOS SDK. The official project page gives you access to the SDK releases, provides the latest SDK news, and has forums for you to engage in mobile app development discussions. Once you configure the Mobile SDK in your app, you can invoke Liferay DXP services in it.

The iOS Mobile SDK app development tutorials cover these topics:

- Making Liferay and Custom Portlet Services Available in Your iOS App
- Invoking Liferay Services in Your iOS App
- Invoking Services Asynchronously from Your iOS App
- Sending Your iOS App's Requests Using Batch Processing

A great way to start is by setting up the Mobile SDK in your iOS project. This makes Liferay DXP's services available in your app.

Related Topics

Invoking Liferay Services in Your iOS App
Building Mobile SDKs
Creating Android Apps that Use the Mobile SDK

79.7 Making Liferay and Custom Portlet Services Available in Your iOS App

Your iOS app is no doubt pretty great, or at least off to a great start. Now you want it to access Liferay services. How do you accomplish this? Use Liferay's iOS Mobile SDK, of course! You must install the correct Mobile SDKs in your iOS project to call the remote services you need in your app. You should first install Liferay's prebuilt Mobile SDK. This is required for any app that leverages Liferay. To call your custom portlet's services,

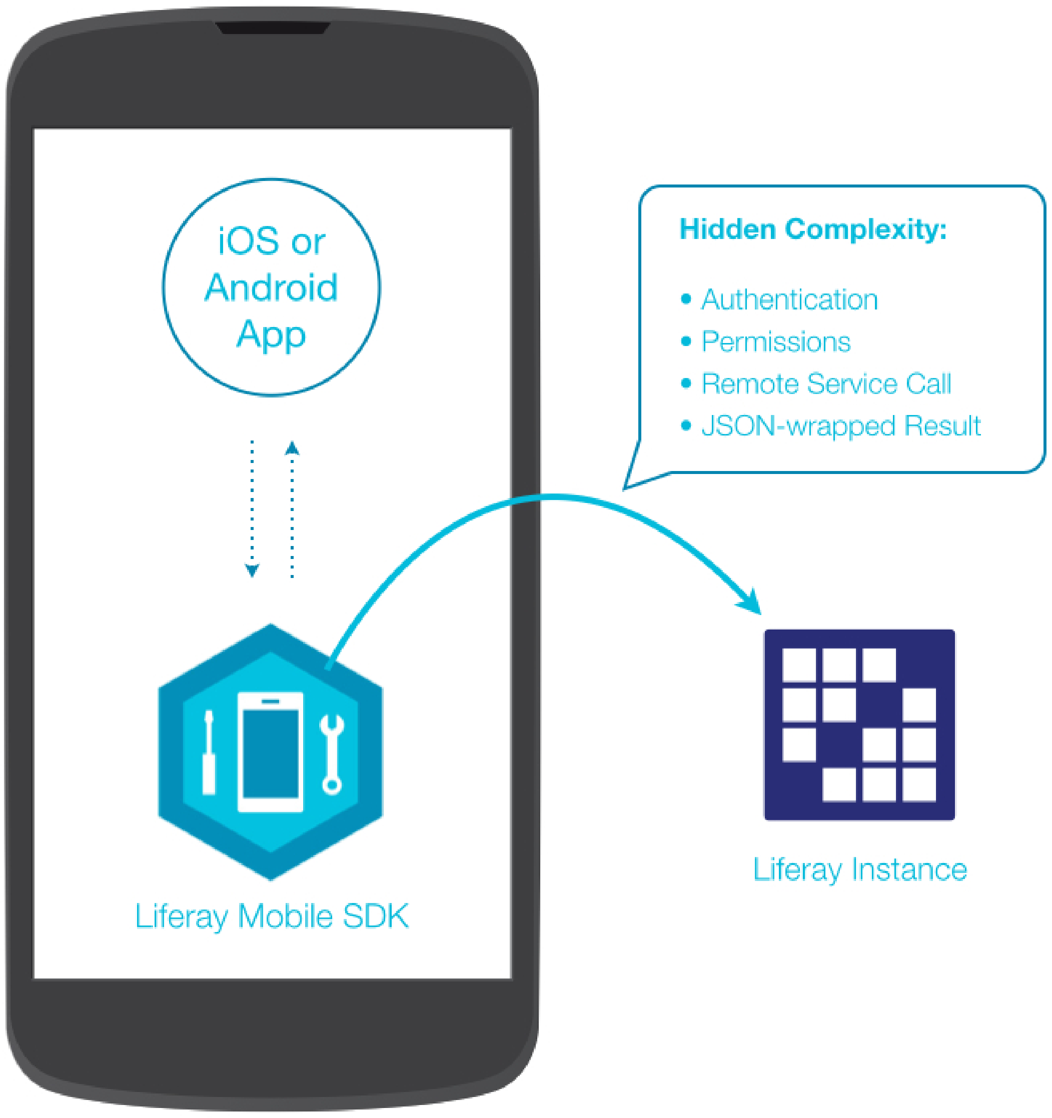


Figure 79.3: Liferay's Mobile SDK enables your native app to communicate with Liferay DXP.

you also need to install the Mobile SDK that you built for it. For instructions on building a Mobile SDK for your custom portlet, see the tutorial Building Mobile SDKs.

This tutorial shows you how to install Liferay's prebuilt Mobile SDK, and any custom built Mobile SDKs. First, you'll learn how to use CocoaPods to install Liferay's prebuilt Mobile SDK. You'll then learn how to install a Mobile SDK manually, which is required for installing any custom built Mobile SDKs. Now go forth and fear no remote service!

Installing the SDK Using CocoaPods

Using CocoaPods is the simplest way to install Liferay's prebuilt Mobile SDK. The steps for doing so are shown here:

1. Make sure you have CocoaPods installed.
2. Create a file called `Podfile` in your project. Add the following line in this file:

   ```
   pod 'Liferay-iOS-SDK'
   ```

3. Run `pod install` from your project's directory. This downloads the latest version of the Liferay iOS Mobile SDK and creates a `.xcworkspace` file. CocoaPods also downloads all the necessary dependencies and configures your workspace. Note that you may have to run `pod repo update` before running `pod install`; this ensures you have the latest version of the CocoaPods repository on your machine.
4. Use the `.xcworkspace` file to open your project in Xcode.
5. If you're importing dependencies as frameworks (`use_frameworks!` in your Podfile), you need to import the `LRMobileSDK` module:

   ```
   @import LRMobileSDK; // (Objective-C)
   import LRMobileSDK // (Swift)
   ```

For more information on how CocoaPods works, see their documentation. Next, you'll learn how to install a Mobile SDK manually.

Installing an iOS SDK Manually

You can also install Mobile SDKs manually. This is required if you built one for your custom portlet's services. You can also install install Liferay's prebuilt Mobile SDK manually if you don't want to use CocoaPods.

1. To install Liferay's prebuilt Mobile SDK, first download the latest version of the Liferay iOS Mobile SDK ZIP file. If you built your own Mobile SDK, find its ZIP file on your machine. This is detailed in the Building Mobile SDKs tutorial.
2. Unzip the file into your Xcode project.
3. Within Xcode, right-click on your project and click *Add Files to 'Project Name'*.
4. Add the `core` and `v7` folders. Note the `v7` folder's name can change for each Liferay version. In this example, the SDK is built for Liferay 7.0.
5. If you're manually installing Liferay's prebuilt Mobile SDK, it also requires AFNetworking 2.6.3. Add its source code to your project.

Great! Now you know how to manually install a Mobile SDK in your iOS apps.

Understanding Liferay and iOS Compatibility

Each Liferay Mobile SDK is designed to work with a specific Liferay version. The Liferay Mobile SDK version number reflects this. The first two digits of each Mobile SDK's version number correspond to the compatible Liferay version. For example, a Mobile SDK version `6.2.*` is compatible with Liferay 6.2, while a Mobile SDK version `7.0.*` is compatible with Liferay 7.0. Any digits after the first two correspond to the internal Liferay Mobile SDK build.

The Mobile SDK's service class names are also suffixed with the Mobile SDK's version number. This lets your app support several Liferay versions. For example, you can add Mobile SDK versions `6.2.0.22` and `7.0.3` to the same project. The Mobile SDK service classes supporting Liferay versions 6.2 and 7.0 end in `_v62.m` and `_v7.m`, respectively. To find out the Liferay versions your app connects to, use the `[LRPortalVersionUtil getPortalVersion:...]` method.

The Liferay iOS Mobile SDK is compatible with iOS versions 7.0 and up. Older iOS versions may work, but compatibility is untested.

Making Custom Portlet Services Available in Your iOS App

If you want to invoke remote web services for your custom portlet, then you need to generate its client libraries by building an iOS Mobile SDK yourself. Building an SDK is covered in the tutorial Building Mobile SDKs. Once you build an SDK to a ZIP file, you can install it using the manual installation steps above (make sure to use the ZIP file you built instead of Liferay's prebuilt ZIP file). Note that because your custom built SDKs contain *only* the client libraries for calling your custom portlet services, you must install them alongside Liferay's prebuilt SDK. Liferay's prebuilt SDK contains additional classes that are required to construct any remote service call.

Related Topics

Building Mobile SDKs

Creating Android Apps that Use the Mobile SDK

79.8 Invoking Liferay Services in Your iOS App

Once the appropriate Mobile SDKs are set up in your iOS project, you can access and invoke Liferay DXP services in your app. This tutorial takes you through the steps you must follow to invoke these services:

1. Create a session.
2. Import the Liferay DXP services you need to call.
3. Create a service object and call the service methods.

Since some service calls require special treatment, this tutorial also shows you how to handle them. Note that the code snippets in this tutorial are written in Objective-C.

First, you'll learn about securing Liferay DXP's JSON web services in the portal.

Securing JSON Web Services

The Liferay Mobile SDK calls Liferay DXP's JSON web services, which are enabled by default. The web services you call via the Mobile SDK must remain enabled for those calls to work. It's possible, however, to disable the web services that you don't need to call. For instructions on this, see the tutorial Configuring JSON Web Services. You can also use Service Access Policies for more fine-grained control over accessible services.

Step 1: Create a Session

A session is a conversion state between the client and server, consisting of multiple requests and responses between the two. You need a session to pass requests between your app and the Mobile SDK. In most cases, sessions need to be created with user authentication. The imports and code required to create a session are shown here:

```
#import "LRBasicAuthentication.h"
#import "LRSession.h"

LRSession *session = [[LRSession alloc] initWithServer:@"http://localhost:8080"
    authentication:[[LRBasicAuthentication alloc] initWithUsername:@"test@example.com" password:@"test"]];
```

The `LRSession` object is created with initializers specifying the Liferay instance to connect to and the credentials of the user to authenticate. The `initWithServer` parameter sets the URL of the Liferay instance you're connecting to. In this case, the Liferay instance is running on `http://localhost:8080`. The iOS emulator is also running on the same machine. Next, the `authentication` parameter takes an `LRBasicAuthentication` instance with the credentials of the user to authenticate. Depending on the authentication method used by your Liferay instance, you need to provide the user's email address, screen name, or user ID to the `initWithUsername` parameter. You also need to provide the user's password to the `password` parameter.

Using `LRBasicAuthentication` tells the session to authenticate each service call with Basic Authentication. The Mobile SDK also supports OAuth authentication, as long as the OAuth Provider app is deployed to your Liferay DXP instance. To learn how to use OAuth authentication with the Mobile SDK, see the OAuth sample app. Also, note that the OAuth Provider app is only available to customers with an active Liferay subscription.

Warning: Be careful when using administrator credentials on a production portal instance, as you'll have permission to call any service. Make sure not to modify data accidentally. Of course, the default administrator credentials should be disabled on a production portal instance.

If you're building a sign in view for your app, you can use the `LRSignIn` utility class to check if the credentials given by the user are valid:

```
#import "LRSignIn.h"

[session
    onSuccess:^(id result) {
        user = result;
        [monitor signal];
    }
    onFailure:^(NSError *e) {
        error = e;
        [monitor signal];
    }
];

[LRSignIn signInWithSession:session callback:session.callback error:&error];
```

The Mobile SDK doesn't keep a persistent connection or session with the server. Each request is sent with the user's credentials (except when using OAuth). However, the `SignIn` class provides a way to return user information after a successful sign-in.

You can persist credentials with `LRCredentialStorage`. It safely saves the username and password in the keychain:

```
[LRCredentialStorage storeCredentialForServer:@"http://localhost:8080"
    username:@"test@example.com" password:@"test"];
```

After credentials are stored, you can retrieve them with:

```
NSURLCredential *credential = [LRCredentialStorage getCredential];
```

Alternatively, you can create an LRSession instance directly with:

```
LRSession *session = [LRCredentialStorage getSession];
```

For more examples of this, see CredentialStorageTest.m.

Next, you're shown how to create an unauthenticated session in the limited cases where this is possible.

Creating an Unauthenticated Session

In some cases, it's possible to create an LRSession instance without user credentials. However, most Liferay remote methods don't accept unauthenticated remote calls. Making a call with an unauthenticated session generates an Authentication access required exception in most cases.

Unauthenticated service calls only work if the remote method in the Liferay instance or your plugin has the @AccessControlled annotation. This is shown here for the hypothetical class FooServiceImpl and its method bar:

```
import com.liferay.portal.security.ac.AccessControlled;
...
public class FooServiceImpl extends FooServiceBaseImpl {
...
    @AccessControlled(guestAccessEnabled = true)
    public void bar() { ... }
...
```

To make such a call, you need to use the constructor that accepts the server URL only:

```
LRSession *session = [[LRSession alloc] initWithServer:@"http://localhost:8080"];
```

Fantastic! Now that you have a session, you can use it to call Liferay's services.

Step 2: Import the Service

First, you should determine the Liferay services you need to call. You can find the available services at http://localhost:8080/api/jsonws. Be sure to replace http://localhost:8080 in this URL with your server's address.

Once you determine the services you need to call, add their imports. For example, if you're building a blogs app, you can import LRBlogsEntryService:

```
#import "LRBlogsEntryService_v62.h"
```

Note that the Liferay version (_v62) is used in the service class's name. This corresponds to the Liferay version it's compatible with. In this example, _v62 is used, which means this Mobile SDK class is compatible with Liferay 6.2. Mobile SDK classes compatible with Liferay 7.0 use _v7 instead. Because service class names contain the Liferay version they're compatible with, you can use several Mobile SDKs simultaneously to support different Liferay versions in the same app.

Step 3: Calling the Service

Once you have a session and have imported the service class, you're ready to make the service call. This is done by creating a service object for the service you want to call, and then calling its service methods. For example, if you're creating a blogs app, you need to use `LRBlogsEntryService` to get all the blogs entries from a site. This is demonstrated by the following code.

```
LRBlogsEntryService_v62 *service = [[LRBlogsEntryService_v62 alloc] initWithSession:session];

NSError *error;
NSArray *entries = [service getGroupEntriesWithGroupId:10184 status:0 start:-1 end:-1 error:&error];
```

This fetches all blog entries from the *Guest* site. In this example, the *Guest* site's `groupId` is `10184`. Note that many service methods require `groupId` as a parameter. You can get the user's groups by calling `[LRGroupService_v62 getUserSites:&error]`.

Service method return types can be `void`, `NSString`, `NSArray`, `NSDictionary`, `NSNumber`, and `BOOL`.

This `LRBlogsEntryService` call is a basic example of a synchronous service call. The method in a synchronous service call returns only after the request is finished.

Non-Primitive Arguments

There are some special cases in which service method arguments aren't primitives. In these cases, you should use `LRJSONObjectWrapper`. For example:

```
LRJSONObjectWrapper *wrapper = [[LRJSONObjectWrapper alloc]
    initWithJSONObject:[NSDictionary dictionary]];
```

You must pass a dictionary containing the object's properties and their values. On the server side, your object is instantiated and setters for each property are called with the values from the dictionary.

There are some other cases in which service methods require interfaces or abstract classes as arguments. Since it's impossible for the SDK to guess which implementation you want to use, you must initialize `LRJSONObjectWrapper` with the class name. For example:

```
LRJSONObjectWrapper *wrapper = [[LRJSONObjectWrapper alloc]
    initWithClassName:@"com.example.MyClass" jsonObject:[NSDictionary dictionary]];
```

The server looks for the class name in its classpath and instantiates the object for you. It then calls setters, as in the previous example. The abstract class `OrderByComparator` is a good example of this. This is discussed next.

OrderByComparator

On the server side, `OrderByComparator` is an abstract class. You must therefore pass the name of a class that implements it. For example:

```
NSString *className = @"com.liferay.portlet.bookmarks.util.comparator.EntryNameComparator";

LRJSONObjectWrapper *orderByComparator = [[LRJSONObjectWrapper alloc] initWithClassName:className jsonObject:[NSDictionary dictionary]];
```

If the service you're calling accepts `null` for a comparator argument, pass `nil` to the service call.

You may want to set the ascending property for a comparator. Unfortunately, as of Liferay 6.2, most Liferay `OrderByComparator` implementations don't have a setter for this property and it isn't possible to set from the Mobile SDK. Future Liferay versions may address this. However, you may have a custom `OrderByComparator` that has a setter for ascending. In this case, you can use the following code:

```
NSString *className = @"com.example.MyOrderByComparator";

NSDictionary *jsonObject = @{
    @"ascending": @(YES)
};

LRJSONObjectWrapper *orderByComparator = [[LRJSONObjectWrapper alloc]
    initWithClassName:className jsonObject:jsonObject];
```

For more examples, see the test case `OrderByComparatorTest.m`.

ServiceContext

Another non-primitive argument is `ServiceContext`. It requires special attention because most Liferay service methods require it. However, you aren't required to pass it to the SDK; you can pass `nil` instead. The server then creates a `ServiceContext` instance for you, using default values.

If you need to set properties for `ServiceContext`, you can do so by adding them to a new `NSDictionary` and then passing it as the `ServiceContext` argument:

```
NSDictionary *jsonObject = @{
    @"addGroupPermissions": @(YES),
    @"addGuestPermissions": @(YES)
};

LRJSONObjectWrapper *serviceContext = [[LRJSONObjectWrapper alloc] initWithJSONObject:jsonObject];
```

For more examples, see the test case `ServiceContextTest.m`.

Binaries

Some Liferay services require binary argument types like `NSData` or `LRUploadData`. The Mobile SDK converts `NSData` instances to `NSString` before sending the POST request. For example, `[@"hello" dataUsingEncoding:NSUTF8StringEncoding]` becomes a JSON array such as `"[104,101,108,108,111]"`. The Mobile SDK does this for you, so you don't have worry about it; you only need to pass the `NSData` instance to the method.

However, you need to be careful when using such methods. This is because you're allocating memory for the whole `NSData`, which may cause memory issues if the content is large.

Other Liferay service methods require `java.io.File` as an argument. In these cases the Mobile SDK requires `LRUploadData` instead. Here are two examples of creating `LRUploadData` instances:

```
LRUploadData *upload = [[LRUploadData alloc]
    initWithData:data fileName:@"file.png" mimeType:@"image/png"];

LRUploadData *upload = [[LRUploadData alloc]
    initWithInputStream:is length:length fileName:@"file.png" mimeType:@"image/png"];
```

The first constructor accepts an `NSData` argument, while the second accepts `NSInputStream`. As you can see, you also need to pass the file's mime type and name. The `length` is the size in bytes of the content being sent. The SDK sends a multipart form request to the Liferay instance. On the server side, a `File` instance is created and sent to the service method you're calling.

It's also possible to monitor service calls that upload data to Liferay. If want to listen for upload progress to create a progress bar, you can create a `LRProgressDelegate` delegate and set it to an `LRUploadData` object. Its `onProgressBytes` method is called for each byte chunk sent. It passes the bytes that were sent, the total number of bytes sent so far, and the total request size. For example:

```
@interface ProgressDelegate : NSObject <LRProgressDelegate>

@end

@implementation ProgressDelegate

- (void)onProgressBytes:(NSUInteger)bytes sent:(long long)sent
    total:(long long)total {

        // bytes contains the byte values that were sent.
        // sent will contain the total number of bytes sent.
        // total will contain the total size of the request in bytes.

}

@end
```

For more examples of this, see the test case `FileUploadTest.m`.

Related Topics

Building Mobile SDKs

Creating Android Apps that Use the Mobile SDK

79.9 Invoking Services Asynchronously from Your iOS App

The main drawback of using synchronous requests from your app is that each request must terminate before another can begin. If you're sending a large number of synchronous requests, performance suffers as a bottleneck forms while each one waits to be processed. Fortunately, Liferay's iOS SDK allows *asynchronous* HTTP requests. To do so, you need to set a callback to the session object. If you want to make synchronous requests again, you can set the callback to `nil`.

With the following steps, this tutorial shows you how to implement asynchronous requests in your iOS app:

1. Implement your callback class.
2. Instantiate your callback class and set it to the session.
3. Call Liferay services.

Objective-C is used in the code snippets that follow. Let the requesting begin!

Implementing Your Callback Class

To configure asynchronous requests, first create a class that conforms to the `LRCallback` protocol. When implementing this callback class, you need to implement its `onFailure` and `onSuccess` methods. These methods respectively determine what your app does when the request fails or succeeds. If a server side exception or a connection error occurs during the request, the `onFailure` method is called with an `NSError` instance that contains information about the error. Note that the `onSuccess` result parameter doesn't have a specific type. When deciding what to cast it to, you need to check the type in the service method signature.

The example code here implements a callback class for an app that retrieves blog entries from a Blogs portlet. The service method for this call is `getGroupEntriesWithGroupId`, which returns an `NSArray` instance. The `onSuccess` method's result parameter is therefore cast to this type:

```
#import "LRCallback.h"

@interface BlogsEntriesCallback : NSObject <LRCallback>

@end


#import "BlogsEntriesCallback.h"

@implementation BlogsEntriesCallback

- (void)onFailure:(NSError *)error {
    // Implement error handling code
}

- (void)onSuccess:(id)result {
    // Called after request has finished successfully
    NSArray *entries = (NSArray *)result;
}

@end
```

Awesome! Now you have a callback class that you can use with the session.

Set the Callback to the Session

Next, create an instance of this callback and set it to the session. If you haven't created a session yet, do so now. The tutorial Invoking Liferay Services in Your iOS App shows you how to create a session. Now you're ready to set the callback to the session. For example, this is done here for `BlogsEntriesCallback`:

```
BlogsEntriesCallback *callback = [[BlogsEntriesCallback alloc] init];

[session setCallback:callback];
```

Pretty simple! Now you're ready to make the service call.

Making the Service Call

Last but certainly not least, make the service call. This is done the same as calling any other service: create a service object from the session and use it to make the service call. This is also described in the tutorial Invoking Liferay Services in Your iOS App. Here, an example service call that gets all the blog entries from a site's Blogs portlet is shown:

```
[service getGroupEntriesWithGroupId:10184 status:0 start:-1 end:-1 error:&error];
```

Since the request is asynchronous, `getGroupEntriesWithGroupId` immediately returns `nil`. Once the request finishes successfully, the `onSuccess` method of your callback is invoked with the results on the main UI thread.

Great! Now you know how to make asynchronous requests in your iOS apps. However, there's another way to accomplish the same thing. This is discussed next.

Using Blocks as Callbacks

Instead of implementing a separate callback class, you can use an Objective-C block as a callback. An example of this is shown here for an asynchronous call that retrieves a user's sites. Note that this includes all the code required to make the call:

```
LRSession *session = [[LRSession alloc]
    initWithServer:@"http://localhost:8080" username:@"test@example.com" password:@"test"];

[session
    onSuccess:^(id result) {
        // Called after request has finished successfully
    }
    onFailure:^(NSError *e) {
        // Implement error handling code
    }
];

LRGroupService_v62 *service = [[LRGroupService_v62 alloc] initWithSession:session];

NSError *error;
[service getUserSites:&error];
```

When using a block as a callback, take care not to also set an `LRCallback` instance to the session. If you do, it gets overridden. Otherwise, support for blocks works the same way as described in the previous sections.

Super! Now you know two different ways to make asynchronous service requests in your iOS apps.

Related Topics

Invoking Liferay Services in Your iOS App

Creating Android Apps that Use the Mobile SDK

79.10 Sending Your iOS App's Requests Using Batch Processing

The Mobile SDK also allows sending requests in batch. This can be much more efficient than sending separate requests. For example, suppose you want to delete ten blog entries in a site's Blogs portlet at the same time. Instead of making a request for each deletion, you can create a batch of calls and send them all together.

This tutorial shows you how to implement batch processing for your iOS app. It's assumed that you already know how to invoke Liferay services from your iOS app. If you don't, see the tutorial Invoking Liferay Services in Your iOS App. Objective-C is used in the code snippets that follow. Now it's time to whip up a fresh batch of requests!

Implementing Batch Processing

Making service calls in batch only requires two extra steps over making them one at a time:

- Create a batch session with `LRBatchSession`.
- Make the batch service calls with the `invoke` method of `LRBatchSession`.

The rest of the steps are the same as making other service calls. You still need a service object, and you still need to call its service methods. As an example, here's a code snippet from an app that deletes a Blogs portlet's blog entries synchronously in batch:

```
#import "LRBatchSession.h"

LRBatchSession *batch = [[LRBatchSession alloc]
    initWithServer:@"http://localhost:8080" username:@"test@example.com" password:@"test"];
LRBlogsEntryService_v62 *service = [[LRBlogsEntryService_v62 alloc] initWithSession:batch];
NSError *error;

[service deleteEntryWithEntryId:1 error:&error];
[service deleteEntryWithEntryId:2 error:&error];
```

```
[service deleteEntryWithEntryId:3 error:&error];

NSArray *entries = [batch invoke:&error];
```

So what's going on here? After the import, `LRBatchSession` is used with a Liferay instance's URL and a user's credentials to create a batch session. You can alternatively pass a pre-existing `session` to the constructor. This is useful when you already have a session object and want to reuse the same credentials. Next, the service calls are made as usual (in this case, `deleteEntryWithEntryId`). With asynchronous calls, these methods return `nil` right away. Finally, call `[batch invoke:&error]`. This returns an `NSArray` containing the results for each service call (the return type for batch calls is always `NSArray`). Since there are three `deleteEntryWithEntryId` calls, the `entries` array contains three objects. The order of the results matches the order of the service calls.

If you want to make batch calls asynchronously, set the callback to the session as usual.

```
[batch setCallback:callback];
```

Great! Now you know how to utilize batch processing to speed up your app's requests.

Related Topics

Invoking Liferay Services in Your iOS App

Creating Android Apps that Use the Mobile SDK

79.11 Building Mobile SDKs

The Liferay Mobile SDK lets you connect your Android and iOS apps to a Liferay DXP instance. By accessing built-in portal services through Liferay's prebuilt Mobile SDK, your apps can access the out-of-the-box functionality in a Liferay DXP instance. But what if you want to call custom services that belong to a custom portlet? No problem! In this case, you need to build your own Mobile SDK that can call these custom portlet services.

Note that when you build a Mobile SDK for a portlet, it contains *only* the classes needed to call that portlet's remote services. You still need to install Liferay's prebuilt Mobile SDK in your app. It contains the framework required to construct remote service calls in general.

The Liferay Mobile SDK project contains a Mobile SDK Builder that generates a custom Mobile SDK for the Android and iOS platforms. The Mobile SDK Builder does this by generating client libraries that let your native mobile apps invoke a custom portlet's remote web services. Think of the Mobile SDK Builder as a Service Builder on the client side.

This tutorial covers how to build a custom Mobile SDK for Android and iOS. You'll begin by making sure the remote services are configured for any custom portlets you have.

Configuring Your Portlet's Remote Services

For the Mobile SDK Builder to discover a portlet's remote services, the services must be available and accompanied by a Web Service Deployment Descriptor (WSDD). For instructions on creating a portlet's remote services and building its WSDD, click here.

Next, you'll download the Liferay Mobile SDK's source code.

Downloading the Liferay Mobile SDK

To build a Mobile SDK for your custom portlet's services, you need to have the Liferay Mobile SDK's source code on your local machine. This code also contains the Mobile SDK Builder. You can get this code by cloning the Mobile SDK project via Git, or by downloading it from GitHub. To clone the Mobile SDK project with Git, open a terminal and navigate to the directory on your machine in which you want to put the Mobile SDK. Then run this command:

```
git clone git@github.com:liferay/liferay-mobile-sdk.git
```

Since the Mobile SDK changes frequently, you should check out the latest stable release for your chosen mobile platform (Android or iOS). Click here to see the list of available stable releases. Stable releases correspond to tags in GitHub that begin with the mobile platform and end with the Liferay Mobile SDK version. For example, the `android-7.0.6` tag corresponds to version 7.0.6 of the Liferay Mobile SDK for Android. To check out this tag in a new branch of the same name, you can use this command:

```
git checkout tags/android-7.0.6 -b android-7.0.6
```

Alternatively, you can download the ZIP or TAR.GZ file listed under each tag on GitHub.

Now you're ready to build the Mobile SDK!

Building a Liferay Mobile SDK

After you've downloaded the Mobile SDK's source code, you must build the module in which you'll build your custom portlet's Mobile SDK. The Mobile SDK Builder comes with a command line wizard that helps you build this module. To start the wizard, run the following command in the Mobile SDK source code's root folder:

```
./gradlew createModule
```

This starts the wizard with the most commonly required properties it needs to generate code for your portlet. If you need more control over these properties, run the same command with the `all` argument:

```
./gradlew createModule -P=all
```

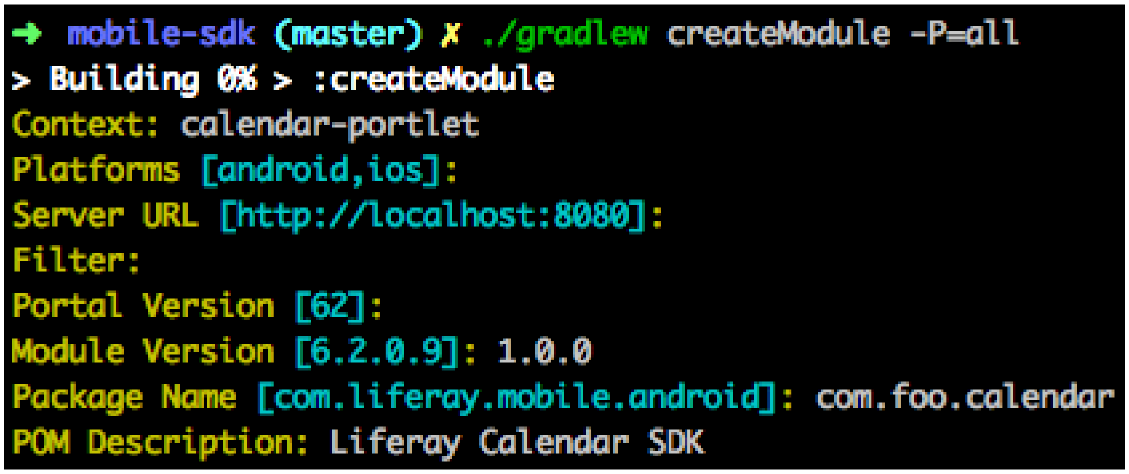
The wizard should look similar to this screenshot. Note that default values are in square brackets with blue text:

So what properties are available, and what do they do? Fantastic question! You can set the following properties during or after running `createModule`. If you want or need to set these properties after running `createModule`, you can do so in your module's `gradle.properties` file. The values in parentheses are the keys used in `gradle.properties`:

- `Context (context)`: Your portlet's web context. For example, if you're generating a Mobile SDK for Liferay DXP's Calendar portlet, which is generally deployed to the `calendar` context, then you should set the context value to `calendar`. If there are no services available at the specified context, you may have forgotten to generate your portlet's WSDD.

- `Platforms (platforms)`: The platforms to build the Mobile SDK for. By default, you can generate code for Android and iOS (`android,ios`).

- `Server URL (url)`: Your Liferay DXP instance's URL. To discover your services, the Mobile SDK Builder tries to connect to this instance at the specified context.



```
➜  mobile-sdk (master) ✗ ./gradlew createModule -P=all
> Building 0% > :createModule
Context: calendar-portlet
Platforms [android,ios]:
Server URL [http://localhost:8080]:
Filter:
Portal Version [62]:
Module Version [6.2.0.9]: 1.0.0
Package Name [com.liferay.mobile.android]: com.foo.calendar
POM Description: Liferay Calendar SDK
```

Figure 79.4: The Mobile SDK Builder's wizard lets you specify property values for building your module.

- `Filter (filter)`: Specifies the portlet entities the Mobile SDK can access. A blank value specifies all portlet entity services. For example, the Calendar portlet's entities include `CalendarBooking` and `CalendarResource`. To generate a Mobile SDK for only the `CalendarBooking` entity, set the filter's value to `calendarbooking` (all lowercase).

- `Module Version (version)`: The version number appended to your Mobile SDK's JAR (Android) and ZIP files (iOS). The sections on packaging your Mobile SDK explain this further.

- `Package Name (packageName)`: On Android, this is the package your Mobile SDK's classes are written to (iOS doesn't use packages). Note that the Liferay DXP version is appended to the end of the package name. For example, if you're using Liferay Portal 7.0 or Liferay DXP Digital Enterprise 7.0, and specify `com.liferay.mobile.android` as the package name, the Mobile SDK Builder appends v7 to the package name, yielding `com.liferay.mobile.android.v7`. This prevents collisions between classes with the same name, which lets you use Mobile SDKs for more than one portal version in the same app. You can use the `Portal Version` property to change the portal version.

- `POM Description (description)`: Your POM file's description.

Note that there's also a `destination` property that can only be set in the `gradle.properties` file. This property specifies the destination for the generated source files. You won't generally need to change this.

After you set the properties you need, the Mobile SDK Builder generates your module in the folder `modules/${your_portlet_context}`.

Now you can build your Mobile SDK. To do this, navigate to your module and run this command:

```
../../gradlew generate
```

By default, the builder writes the source files to `android/src/gen/java` and `ios/Source` in your module's folder.

If you update your portlet's remote services on the server side and need to update your Mobile SDK, simply run `../../gradlew generate` again.

Awesome! Now you know how to create and regenerate a Mobile SDK for your custom portlet's remote services. Next, you'll finish by packaging your Mobile SDK for the Android and iOS.

Packaging Your Mobile SDK for Android

To package your Mobile SDK in a JAR file for use in an Android project, run the following command from your module's folder:

```
../../gradlew jar
```

This packages your Mobile SDK in the following file:

- `modules/${your_portlet_context}/build/libs/liferay-${your_portlet_context}-android-sdk-${version}.jar`

To call your portlet's remote services, you must first install this file in your Android project. To do so, copy the file into your Android app's `app/libs` folder. Note that you must also install Liferay's prebuilt Mobile SDK in your app. Click here for instructions on doing this.

Also note that if you regenerate your Mobile SDK to include new functionality, you can update your module's version in its `gradle.properties` file. For example, if you added or changed a service method in the Mobile SDK you initially built, you could update it's version by setting `version=1.1` in your module's `gradle.properties` file.

To learn how to use the Mobile SDK in your Android app, see the rest of the Android Mobile SDK documentation. You can also use your Mobile SDK to create custom Screenlets in Liferay Screens.

Packaging Your Mobile SDK for iOS

To package your Mobile SDK in a ZIP file for use in an iOS project, run the following command from your module's folder:

```
../../gradlew zip
```

This packages your Mobile SDK in the following file:

- `modules/${your_portlet_context}/build/liferay-${your_portlet_context}-ios-sdk-${version}.zip`

To call your portlet's remote services, you must first install this file in your Xcode project. To do so, simply unzip it and add its files to your Xcode project.

To learn how to use the Mobile SDK in your iOS app, see the rest of the iOS Mobile SDK documentation. You can also use your Mobile SDK to create custom Screenlets in Liferay Screens.

Related Topics

Creating Android Apps that Use the Mobile SDK

Creating iOS Apps that Use the Mobile SDK

Android Apps with Liferay Screens

iOS Apps with Liferay Screens

Chapter 80

Service Builder

An application without reliable business logic or persistence isn't much of an application at all. Unfortunately, writing your own persistence code often takes a great deal of time. Fortunately, Liferay provides the Liferay Service Builder to generate it for you. You might now be thinking, "What?! I hate code generators!" Not to fear; you can still write your own persistence code if you wish. And if you choose to use Service Builder, you can edit and customize the code it generates. Regardless of how you produce your persistence code, you can then use it to implement your app's business logic.

This section of tutorials shows you how to use Service Builder to generate your persistence framework and implement your business logic. You're also shown how to use Spring in your app.

80.1 What is Service Builder?

Service Builder is a model-driven code generation tool built by Liferay that allows developers to define custom object models called entities. Service Builder generates a service layer through object-relational mapping (ORM) technology that provides a clean separation between your object model and code for the underlying database. This frees you to add the necessary business logic for your application. Service Builder takes an XML file as input and generates the necessary model, persistence, and service layers for your application. These layers provide a clean separation of concerns. Service Builder generates most of the common code needed to implement create, read, update, delete, and find operations on the database, allowing you to focus on the higher level aspects of service design. In this section, you'll learn some of the main benefits of using Service Builder:

- Integration with Liferay
- Automatically generated model, persistence, and service layers
- Automatically generated local and remote services
- Automatically generated Hibernate and Spring configurations
- Support for generating finder methods for entities and finder methods that account for permissions
- Built-in entity caching support
- Support for custom SQL queries and dynamic queries
- Saved development time

Liferay uses Service Builder to generate all of its internal database persistence code. In fact, all of Liferay's services, both local and remote, are generated by Service Builder. Additionally, the service modules in Liferay are generated by Service Builder. Service Builder's use in Liferay Portal demonstrates it to be a robust and reliable tool. Service Builder is easy to use and can save developers *lots* of development time. Although the number of files Service Builder generates can seem intimidating at first, developers only need to work with a few files in order to make customizations to their applications and add business logic.

Note: You don't have to use Service Builder for Liferay application development. It's entirely possible to develop Liferay plugins by writing custom code for database persistence using your persistence framework of choice. If you so choose, you can work directly with JPA or Hibernate.

One of the main ways Service Builder saves development time is by completely eliminating the need to write and maintain database access code. To generate a basic service layer, you only need to create a `service.xml` file and run Service Builder. This generates a new service `.jar` file for your project. The generated service `.jar` file includes a model layer, a persistence layer, a service layer, and related infrastructure. These distinct layers represent a healthy separation of concerns. The model layer is responsible for defining objects to represent your project's entities, the persistence layer is responsible for saving entities to and retrieving entities from the database, and the service layer is responsible for exposing CRUD and related methods for your entities as an API. The code Service Builder generates is database-agnostic, as is Liferay itself.

Each entity generated by Service Builder contains a model implementation class. Each entity also contains a local service implementation class, a remote service implementation class, or both, depending on how you configure Service Builder in your `service.xml` file. Customizations and business logic can be implemented in these three classes; in fact, these are the only classes generated by Service Builder that are intended to be customized. Ensuring that all customizations take place in only a few classes makes Service Builder projects easy to maintain. The local service implementation class is responsible for calling the persistence layer to retrieve and store data entities. Local services contain the business logic and access the persistence layer. They can be invoked by client code running in the same Java Virtual Machine. Remote services usually have additional code for permission checking and are meant to be accessible from anywhere over the Internet or your local network. Service Builder automatically generates the code necessary to allow access to the remote services. The remote services generated by Service Builder include SOAP utilities and can be accessed via SOAP or JSON.

Another way Service Builder saves development time is by providing Spring and Hibernate configurations for your project. Service Builder uses Spring dependency injection for making service implementation classes available at runtime and uses Spring AOP for database transaction management. Service Builder also uses the Hibernate persistence framework for object-relational mapping. As a convenience to developers, Service Builder hides the complexities of using these technologies. Developers can take advantage of Dependency Injection (DI), Aspect Oriented Programming (AOP), and Object-Relational Mapping (ORM) in their projects without having to manually set up a Spring or Hibernate environment or make any configurations.

Another benefit of using Service Builder is that it provides support for generating finder methods. Finder methods retrieve entity objects from the database based on specified parameters. You just need to specify the kinds of finder methods to be generated in the `service.xml` configuration file and Service Builder does the rest. The generated finder methods allow you, for example, to retrieve a list of all entities associated with a certain site or a list of all entities associated with a certain site *and* a certain user. Service Builder not only provides support for generating these kinds of simple finder methods but also for finder methods that take Liferay's permissions into account. For example, if you are using Liferay's permissions system to protect access to your entities, Service Builder can generate a different kind of finder method that only returns entities that the logged-in user has permission to view.

Service Builder also provides built-in caching support. Liferay caches objects at three levels: *entity*, *finder*, and *Hibernate*. By default, Liferay uses Ehcache as an underlying cache provider for each of these cache levels. However, this is configurable via portal properties. All you have to do to enable entity and finder caching for an entity in your project is to set the `cache-enabled=true` attribute of your entity's `<entity>` element in your `service.xml` configuration file. Please refer to the Distributed Caching documentation for more details about Liferay caching.

Service Builder is a flexible tool. It automates many of the common tasks associated with creating database persistence code but it doesn't prevent you from creating custom SQL queries or custom finder methods. Service Builder allows developers to define custom SQL queries in an XML file and to implement custom finder methods to run the queries. This could be useful, for example, for retrieving specific pieces of information from multiple tables via an SQL join. Service Builder also supports retrieving database information via dynamic query. Liferay's dynamic query API leverages Hibernate's criteria API.

In summary, we encourage developers to use Service Builder for portlet and plugin development because it's a proven solution used by many Liferay plugins and by Liferay Portal itself. It generates distinct model, persistence, and service layers, local and remote services, Spring and Hibernate configurations, and related infrastructure without requiring any manual intervention by developers. It also allows basic SQL queries and finder methods to be generated and ones that filter results, taking Liferay's permissions into account. Service Builder also provides support for entity and query caching. Each of these features can save lots of development time, both initial development time and time that would have to be spent maintaining, extending, or customizing a project. Finally, Service Builder is not a restrictive tool: it allows custom SQL queries and finder methods to be added and it also supports dynamic query.

Service Builder supports Liferay 7's modular application development style. When you configure Service Builder in a Liferay 7 OSGi module, this module functions as a *service* module and its a convention to append `service` to the module's name. Liferay includes a Service Builder plugin and a build tool-specific plugins such as the Gradle Service Builder plugin. The Gradle Service Builder plugins includes Liferay's Service Builder plugin as a dependency. The `basic-api`, `basic-service`, `basic-web` modules in the Service Builder Blade Sample project serve as example Service Builder projects.

Chapter 81

Service Builder Persistence

Liferay's Service Builder can generate your project's persistence layer by automating the creation of interfaces and classes. Your application's persistence layer persists data represented by your configured entities to a database. In fact, your local service implementation classes are responsible for calling the persistence layer to retrieve and store your application's data. So instead of taking the time-consuming route of writing your own persistence layer, you can use Liferay's Service Builder to quickly define your entities and generate the layer instantaneously.

In this section of tutorials, you'll learn how to define an object-relational map and generate your persistence layer from that map. You'll also learn about the local and remote services Service Builder generates, and how you can use them for your own application. You'll also discuss how to use the `ServiceContext` class, model hints, SQL queries, and Hibernate's criteria API.

81.1 Defining an Object-Relational Map with Service Builder

In this tutorial, you'll learn how to define an object relational map for your application so that it can persist data. As an example, you'll examine the existing Liferay Bookmarks application that uses Service Builder.

The Bookmarks application is an example portlet project that an organization can use to bookmark assets in Liferay. The application defines two entities, or model types, to represent an organization's bookmarks and their folders. These entities are called *bookmark entries* and *bookmark folders*. Since a bookmark must have a folder (even if it's a root folder), the entry entity references a folder entity as one of its attributes.

You can design your application's modules anyway you like, but for the Bookmarks application, its Java sources reside in the `bookmarks-api`, `bookmarks-service`, and `bookmarks-web` modules. Notice the `BookmarksAdminPortlet.java` and `BookmarksPortlet.java` files in the `com.liferay.bookmarks.web.portlet` package in the `bookmarks-web` module. These portlet classes extend Liferay's `MVCPortlet` class. They act as the controllers in the MVC pattern. These classes contain the business logic that invokes the Service Builder generated bookmarks services that you'll learn how to create in this section. The application's view layer is implemented in the JSPs in the `bookmarks-web/src/main/resources/META-INF/resources` folder.

You can learn how to generate a generic modular application from scratch that includes the `*api`, `*service`, and `*web` modules by default in the Modularizing an Existing Portlet tutorial. This tutorial assumes you've assembled your application's modules similarly to the linked tutorial above. Be sure to also visit the Fundamentals tutorial for additional info on the `*api`, `*service`, and `*web` modules.

The first step in using Service Builder is to define your model classes and their attributes in a `service.xml` file. This file's location typically resides in the root folder of the `*-service` module, although you can configure your build tool to recognize it from other directories. In Service Builder terminology, your model classes are called entities. For example, the Bookmarks application has two entities: `BookmarksEntry` and `BookmarksFolder`. The requirements for each of these entities are defined in the `bookmarks-service` module's service.xml listed in the `<column />` elements.

Once Service Builder reads the `service.xml` file, you can define your entities. Liferay @ide@ makes it very easy to define entities in your application's `service.xml` file. To define a custom entity, follow these steps:

1. Create the `service.xml` file in your project's `*-service` module. It resides in the root folder of that module, if one does not already exist there.

2. Define global information for the service.

3. Define service entities.

4. Define the columns (attributes) for each service entity.

5. Define relationships between entities.

6. Define a default order for the entity instances to be retrieved from the database.

7. Define finder methods that retrieve objects from the database based on specified parameters.

You'll examine these steps in detail next, starting with creating a `service.xml` file.

Step 1: Creating the service.xml File

To define a service for your portlet project, you must create a `service.xml` file. The DTD (Document Type Declaration) file http://www.liferay.com/dtd/liferay-service-builder_7_0_0.dtd specifies the format and requirements of the XML to use. You can create your `service.xml` file manually, following the DTD, or you can use Liferay @ide@. @ide@ helps you build the `service.xml` file piece-by-piece, taking the guesswork out of creating XML that adheres to the DTD.

If a default `service.xml` file already exists in your `*-service` module folder, check to see if it has an `<entity />` element named *Foo*. If it has the Foo entity, remove the entire `<entity name="Foo" ...> ... </entity>` element. The Liferay @ide@ project wizard creates the Foo entity as an example. It has no practical use for you.

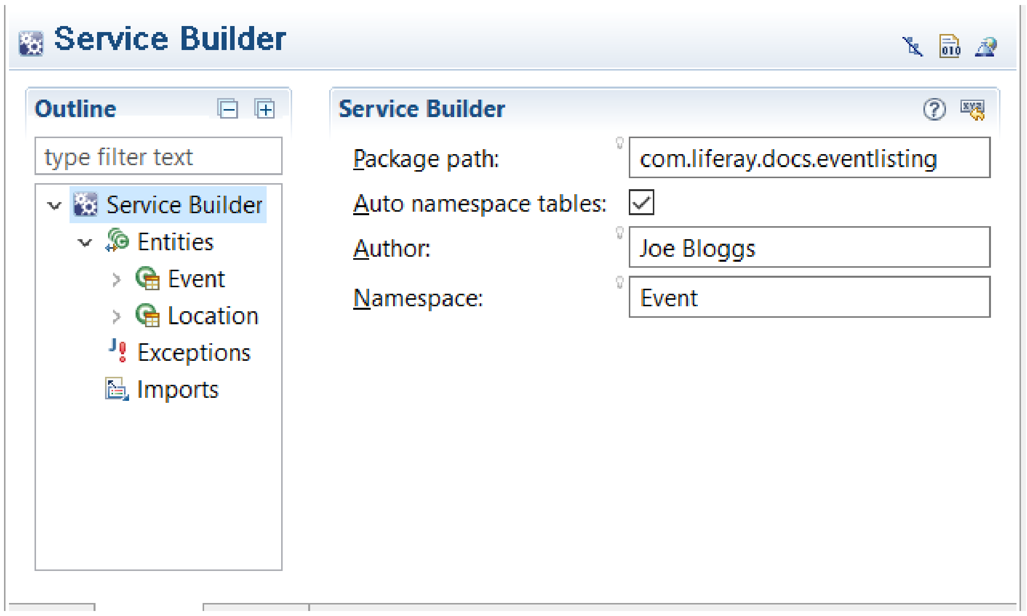
If you don't already have a `service.xml` file, create one in your `*-service` module. Once it's created, open it. Liferay @ide@ provides a Diagram mode and a Source mode to give you different perspectives of the service information in your `service.xml` file. Diagram mode is helpful for creating and visualizing relationships between service entities. Source mode brings up the `service.xml` file's raw XML content in the editor. You can switch between these modes as you wish.

Next, you can start filling out the global information for your service.

Step 2: Defining Global Service Information

A service's global information applies to all of its entities, so it's a good place to start. In Liferay @ide@, select the *Service Builder* node in the upper left corner of the Overview mode of your `service.xml` file. The main section of the view now shows the Service Builder form in which you can enter your service's global information. The fields include the service's package path, author, and namespace options.

The package path specifies the package in which the service and persistence classes are generated. The package path defined above ensures that the service classes are generated in the `com.liferay.docs.eventlisting` package in the `*-api` module. The persistence classes are generated



Package path: com.liferay.docs.eventlisting
Auto namespace tables: ✓
Author: Joe Bloggs
Namespace: Event

Figure 81.1: This is the Service Builder form from a fictitious Event Listing application's `service.xml`.

in a package of the same name in the `*-service` module. For examples, you can look in the Bookmarks application's bookmarks-api and bookmarks-service modules to see an example of how these are automatically generated for you. Refer to the Running Service Builder and Understanding the Generated Code tutorial for a description of the contents of these packages.

Service Builder uses the service namespace in naming the database tables it generates for the service. For example, *Event* could serve as the namespace for an Event Listing portlet service.

```
<namespace>Event</namespace>
```

Service Builder uses the namespace in the following SQL scripts it generates in your `src/main/resources/META-INF/sql` folder:

- `indexes.sql`
- `sequences.sql`
- `tables.sql`

Note: The folder location for holding your generated SQL scripts is configurable. For example, if you're using Ant can configure an argument in your `build.xml` similar to this:

```
 <arg value="service.sql.dir=${basedir}/../sql"/>
```

If you're using Gradle, you can define the `sqlDir` setting in the project's `build.gradle` file or Maven `pom.xml` file, the same way the `databaseNameMaxLength` setting is applied in the examples below.

Liferay DXP uses these scripts to create database tables for all the entities defined in the `service.xml` file. Service Builder prepends the namespace to the database table names. Since the namespace value above is `Event`, the names of the database tables created for the entities start with `Event_` as their prefix. The namespace for each Service Builder project must be unique. Separate plugins should use separate namespaces and should not use a namespace already used by Liferay (such as `Users` or `Groups`). Check the table names in Liferay's database if you're wondering which namespaces are already in use.

Warning: Use caution when assigning namespace values. Some databases have strong restrictions on database table and column name lengths. The Service Builder Gradle and Maven plugin parameter `databaseNameMaxLength` sets the maximum length you can use for your table and column names. Here are paraphrased examples of setting `databaseNameMaxLength` in build files:

Gradle `build.gradle`

```
buildService {
    ...
    databaseNameMaxLength = 64
    ...
}
```

Maven `pom.xml`

```
<configuration>
    ...
    <databaseNameMaxLength>64</databaseNameMaxLength>
    ...
</configuration>
```

As the last piece of global information, enter your name as the service's *author* in your `service.xml` file. Service Builder adds `@author` annotations with the specified name to all of the generated Java classes and interfaces. Save your `service.xml` file to preserve the information you added. Next, you'll add entities for your service's events and locations.

Step 3: Defining Service Entities

Entities are the heart and soul of a service. Entities represent the map between the model objects in Java and the fields and tables in your database. Once your entities are defined, Service Builder handles the mapping automatically, giving you a facility for taking Java objects and persisting them. For the Bookmarks application, two entities are created according to its service.xml –one for bookmark entries and one for bookmark folders.

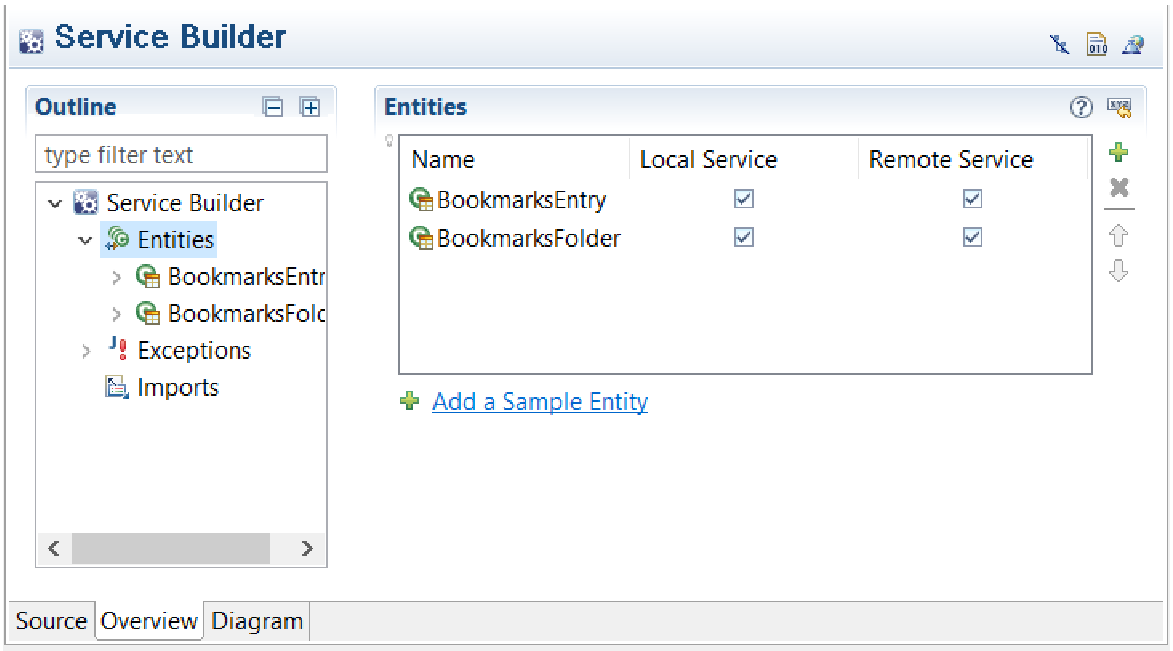
Here's a summary of the information used for the BookmarksEntry entity:

- **Name:** *BookmarksEntry*
- **Local service:** *yes*
- **Remote service:** *yes*

And here's what was used for the BookmarksFolder entity:

- **Name:** *BookmarksFolder*
- **Local service:** *yes*
- **Remote service:** *yes*

To create your entities using Liferay @ide@, select the *Entities* node under the Service Builder node in the outline on the left side of the `service.xml` editor in Overview mode. In the main part of the view, notice that the Entities table is empty. Create an entity by clicking on the *Add Entity* icon (✚) to the right of the table. Enter your entity's name and if you'd like to generate local and remote services for that entity. Add as many entities as you need.



Name	Local Service	Remote Service
BookmarksEntry	✓	✓
BookmarksFolder	✓	✓

Figure 81.2: Adding service entities is easy with Liferay @ide@'s *Overview* mode of your `service.xml` file.

An entity's name is used to name the database table for persisting instances of the entity. The actual name of the database table is prefixed with the namespace; the Bookmarks example creates one database table named `Bookmarks_BookmarksEntry` and another named `Bookmarks_BookmarksFolder`.

Setting the *local service* attribute to `true` instructs Service Builder to generate local interfaces for the entity's services. The default value for local service is `false`. Local services can only be invoked from the Liferay server on which they're deployed. Therefore, if your application will be deployed to Liferay, the service will be local from your Liferay server's point of view.

Setting the *remote service* attribute to `true` instructs Service Builder to generate remote interfaces for the service. The default value for remote service is `true`. You could build a fully-functional application without generating remote services. In that case, you could set local service to `true` and remote service to `false` for your entities. If, however, you want to enable remote access to your application's services, you should set both local service and remote service to `true`.

Tip: Suppose you have an existing DAO service for an entity built using some other framework such as JPA. You can set local service to `false` and remote service to `true` so that the methods of your remote `-Impl` class can call the methods of your existing DAO. This enables your entity to integrate with Liferay's permission-checking system and provides access to the web service APIs generated by Service Builder. This is a very handy, quite powerful, and often used feature of Liferay.

Now that you've seen how to create your application's entities, you'll learn how to describe their attributes using entity *columns*.

Step 4: Defining the Columns (Attributes) for Each Service Entity

Each entity is described by its columns, which represent an entity's attributes. These attributes map on the one side to fields in a table and on the other side to attributes of a Java object. To add attributes for your entity, you need to drill down to its columns in the Overview mode outline of the `service.xml` file. From the outline, expand the *Entities* node and expand an entity node. Then select the *Columns* node. Liferay @ide@ displays a table of the entity's columns.

Service Builder creates a database field for each column you add to the `service.xml` file. It maps a database field type appropriate to the Java type specified for each column, and it does this across all the databases Liferay supports. Once Service Builder runs, it generates a Hibernate configuration that handles the object-relational mapping. Service Builder automatically generates getter/setter methods in the model class for these attributes. The column's Name specifies the name used in the getters and setters that are created for the entity's Java field. The column's Type indicates the Java type of this field for the entity. If a column's Primary (i.e., primary key) attribute value is set to `true`, then the column becomes part of the primary key for the entity. An entity's primary key is a unique identifier for the entity. If only one column has Primary set to `true`, then that column represents the entire primary key for the entity. This is the case in the Event Listing example. However, it's possible to use multiple columns as the primary key for an entity. In this case, the combination of columns makes up a compound primary key for the entity.

Note: Each Service Builder generated `*LocalServiceImpl` class has a `CounterLocalService` field for generating unique entity instance primary keys. For details, see Creating Local Services.

Note: On deploying a `*service` module, Service Builder automatically generates indexes for all entity primary keys.

Similar to the way you used the form table for adding entities, add attribute columns for each of your entities. Create each attribute by clicking on the Add icon (➕). Then fill in the name of the attribute, select its type, and specify whether it is a primary key for the entity. While your cursor is in a column's *Type* field, an option icon appears. Click this icon to select the appropriate type for the column. Create a column for each attribute of your entity or entities.

In addition to columns for your entity's primary key and attributes, it's recommended to add columns for portal instance ID and site ID. They allow your portlet to support the multi-tenancy features of Liferay, so that each portal instance and each site in a portal instance can have independent sets of portlet data. To hold the site's ID, add a column called `groupId` of type `long`. To hold the portal instance's ID, add a column called `companyId` of type `long`. If you'd like to add these columns to your entities, follow the table below.

Portal and site scope columns

Name	Type	Primary
`companyId`	long	no
`groupId`	long	no

You'll also want to know who owns each entity instance. To keep track of that, add a column called `userId` of type `long`.

User column

Name	Type	Primary
`userId`	long	no

Lastly, you can add columns to help audit your entities. For example, you could create a column named `createDate` of type `Date` to note the date an entity instance was created. And add a column named `modifiedDate` of type `Date` to track the last time an entity instance was modified.

Audit columns

Name	Type	Primary
`userId`	long	no
`createDate`	Date	no
`modifiedDate`	Date	no

Great! Your entities are set with the columns that not only represent their attributes, but also support multi-tenancy and entity auditing. Next, you'll learn how to specify the relationship service entities.

Step 5: Defining Relationships Between Service Entities

Often you'll want to reference one type of entity in the context of another entity. That is, you'll want to *relate* the entities. Liferay's Bookmarks application defines a relationship between an entry and its folder.

As mentioned earlier, each bookmark must have a folder. Therefore, each `BookmarksEntry` entity must relate to a `BookmarksFolder` entity. Liferay @ide@'s Diagram mode for `service.xml` makes relating entities easy. First, select Diagram mode for the `service.xml` file. Then select the *Relationship* option under *Connections* in the palette on the right side of the view. This relationship tool helps you draw relationships between entities in the diagram. Click your first entity and move your cursor over to the entity you'd like to relate it with. Liferay @ide@ draws a dashed line from your selected entity to the cursor. Click the second entity to complete drawing the relationship. Liferay IDE turns the dashed line into a solid line, with an arrow pointing to the second entity. Save the `service.xml` file.

Congratulations! You've related two entities. Their relationship should show in Diagram mode and look similar to that of the figure below.

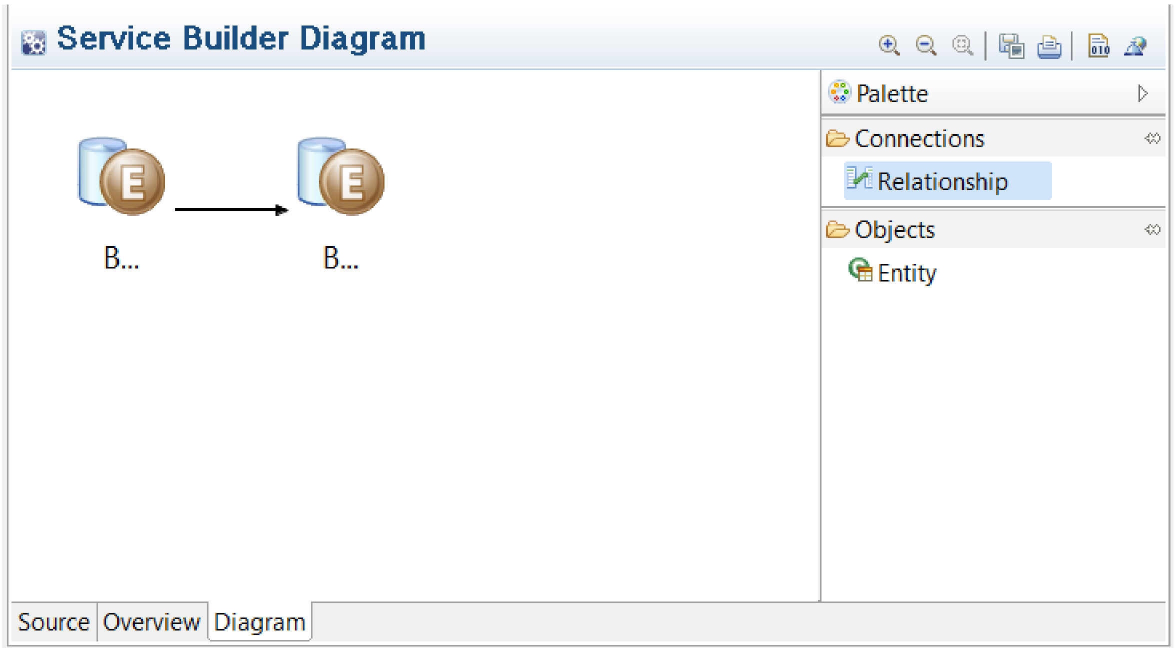


Figure 81.3: Relating entities is a snap in Liferay @ide@'s *Diagram* mode for `service.xml`.

Switch to *Source* mode in the editor for your `service.xml` file and note that Liferay @ide@ created a column element in the first selected entity to hold the ID of the corresponding entity instance reference. For example:

```
<column name="folderId" type="long" />
```

Now that your entity columns are in place, you can specify the default order in which the entity instances are retrieved from the database.

Step 6: Defining Ordering of Service Entity Instances

Often, you want to retrieve multiple instances of a given entity and list them in a particular order. Liferay lets you specify the default order of the entities in your `service.xml` file.

Suppose you want to return `BookmarksEntry` entities alphabetically by name. It's easy to specify these default orderings using Liferay @ide@. Switch back to *Overview* mode in the editor for your `service.xml` file. Then select the *Order* node under the entity node in the outline on the left side of the view. The IDE displays a form for ordering the chosen entity. Check the *Specify ordering* checkbox to show the form for specifying the ordering. Create an order column by clicking the *Add* icon ( ) to the right of the table. Enter the column name (e.g., *name*, *date*, etc.) to use in ordering the entity. Click the *Browse icon* ( ) to the right of the *By* field and choose the *asc* or *desc* option. This orders the entity in ascending or descending order.

Now that you know how to order your service entities, the last thing to do is to define the finder methods for retrieving entity instances from the database.

Step 7: Defining Service Entity Finder Methods

Finder methods retrieve entity objects from the database based on specified parameters. You'll probably want to create at least one finder method for each entity you create in your services. Service Builder generates several methods based on each finder you create for an entity. It creates methods to fetch, find, remove, and count entity instances based on the finder's parameters.

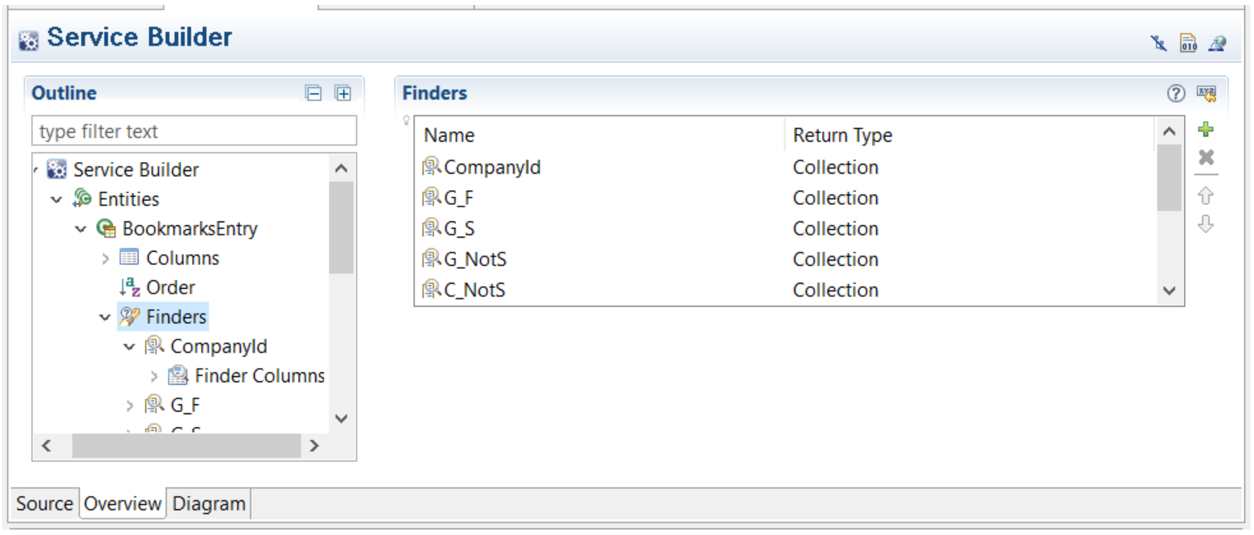
For many applications, it's important to be able to find its entities per site. You can specify these finders using Liferay @ide@'s Overview mode of `service.xml`. Select the *Finders* node under the entity node in the Outline on the left side of the screen. The IDE displays an empty *Finders* table in the main part of the view. Create a new finder by clicking the *Add* icon ( ) to the right of the table. Give your finder a name and return type. Use the Java camel-case naming convention when naming finders since the finder's name is used to name the methods that Service Builder creates. The IDE creates a new finder sub-node under the *Finders* node in the outline. Next, you'll learn how to specify the finder column for this node.

Important: DO NOT create finders that use entity primary key as parameters. They're unnecessary as Service Builder automatically generates `findByPrimaryKey` and `fetchByPrimaryKey` methods for all entity primary keys. On deploying a `*service` module, Service Builder creates indexes for all entity primary key columns and finder columns. Adding finders that use entity primary keys results in attempts to create multiple indexes for the same columns—Oracle DB, for example, reports these attempts as errors.

Under the new finder node, the IDE created a *Finder Columns* node. Select the *Finder Columns* node to specify the columns for your finder's parameters. Create a new finder column by clicking the *Add* icon and specifying the column's name. Keep in mind that you can specify multiple parameters (columns) for a finder.

If you're creating site-scoped entities (entities whose data should be unique to each site), you should follow the steps described above to create finders by `groupId` for retrieving your entities. Remember to save your `service.xml` file after editing it to preserve the finders you define.

When you run Service Builder, it generates finder-related methods (e.g., `fetchByGroupId`, `findByGroupId`, `removeByGroupId`, `countByGroupId`) for the your entities in the `*Persistence` and `*PersistenceImpl` classes.



Name	Return Type
CompanyId	Collection
G_F	Collection
G_S	Collection
G_NotS	Collection
C_NotS	Collection

Figure 81.4: Creating Finder entities is easy with Liferay @ide@.

The first of these classes is the interface; the second is its implementation. For example, Liferay's Bookmarks application generates its entity finder methods in the `-Persistence` classes found in the `/bookmarks-api/src/main/java/com/liferay/bookmarks/service/persistence` folder and the `-PersistenceImpl` classes in the `/bookmarks-service/src/main/java/com/liferay/bookmarks/service/persistence/impl` folder.

Now you know to configure Service Builder to create finder methods for your entity. Terrific!

Now that you've specified the service for your project, you're ready to *build* the service by running Service Builder. To learn how to run Service Builder and to learn about the code that Service Builder generates, please refer to the Running Service Builder and Understanding the Generated Code tutorial.

Related Topics

What is Service Builder?

Running Service Builder and Understanding the Generated Code

81.2 Running Service Builder and Understanding the Generated Code

This tutorial explains how to run Service Builder and provides an overview of the code that Service Builder generates. If you'd like to use Service Builder in your application but haven't yet created a `service.xml` file, visit the Defining an Object-Relational Map with Service Builder tutorial and then come back to this one.

Running Service Builder

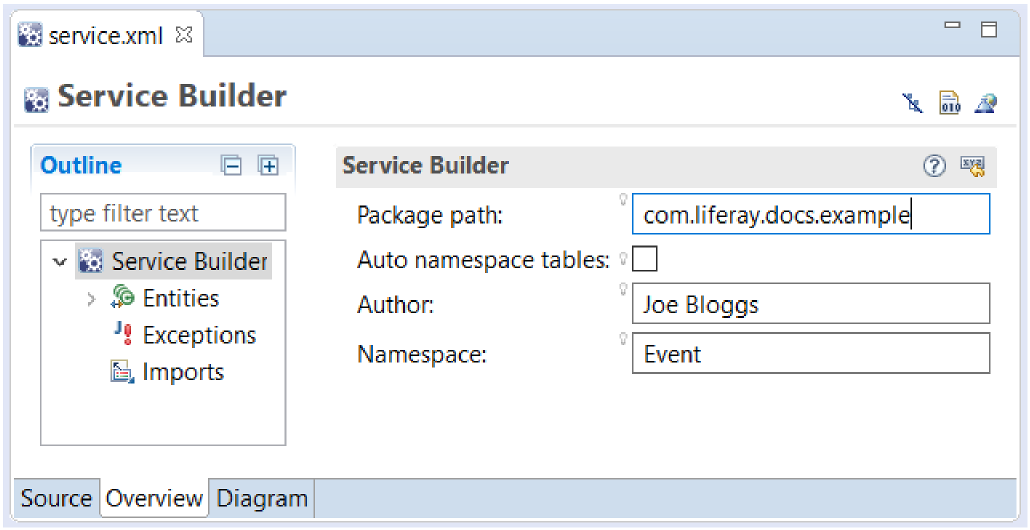
To build a service from a `service.xml` file, you can use Liferay @ide@ or a terminal window. In this tutorial, you'll refer to the Event Listing example project that's referenced throughout the Liferay Service Builder tutorials.

Now let's learn how to run Service Builder.

Using Liferay @ide@

From the Package Explorer, open the `service.xml` file from your `*-service` module's root folder. By default, the file opens up in the Service Builder Editor. Make sure you are in Overview mode. Then click the *Build Services* button ( ) near the top-right corner of the view.

Make sure to click the *Build Services* button and not the *Build WSDD* button ( ) that appears next to it. Building the WSDDs won't hurt anything, but you'll generate files for the remote service instead of the local one. For information about WSDDs (web service deployment descriptors), please refer to the SOAP Web Services tutorial.



Service Builder

Package path: com.liferay.docs.example

Auto namespace tables: ☐

Author: Joe Bloggs

Namespace: Event

Figure 81.5: The *Overview* mode in the editor provides a nested outline which you can expand, a form for editing basic Service Builder attributes, and buttons for building services or building web service deployment descriptors.

Another simple way to run Service Builder is to right-click on your project's name in the Package Explorer and then select *Liferay* → *build-service*.

After running Service Builder, your generated files are available. More information about the generated files appears below.

Using the Terminal

Open a terminal window and navigate to your module project's root folder, which should be located in your Liferay Workspace's `modules` directory. To learn more about creating your module project in a Liferay Workspace, visit the Creating Modules with Blade CLI tutorial. You can leverage the Service Builder Template to create your own predefined Service Builder project.

Liferay Workspace offers a Gradle or Maven build environment; this tutorial shows how to use both. Liferay is tool agnostic, however, and you can use other tools, as well.

For Gradle projects, enter the following command in your module project's root folder to build your services:

```
gradlew buildService
```

Note: Liferay Workspaces provide the Gradle Wrapper script for usage, if you don't have Gradle installed globally in your classpath. It is located in the workspace's root folder, so you can call it from your module project's root folder, if necessary (e.g., `../../gradlew buildService`).

If your module project uses Maven, you can build services running the following command from the module project's root folder:

```
mvn service-builder:build
```

Important: The `mvn service-builder:build` command only works if you're using the `com.liferay.portal.tools.service.bu`
plugin version 1.0.145+. Maven projects using an earlier version of the Service Builder plugin should update their POM accordingly. See the Using Service Builder in a Maven Project tutorial for more information on using Maven to run Service Builder.

When the service has been successfully generated, a `BUILD SUCCESSFUL` message appears in your terminal window. You should also see that a large number of files have been generated in your project. These files include a model layer, service layer, and persistence layer. Don't worry about the number of generated files–you'll never have to customize more than three of them. To review the code that Service Builder generates for your entities, see the next section.

Understanding the Code Generated by Service Builder

Now you'll examine the files Service Builder generated for your entity. Note that the files listed under Local Service and Remote Service below are only generated for an entity that has both `local-service` and `remote-service` attributes set to `true`. Service Builder generates services for these entities in two locations in your project. These locations use the package path that you specified in your `service.xml` file. For Liferay's Bookmarks application, for example, these two locations are the following ones:

- `/bookmarks-api/src/main/java/com/liferay/bookmarks`
- `/bookmarks-service/src/main/java/com/liferay/bookmarks`

The `bookmarks-api` module contains the API for the Bookmarks project. All the classes and interfaces in the `*-api` module are packaged in a `.jar` file called `PROJECT_NAME-api.jar` in the module's `build/libs` folder. This `.jar` file is generated whenever you compile and deploy your module. When deploying this JAR to Liferay, the necessary interfaces to *define* the service API are available.

The `bookmarks-service` module contains the implementation of the interfaces defined in the `bookmarks-api` module. These interfaces provide OSGi services for the portal instance to which your application is deployed. Service Builder generates classes and interfaces belonging to the persistence layer, service layer, and model layer in the `/bookmarks-api/src/main/java/com/liferay/bookmarks` and `/bookmarks-service/src/main/java/com/liferay/bookmarks` packages.

Now you'll look at the classes and interfaces generated for the entities you specified. Each entity has similar classes generated for it, depending on what you specfied for them in the `service.xml`. You won't have to customize more than three classes for each entity. These customizable classes are `*LocalServiceImpl`, `*ServiceImpl`, and `*Impl`.

- Persistence

    - `[ENTITY_NAME]Persistence`: Persistence interface that defines CRUD methods for the entity such as `create`, `remove`, `countAll`, `find`, `findAll`, etc.
    - `[ENTITY_NAME]PersistenceImpl`: Persistence implementation class that implements `[ENTITY_NAME]Persistence`.

  - `[ENTITY_NAME]Util`: Persistence utility class that wraps `[ENTITY_NAME]PersistenceImpl` and provides direct access to the database for CRUD operations. This utility should only be used by the service layer; in your portlet classes, use `[ENTITY_NAME]LocalServiceUtil` or `[ENTITY_NAME]ServiceUtil` instead.

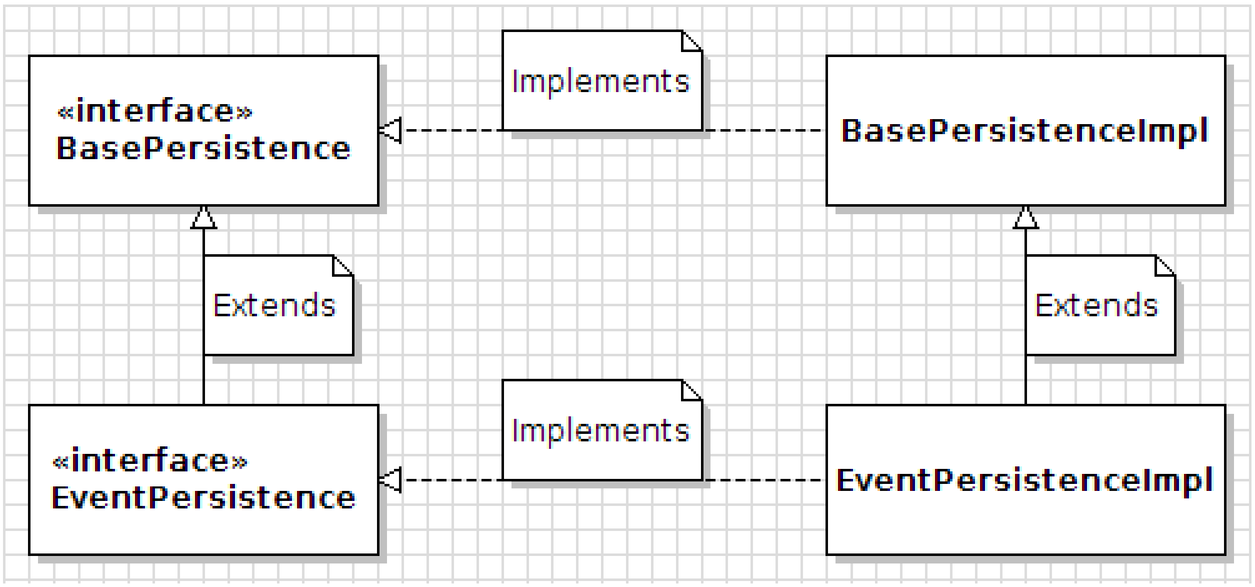


Figure 81.6: Service Builder generates these persistence classes and interfaces. You shouldn't (and you won't need to) customize any of these classes or interfaces.

- Local Service (generated for an entity only if an entity's `local-service` attribute is set to `true` in `service.xml`)

  - `[ENTITY_NAME]LocalService`: Local service interface.
  - `[ENTITY_NAME]LocalServiceImpl` (**LOCAL SERVICE IMPLEMENTATION**): Local service implementation. This is the only class in the local service that you should modify manually. You can add custom business logic here. For any custom methods added here, Service Builder adds corresponding methods to the `[ENTITY_NAME]LocalService` interface the next time you run it.
  - `[ENTITY_NAME]LocalServiceBaseImpl`: Local service base implementation. This is an abstract class. Service Builder injects a number of instances of various service and persistence classes into this class. `@abstract`
  - `[ENTITY_NAME]LocalServiceUtil`: Local service utility class which wraps `[ENTITY_NAME]LocalServiceImpl` and serves as the primary local access point to the service layer.
  - `[ENTITY_NAME]LocalServiceWrapper`: Local service wrapper which implements `[ENTITY_NAME]LocalService`. This class is designed to be extended and it allows developers to customize the entity's local services.

- Remote Service (generated for an entity only if an entity's `remote-service` attribute is *not* set to `false` in `service.xml`)

  - `[ENTITY_NAME]Service`: Remote service interface.
  - `[ENTITY_NAME]ServiceImpl` (**REMOTE SERVICE IMPLEMENTATION**): Remote service implementation. This is the only class in the remote service that you should modify manually.

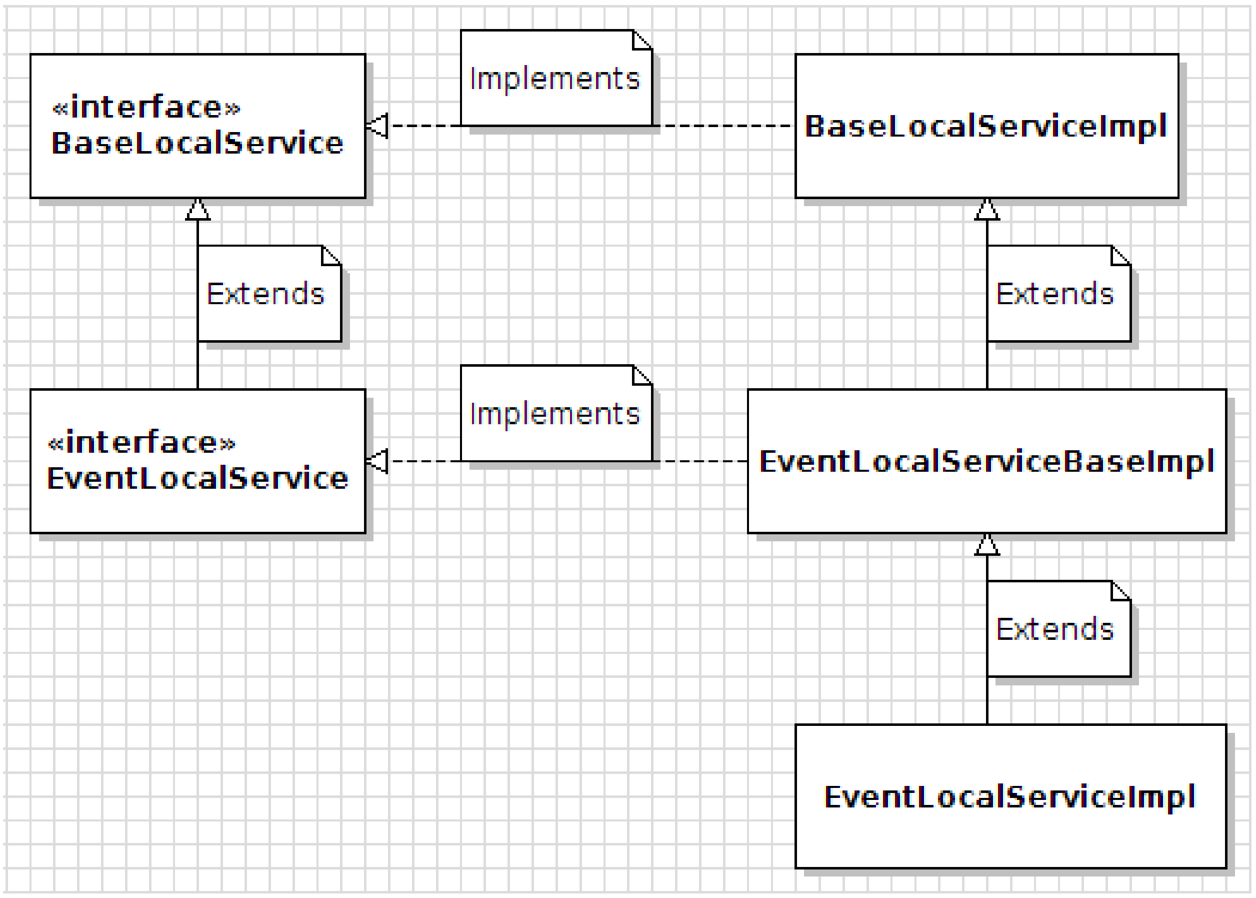


Figure 81.7: Service Builder generates these service classes and interfaces. Only EventLocalServiceImpl allows custom methods to be added to the service layer.

Here, you can write code that adds additional security checks and invokes the local services. For any custom methods added here, Service Builder adds corresponding methods to the `[ENTITY_NAME]Service` interface the next time you run it.

- `[ENTITY_NAME]ServiceBaseImpl`: Remote service base implementation. This is an abstract class. `@abstract`
- `[ENTITY_NAME]ServiceUtil`: Remote service utility class which wraps `[ENTITY_NAME]ServiceImpl` and serves as the primary remote access point to the service layer.
- `[ENTITY_NAME]ServiceWrapper`: Remote service wrapper which implements `[ENTITY_NAME]Service`. This class is designed to be extended and it allows developers to customize the remote entity's services.
- `[ENTITY_NAME]ServiceSoap`: SOAP utility which the remote `[ENTITY_NAME]ServiceUtil` remote service utility can access.
- `[ENTITY_NAME]Soap`: SOAP model, similar to `[ENTITY_NAME]ModelImpl`. `[ENTITY_NAME]Soap` is serializable; it does not implement `[ENTITY_NAME]`.

• Model

- `[ENTITY_NAME]Model`: Base model interface. This interface and its `[ENTITY_NAME]ModelImpl` implementation serve only as a container for the default property accessors generated by Service Builder. Any helper methods and all application logic should be added to `[ENTITY_NAME]Impl`.

- `[ENTITY_NAME]ModelImpl`: Base model implementation.
- `[ENTITY_NAME]`: [ENTITY_NAME] model interface which extends `[ENTITY_NAME]Model`.
- `[ENTITY_NAME]Impl`: (**MODEL IMPLEMENTATION**) Model implementation. You can use this class to add helper methods and application logic to your model. If you don't add any helper methods or application logic, only the auto-generated field getters and setters are available. Whenever you add custom methods to this class, Service Builder adds corresponding methods to the `[ENTITY_NAME]` interface the next time you run it.
- `[ENTITY_NAME]Wrapper`: Wrapper, wraps `[ENTITY_NAME]`.

Note: `*Util` classes are generated for backwards compatibility purposes only. Your module applications should avoid calling the util classes.

Each file that Service Builder generates is assembled from an associated FreeMarker template. You can find Service Builder's FreeMarker templates in the portal-tools-service-builder module. For example, if you want to find out how a `*ServiceImpl.java` file is generated, just look at the service_impl.ftl template.

Of all the classes generated by Service Builder, only three should be manually modified: `*LocalServiceImpl`, `*ServiceImpl` and `*Impl`. If you manually modify the other classes, your changes are overwritten the next time you run Service Builder. Whenever you add methods to, remove methods from, or change a method signature of a `*LocalServiceImpl` class, `*ServiceImpl` class, or `*Impl` class, you should run Service Builder again to regenerate the affected interfaces and the service JAR.

Congratulations! You've generated your application's initial model, persistence, and service layers and you understand the generated code.

Related Topics

What is Service Builder

Running Service Builder and Understanding the Generated Code

Understanding Service Context

Creating Local Services

81.3 Iterative Development

During the course of development, you're likely to need to add fields to your database. This is a normal process of iterative development: you get an idea for a new feature, or it's suggested to you, and that feature requires additional data in the database. It's important to note, then, that **new fields added to `service.xml` are not automatically added to the database.** To add the fields, you must do one of two things:

1. Write an upgrade process to modify the tables and preserve the data, or

2. Run the `cleanServiceBuilder` Gradle task (also supported on Maven and Ant), which drops your tables so they get re-created the next time your app is deployed. See the Maven DB Support Plugin reference article for instructions on how to run this command from a Maven project.

Use the first option when you have a released application and you must preserve your users' data. Use the second option when you're in the middle of development and you're adding new columns during that process.

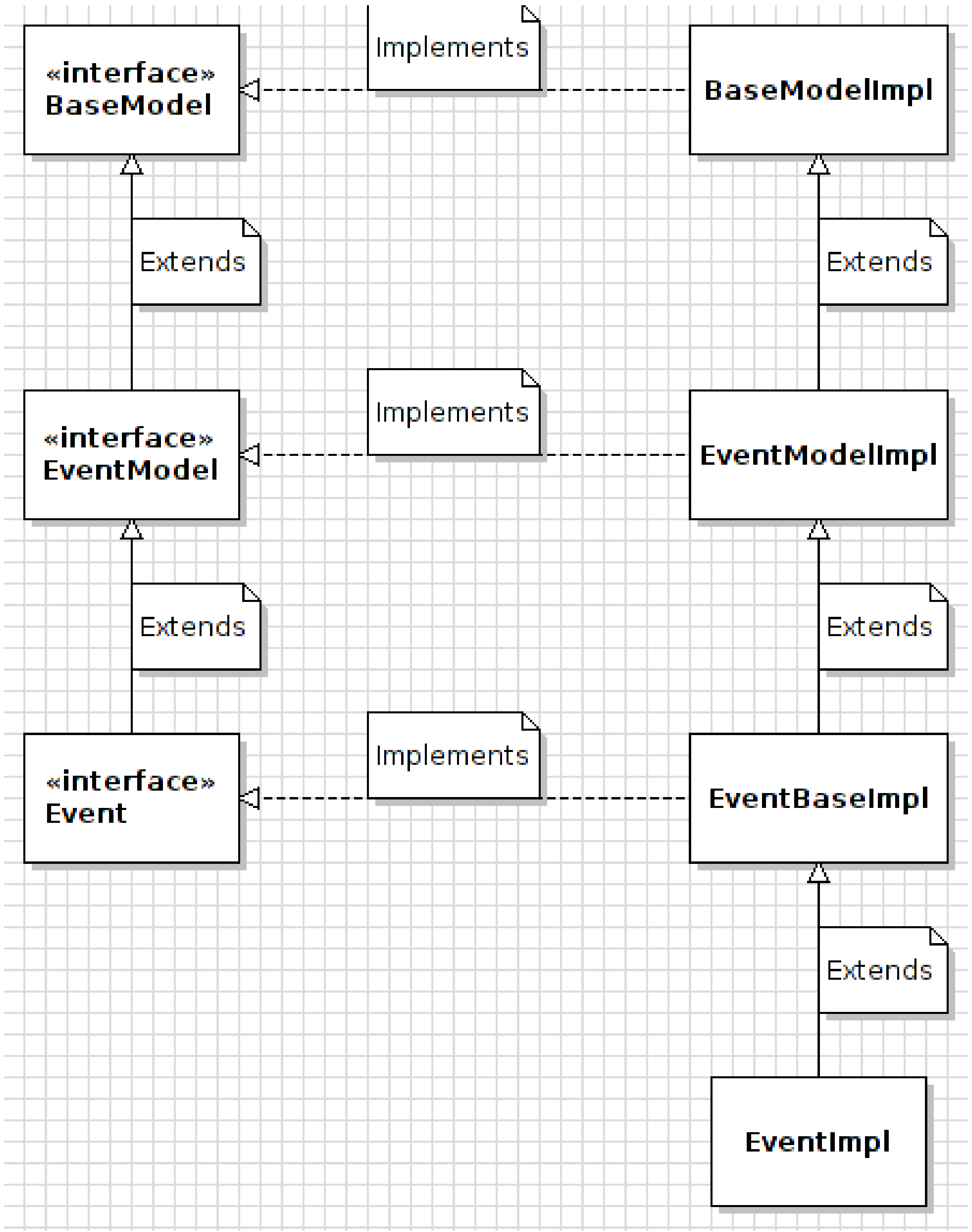


Figure 81.8: Service Builder generates these model classes and interfaces. Only `EventImpl` allows custom methods to be added to the service layer.

Related Topics

Upgrade Processes
    Gradle DB Support Plugin
    Maven DB Support Plugin

81.4 Understanding ServiceContext

The `ServiceContext` class is a parameter class used for passing contextual information for a service. Using a parameter class lets you consolidate many different methods with different sets of optional parameters into a single, easier-to-use method. The class also aggregates information necessary for features used throughout Liferay's core portlets, such as permissions, tagging, categorization, and more.

In this section, you'll examine the Service Context fields, learn how to create and populate a Service Context, and learn to access Service Context data. First, you'll look at the fields of the `ServiceContext` class.

Service Context Fields

The `ServiceContext` class has many fields. The best field descriptions are found in the Javadoc: @platform-ref@/7.0-latest/javadocs/portal-kernel/com/liferay/portal/kernel/service/ServiceContext.html. Here, you can review a categorical listing of the fields:

- Actions:
    - `_command`
    - `_workflowAction`
- Attributes:
    - `_attributes`
    - `_expandoBridgeAttributes`
- Classification:
    - `_assetCategoryIds`
    - `_assetTagNames`
- Exception
    - `_failOnPortalException`
- IDs and Scope:
    - `_companyId`
    - `_portletPreferencesIds`
    - `_plid`
    - `_scopeGroupId`
    - `_userId`
    - `_uuid`
- Language:

    - `_languageId`

- Miscellaneous:
    - `_headers`
    - `_signedIn`
- Permissions:
    - `_addGroupPermissions`
    - `_addGuestPermissions`
    - `_deriveDefaultPermissions`
    - `_modelPermissions`
- Request
    - `_request`
- Resources:
    - `_assetEntryVisible`
    - `_assetLinkEntryIds`
    - `_assetPriority`
    - `_createDate`
    - `_formDate`
    - `_indexingEnabled`
    - `_modifiedDate`
    - `_timeZone`
- URLs, paths and addresses:
    - `_currentURL`
    - `_layoutFullURL`
    - `_layoutURL`
    - `_pathMain`
    - `_pathFriendlyURLPrivateGroup`
    - `_pathFriendlyURLPrivateUser`
    - `_pathFriendlyURLPublic`
    - `_portalURL`
    - `_remoteAddr`
    - `_remoteHost`
    - `_userDisplayURL`

Are you wondering how the `ServiceContext` fields get populated? Good! You'll learn about that next.

Creating and Populating a Service Context

Although all the ServiceContext class fields are optional, services that store any kind of scopeable data need to at least specify the scope group ID. Here's a simple example of creating a ServiceContext instance and passing it as a parameter to a Liferay service API using Java:

```
ServiceContext serviceContext = new ServiceContext();
serviceContext.setScopeGroupId(myGroupId);

...

_blogsEntryService.addEntry(..., serviceContext);
```

If you invoke the service from a servlet, a Struts action, or any other front-end end class which has access to the PortletRequest, use one of the ServiceContextFactory.getInstance(...) methods. These methods create a ServiceContext object from the request and automatically populate its fields with all the values specified in the request. The above example looks different if you invoke the service from a servlet:

```
ServiceContext serviceContext =
    ServiceContextFactory.getInstance(BlogsEntry.class.getName(), portletRequest);

...

_blogsEntryService.addEntry(..., serviceContext);
```

You can see an example of populating a ServiceContext with information from a request object in the code of the ServiceContextFactory.getInstance(...) methods. The methods demonstrate how to set parameters like *scope group ID*, *company ID*, *language ID*, and more. They also demonstrate how to access and populate more complex context parameters like *tags*, *categories*, *asset links*, *headers*, and the *attributes* parameter. By calling ServiceContextFactory.getInstance(String className, PortletRequest portletRequest), you can assure that your Expando bridge attributes are set on the ServiceContext. Expandos are the back-end implementation of custom fields for entities in Liferay.

Creating and Populating a Service Context in JavaScript

Liferay's API can be invoked in languages other than Java. Some methods of Liferay's API require or allow a ServiceContext parameter. If you're invoking such a method via Liferay's JSON web services, you might want to create and populate a ServiceContext object in JavaScript. Creating a ServiceContext object in JavaScript is no different from creating any other object in JavaScript.

Before examining a JSON web service invocation that uses a ServiceContext object, it helps to see a simple JSON web service example in JavaScript:

```
Liferay.Service(
    '/user/get-user-by-email-address`,
    {
        companyId: Liferay.ThemeDisplay.getCompanyId(),
        emailAddress: 'test@example.com`
    },
    function(obj) {
        console.log(obj);
    }
);
```

If you run this code, the *test@example.com* user (JSON object) is logged to the JavaScript console.

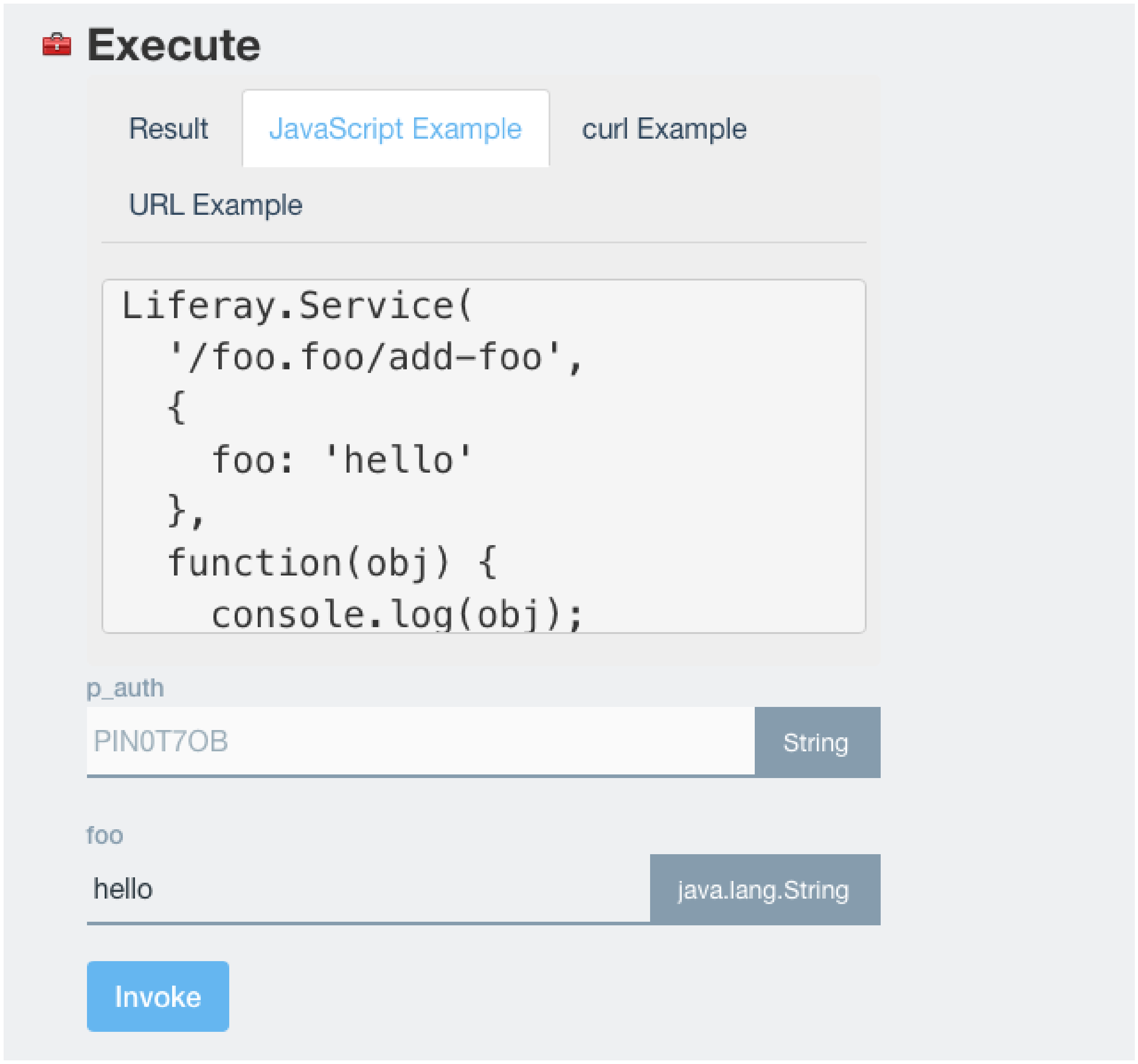
The Liferay.Service(...) function takes three arguments:

1. A string representing the service being invoked

2. A parameters object
3. A callback function

The callback function takes the result of the service invocation as an argument.

The Liferay JSON web services page (its URL is localhost:8080/api/jsonws if you're running Liferay locally on port 8080) generates example code for invoking web services. To see the generated code for a particular service, click on the name of the service, enter the required parameters, and click *Invoke*. The JSON result of your service invocation appears. There are multiple ways to invoke Liferay's JSON web services: click on *JavaScript Example* to see how to invoke the web service via JavaScript, click on *curl Example* to see how to invoke the web service via curl, or click on *URL example* to see how to invoke the web service via a URL.



```
Liferay.Service(
  '/foo.foo/add-foo',
  {
    foo: 'hello'
  },
  function(obj) {
    console.log(obj);
```

Figure 81.9: When you invoke a service from Liferay's JSON web services page, you can view the result of your service invocation as well as example code for invoking the service via JavaScript, curl, or URL.

To learn more about Liferay's JSON web services, see the JSON Web Services tutorial.

Next, you'll learn how to access information from a ServiceContext object.

Accessing Service Context Data

In this section, you'll find code snippets from BlogsEntryLocalServiceImpl.addEntry(..., ServiceContext). This code demonstrates how to access information from a ServiceContext and provides an example of how the context information can be used.

As mentioned above, services for scopeable entities need to get a scope group ID from the ServiceContext object. This is true for the Blogs entry service because the scope group ID provides the scope of the Blogs entry (the entity being persisted). For the Blogs entry, the scope group ID is used in the following way:

- It's used as the groupId for the BlogsEntry entity.
- It's used to generate a unique URL for the blog entry.
- It's used to set the scope for comments on the blog entry.

Here are the corresponding code snippets:

```
long groupId = serviceContext.getScopeGroupId();

...

entry.setGroupId(groupId);

...

entry.setUrlTitle(getUniqueUrlTitle(entryId, groupId, title));

...

// Message boards

if (PropsValues.BLOGS_ENTRY_COMMENTS_ENABLED) {
    mbMessageLocalService.addDiscussionMessage(
        userId, entry.getUserName(), groupId,
        BlogsEntry.class.getName(), entryId,
        WorkflowConstants.ACTION_PUBLISH);
}
```

Can ServiceContext be used to access the UUID of the blog entry? Absolutely! Can you use ServiceContext to set the time the blog entry was added? You sure can. See here:

```
entry.setUuid(serviceContext.getUuid());
...
entry.setCreateDate(serviceContext.getCreateDate(now));
```

Can ServiceContext be used in setting permissions on resources? You bet! When adding a blog entry, you can add new permissions or apply existing permissions for the entry, like this:

```
// Resources

if (serviceContext.isAddGroupPermissions() ||
    serviceContext.isAddGuestPermissions()) {

    addEntryResources(
        entry, serviceContext.isAddGroupPermissions(),
        serviceContext.isAddGuestPermissions());
}
else {
    addEntryResources(
        entry, serviceContext.getGroupPermissions(),
        serviceContext.getGuestPermissions());
}
```

ServiceContext helps apply categories, tag names, and the link entry IDs to asset entries too. Asset links are the back-end term for related assets in Liferay.

```
// Asset

updateAsset(
    userId, entry, serviceContext.getAssetCategoryIds(),
    serviceContext.getAssetTagNames(),
    serviceContext.getAssetLinkEntryIds());
```

Does ServiceContext also play a role in starting a workflow instance for the blogs entry? Must you ask?

```
// Workflow

if ((trackbacks ≠ null) && (trackbacks.length > 0)) {
    serviceContext.setAttribute("trackbacks", trackbacks);
}
else {
    serviceContext.setAttribute("trackbacks", null);
}

_workflowHandlerRegistry.startWorkflowInstance(
    user.getCompanyId(), groupId, userId, BlogsEntry.class.getName(),
    entry.getEntryId(), entry, serviceContext);
```

The snippet above also demonstrates the trackbacks attribute, a standard attribute for the blogs entry service. There may be cases where you need to pass in custom attributes to your blogs entry service. Use Expando attributes to carry custom attributes along in your ServiceContext. Expando attributes are set on the added blogs entry like this:

```
entry.setExpandoBridgeAttributes(serviceContext);
```

You can see that the ServiceContext can be used to transfer lots of useful information for your services. Understanding how ServiceContext is used in Liferay helps you determine when and how to use ServiceContext in your own Liferay application development.

Related Topics

Creating Local Services

Invoking Local Services

81.5 Customizing Model Entities With Model Hints

If you've already used Service Builder to define your model entities and have implemented business logic for creating and modifying those entities, you might have some ideas for helping users to submit valid model entity data. For example, suppose you're working on a calendar app where users can select a date for a calendar event. How can you specify that only future dates are selectable? Easy! Use portlet model hints! Liferay Service Builder's model hints provide a single place in your application where you can specify entity data restrictions. Model hints are specified in a single file called portlet-model-hints.xml in your project. If your project is comprised of an API module, a service module, and a web module, as is common for 7.0 applications, portlet-model-hints.xml should go in the service module. For example, in Liferay's Bookmarks application, the portlet-model-hints.xml file goes in the bookmarks-service/src/main/resources/META-INF/ folder. Model hints are referred to as such because they suggest how entities should be presented to users. Model hints let you to configure how the AlloyUI tag library, aui, shows model fields. As Liferay displays form fields in your application, it first checks the model hints you specified and customizes the form's input fields based on these hints. Model hints can also be used to specify the size of the database columns used to store the entities and to specify other entity details.

Note: Service Builder generates a number of XML configuration files in your service module's `src/main/resources/META-INF` folder. Service Builder uses most of these files to manage Spring and Hibernate configurations. Don't modify the Spring or Hibernate configuration files; your changes will be overwritten the next time Service Builder runs. However, you can safely edit the `portlet-model-hints.xml` file.

As an example, consider the Bookmarks app's model hints file:

```
<?xml version="1.0"?>

<model-hints>
    <model name="com.liferay.bookmarks.model.BookmarksEntry">
        <field name="uuid" type="String" />
        <field name="entryId" type="long" />
        <field name="groupId" type="long" />
        <field name="companyId" type="long" />
        <field name="userId" type="long" />
        <field name="userName" type="String" />
        <field name="createDate" type="Date" />
        <field name="modifiedDate" type="Date" />
        <field name="resourceBlockId" type="long" />
        <field name="folderId" type="long" />
        <field name="treePath" type="String">
            <hint name="max-length">4000</hint>
        </field>
        <field name="name" type="String">
            <hint name="max-length">255</hint>
        </field>
        <field name="url" type="String">
            <hint-collection name="URL" />
            <validator name="required" />
            <validator name="url" />
        </field>
        <field name="description" type="String">
            <hint-collection name="TEXTAREA" />
        </field>
        <field name="visits" type="int" />
        <field name="priority" type="int">
            <hint name="display-width">20</hint>
        </field>
        <field name="lastPublishDate" type="Date" />
        <field name="status" type="int" />
        <field name="statusByUserId" type="long" />
        <field name="statusByUserName" type="String" />
        <field name="statusDate" type="Date" />
    </model>
    <model name="com.liferay.bookmarks.model.BookmarksFolder">
        ...
    </model>
</model-hints>
```

The root-level element is `model-hints`. All the model entities are represented by `model` sub-elements of the `model-hints` element. Each `model` element must have a `name` attribute specifying the fully-qualified model class name. Each model has `field` elements representing its model entity's columns. Lastly, each `field` element must have a name and a type. Each `field` element's names and types correspond to the names and types specified for each entity's columns in the service module's `service.xml` file. Service Builder generates all these elements for you, based on `service.xml`.

To add hints to a field, add a `hint` tag inside its `field` tag. For example, you can add a `display-width hint` to specify the pixel width used to display the field. The default pixel width is 350. To show a `String` field with 50 pixels, you could nest a `hint` element named `display-width` and give it a value of `50`.

To see the effect of a hint on a field, you must run Service Builder again and redeploy your project. Note that changing `display-width` doesn't limit the number of characters a user can enter into the `name` field; it only controls the field's width in the AlloyUI input form.

To configure the maximum size of a model field's database column (i.e., the maximum number of characters that can be saved for the field), use the `max-length` hint. The default `max-length` value is 75 characters. If you want the `name` field to persist up to 100 characters, add a `max-length` hint to that field:

```
<field name="name" type="String">
    <hint name="display-width">50</hint>
    <hint name="max-length">100</hint>
</field>
```

Remember to run Service Builder and redeploy your project after updating the `portlet-model-hints.xml` file.

So far, you've seen a few different hints. The following table describes the portlet model hints available for use.

Model Hint Values and Descriptions

Name	Value Type	Description	Default
`auto-escape`	boolean	sets whether text values should be escaped via `HtmlUtil.escape`	true
`autoSize`	boolean	displays the field in a for scrollable text area	false
`day-nullable`	boolean	allows the day to be null in a date field	false
`default-value`	String	sets the default value of the form field rendered using the aui taglib	(empty String)
`display-height`	integer	sets the display height of the form field rendered using the aui taglib	15
`display-width`	integer	sets the display width of the form field rendered using the aui taglib	350
`editor`	boolean	sets whether to provide an editor for the input	false
`max-length`	integer	sets the maximum column size for SQL file generation	75
`month-nullable`	boolean	allows the month to be null in a date field	false
`secret`	boolean	sets whether to hide the characters input by the user	false
`show-time`	boolean	sets whether to show the time along with the date	true
`upper-case`	boolean	converts all characters to upper case	false
`year-nullable`	boolean	allows a date field's year to be null	false
`year-range-delta`	integer	specifies the number of years to display from today's date in a date field rendered with the aui taglib	5
`year-range-future`	boolean	sets whether to include future dates	true
`year-range-past`	boolean	sets whether to include past dates	true

Note that Liferay has its own `portal-model-hints.xml` file. This file contains many hint examples, so you can reference it when customizing your `portlet-model-hints.xml` file.

You can use the `default-hints` element to define a list of hints to apply to every field of a model. For example, adding the following element inside a model element applies a `display-width` of 300 to each field:

```
<default-hints>
    <hint name="display-width">300</hint>
</default-hints>
```

You can define `hint-collection` elements inside the `model-hints` root-level element to define a list of hints to apply together. A hint collection must have a name. For example, Liferay's `portal-model-hints.xml` defines the following hint collections:

```
<hint-collection name="CLOB">
    <hint name="max-length">2000000</hint>
</hint-collection>
<hint-collection name="EDITOR">
    <hint name="editor">true</hint>
    <hint name="max-length">2000000</hint>
</hint-collection>
<hint-collection name="SEARCHABLE-DATE">
    <hint name="month-nullable">true</hint>
    <hint name="day-nullable">true</hint>
    <hint name="year-nullable">true</hint>
    <hint name="show-time">false</hint>
</hint-collection>
<hint-collection name="TEXTAREA">
    <hint name="display-height">105</hint>
    <hint name="display-width">500</hint>
    <hint name="max-length">4000</hint>
</hint-collection>
<hint-collection name="URL">
    <hint name="max-length">4000</hint>
</hint-collection>
```

You can apply a hint collection to a model field by referencing the hint collection's name. For example, if you define a `SEARCHABLE-DATE` collection like the one above in your `model-hints` element, you can apply it to your model's date field by using a `hint-collection` element that references the collection by its name:

```
<field name="date" type="Date">
    <hint-collection name="SEARCHABLE-DATE" />
</field>
```

As always, remember to run Service Builder and redeploy your project after updating your `portlet-model-hints.xml` file.

For example, suppose you want to use a couple of model hints in your project. Start by providing users with an editor for filling in their comment fields. To apply the same hint to multiple entities, define it as a hint collection. Then reference the hint collection in each entity.

To define a hint collection, add a `hint-collection` element inside the `model-hints` root element in your `portlet-model-hints.xml` file. For example:

```
<hint-collection name="COMMENT-TEXTAREA">
    <hint name="display-height">105</hint>
    <hint name="display-width">500</hint>
    <hint name="max-length">4000</hint>
</hint-collection>
```

To reference a hint collection for a specific field, add the `hint-collection` element inside the field's `field` element:

```
<field name="comment" type="String">
    <hint-collection name="COMMENT-TEXTAREA" />
</field>
```

After defining hint collections and adding hint collection references, rebuild your services using Service Builder, redeploy your project, and check that the hints defined in your hint collection have taken effect.

Nice work! You've learned the art of persuasion through Liferay's model hints. Now you can not only influence how your model's input fields are displayed, but you can also can set its database table column sizes. You can organize hints, insert individual hints directly into your fields, apply a set of default hints to all of a model's fields, or define collections of hints to apply at either of those scopes. You've picked up the "hints" on how Liferay model hints specify how to display app data!

81.6 Custom SQL

Service Builder creates finder methods that retrieve entities by their attributes: their column values. When you add a column as a parameter for the finder in your `service.xml` file and run Service Builder, it generates the finder method in your persistence layer and adds methods to your service layer that invoke the finder. If your queries are simple enough, consider using Dynamic Query to access Liferay's database. If you want to do something more complicated like JOINs, you can write your own custom SQL queries. You'll learn how in this tutorial.

Say you have a Guestbook application with two tables, one for guestbooks and one for guestbook entries. The entry entity's foreign key to its guestbook is the guestbook's ID. That is, the entry entity table, `GB_Entry`, tracks an entry's guestbook by its long integer ID in the table's `guestbookId` column. If you want to find a guestbook entry based on its name, message, and guestbook name, you must access the *name* of the entry's guestbook. Of course, with SQL you can join the entry and guestbook tables to include the guestbook name. Service Builder lets you do this by specifying the SQL as *Liferay custom SQL* and invoking it in your service via a *custom finder method*.

Liferay custom SQL is a Service Builder-supported method for performing custom, complex queries against the database by invoking custom SQL from a finder method in your persistence layer. Service Builder helps you generate the interfaces to your finder method. It's easy to do by following these steps:

1. Specify your custom SQL.
2. Implement your finder method.
3. Access your finder method from your service.

Next, using the Guestbook application as an example, you'll learn how to accomplish these steps.

Step 1: Specify Your Custom SQL

After you've tested your SQL, you must specify it in a particular file for Liferay to access it. `CustomSQLUtil` class (from module `com.liferay.portal.dao.orm.custom.sql`) retrieves SQL from a file called `default.xml` in your service module's `src/main/resources/META-INF/custom-sql/` folder. You must create the `custom-sql` folder and create the `default.xml` file in that `custom-sql` folder. The `default.xml` file must adhere to the following format:

```
<custom-sql>
    <sql id="[fully-qualified class name + method]">
    SQL query wrapped in <![CDATA[...]]>
    No terminating semi-colon
    </sql>
</custom-sql>
```

Create a `custom-sql` element for every SQL query you want in your application, and give each query a unique ID. The recommended convention to use for the ID value is the fully-qualified class name of the finder followed by a dot (`.`) character and the name of the finder method. More detail on the finder class and finder methods is provided in Step 2.

For example, in the Guestbook application, you could use the following ID value to specify a query:

```
com.liferay.docs.guestbook.service.persistence.\
EntryFinder.findByEntryNameEntryMessageGuestbookName
```

Custom SQL must be wrapped in character data (CDATA) for the sql element. Importantly, do not terminate the SQL with a semi-colon. Following these rules, the default.xml file of the Guestbook application specifies an SQL query that joins the GB_Entry and GB_Guestbook tables:

```
<?xml version="1.0" encoding="UTF-8"?>
<custom-sql>
    <sql id="com.liferay.docs.guestbook.service.persistence.EntryFinder.findByEntryNameEntryMessageGuestbookName">
        <![CDATA[
            SELECT GB_Entry.*
            FROM GB_Entry
            INNER JOIN
                GB_Guestbook ON GB_Entry.guestbookId = GB_Guestbook.guestbookId
            WHERE
                (GB_Entry.name LIKE ?) AND
                (GB_Entry.message LIKE ?) AND
                (GB_Guestbook.name LIKE ?)
        ]]>
    </sql>
</custom-sql>
```

Now that you've specified some custom SQL, the next step is to implement a finder method to invoke it. The method name for the finder should match the ID you just specified for the sql element.

Step 2: Implement Your Finder Method

Next, implement the finder method in your persistence layer to invoke your custom SQL query. Service Builder generates the interface for the finder in your API module but you must create the implementation.

The first step is to create a *FinderImpl class in the service persistence package. For the Guestbook application, you could create a EntryFinderImpl class in the com.liferay.docs.guestbook.service.persistence.impl package. Your class should extend BasePersistenceImpl<Entry>.

Run Service Builder to generate the *Finder interface based on the *FinderImpl class. Modify your *FinderImpl class to make it a component (annotated with @Component) that implements the *Finder interface you just generated:

```
@Component(service = EntryFinder.class)
public class EntryFinderImpl extends BasePersistenceImpl<Event>
    implements EntryFinder {

}
```

Now you can create a finder method in your EntryFinderImpl class. Add your finder method and static field to the *FinderImpl class. For example, here's how you could write the EntryFinderImpl class:

```
public List<Entry> findByEntryNameEntryMessageGuestbookName(
    String entryName, String entryMessage, String guestbookName,
    int begin, int end) {

    Session session = null;
    try {
        session = openSession();

        String sql = CustomSQLUtil.get(
            getClass(),
            FIND_BY_ENTRYNAME_ENTRYMESSAGE_GUESTBOOKNAME);

        SQLQuery q = session.createSQLQuery(sql);
        q.setCacheable(false);
        q.addEntity("GB_Entry", EntryImpl.class);

        QueryPos qPos = QueryPos.getInstance(q);
        qPos.add(entryName);
        qPos.add(entryMessage);
        qPos.add(guestbookName);

        return (List<Entry>) QueryUtil.list(q, getDialect(), begin, end);
    }
    catch (Exception e) {
        try {
            throw new SystemException(e);
        }
        catch (SystemException se) {
            se.printStackTrace();
        }
    }
    finally {
        closeSession(session);
    }

    return null;
}

public static final String FIND_BY_ENTRYNAME_ENTRYMESSAGE_GUESTBOOKNAME =
    EntryFinder.class.getName() +
        ".findByEntryNameEntryMessageGuestbookName";
```

The custom finder method opens a new Hibernate session and uses Liferay's `com.liferay.portal.dao.orm.custom.sql.Cust` `clazz, String id)` method to get the custom SQL to use for the database query. The `FIND_BY_ENTRYNAME_ENTRYMESSAGE_GUESTBOO` static field contains the custom SQL query's ID. The `FIND_BY_EVENTNAME_EVENTDESCRIPTON_LOCATIONNAME` string is based on the fully-qualified class name of the `*Finder` interface (`EventFinder`) and the name of the finder method (`findByEntryNameEntryMessageGuestbookName`).

Awesome! Your custom SQL is in place and your finder method is implemented. Next, you'll call the finder method from your service.

Step 3: Access Your Finder Method from Your Service

So far, you've created a `*FinderImpl` class, generated the `*Finder` interface, and created a custom finder method that gets your custom SQL. Your last step is to add a service method that calls your finder.

When you ran Service Builder after defining your custom finder method, the `*Finder` interface was generated (e.g., `GuestbookFinder`). Your portlet class, however, should not call the `*Finder` interface: only a local or remote service implementation (i.e., `*LocalServiceImpl` or `*ServiceImpl`) in your service module should invoke the `*Finder` class. This encourages a proper separation of concerns: the portlet classes in your application's web module invoke the business logic of the services published from your application's service module. The services, in turn, access the data model using the persistence layer's finder classes.

So you'll add a method in the `*LocalServiceImpl` class that invokes the finder method implementation via the `*Finder` class. Then you'll rebuild your application's service layer so that the portlet classes and JSPs in your web module can access the services.

For example, for the Guestbook application, you'd add the following method to the `EntryLocalServiceImpl` class:

```
public List<Entry> findByEntryNameGuestbookName(String entryName,
    String guestbookName) throws SystemException {

    return entryFinder.findByEntryNameGuestbookName(String entryName,
        String guestbookName);
}
```

After you've added your `findBy-` method to your `*LocalServiceImpl` class, run Service Builder to generate the interface and make the finder method available in the `EntryLocalService` class.

Now you can indirectly call the finder method from your portlet class or a JSP in your web module. For example, to call the finder method in the Guestbook application, just call `entryLocalService.findByEntryNameEntryMessageGuestb`

Congratulations on developing a custom SQL query and custom finder for your application!

Related Topics

Customizing Liferay Services

Service Builder Web Services

81.7 Dynamic Query

Liferay lets you use custom SQL queries to retrieve data from the database. However, it's sometimes more convenient to build queries dynamically at runtime than it is to invoke predefined SQL queries. Liferay allows you to build queries dynamically using its DynamicQuery API, which wraps Hibernate's Criteria API. Using Liferay's DynamicQuery API allows you to build queries without writing a single line of SQL. The DynamicQuery API helps you think in terms of objects and member variables instead of in terms of tables and columns. Complex queries constructed via Hibernate's Criteria API can be significantly easier to understand and maintain than the equivalent custom SQL (or HQL) queries. While you technically don't need to know SQL to construct queries via Hibernate's Criteria API, you still need to take care to construct efficient queries. For information on Hibernate's Criteria API, please see Hibernate's manual. In this tutorial, you'll learn how to create custom finders for Liferay applications using Service Builder and Liferay's Dynamic Query API.

To use Liferay's Dynamic Query API, you need to create a finder implementation for your model entity. You can define model entities in `service.xml` and run Service Builder to generate model, persistence, and service layers for your application. See the Running Service Builder and Understanding the Generated Code learning path for more information on using Service Builder. This tutorial assumes that you're creating a Liferay application that consists of a service module, an API module, and a web module. Once you've used Service Builder to generate model, persistence, and service layers for your application, you can call custom finders using Liferay's Dynamic Query API by following these steps:

1. Create a custom `-FinderImpl` class and define a `findBy-` finder method in this class. Run Service Builder to generate the required interfaces and utility classes.

2. Implement your finder method using Liferay's Dynamic Query API.

3. Add a method to your `-LocalServiceImpl` class that invokes your finder method. Run Service Builder to add the required method to the service interface.

Once you've taken these steps, you can access your custom finder as a service method. Note: You can create multiple or overloaded `findBy-` finder methods in your `-FinderImpl` class. Next, let's examine these steps in more detail.

Step 1: Defining a Custom Finder Method

To define any custom query, either by specifying custom SQL or by defining a dynamic query, you need a finder class. Create a `[Entity]FinderImpl` class in the generated `[package path].service.persistence.impl` package of your service module's `src/main/java` folder. Recall that you specify the package path in `service.xml`. Here's an example:

```
<service-builder package-path="com.liferay.docs.guestbook">
    ...
</service-builder>
```

Then define a `findBy-` finder method in the class you created. Make sure to add any required arguments to your finder method's method signature.

For example, consider a fictitious Guestbook application. In this application, there are two entities: guestbooks and entries. Each entry belongs to a guestbook so the entry entity has a `guestbookId` field as a foreign key. Suppose you need to create a custom finder to search for guestbook entries by the entry name and the guestbook name. In this case, you'd add a custom finder method to `GuestbookFinderImpl` and name it something like `findByEntryNameGuestbookName`. The full method signature would appear as `findByEntryNameGuestbookName(String entryName, String guestbookName)`.

Once you've created a finder method with the appropriate method signature in your finder class, run Service Builder to generate the appropriate interface in the `[package path].service.persistence` package in the `service` folders of your API and service modules.

For example, after adding `findByEntryNameGuestbookName(String entryName, String guestbookName)` to `GuestbookFinderImpl` and running Service Builder, the interface `com.liferay.docs.guestbook.service.persistence.GuestbookF` is generated.

Once the finder interface has been generated, make sure that the finder class implements the interface. For example, the class declaration should look like this:

```
public class GuestbookFinderImpl extends BasePersistenceImpl<Guestbook> implements GuestbookFinder
```

Your next step is to actually define your query in your custom finder method using the Dynamic Query API.

Step 2: Implementing Your Custom Finder Method Using Dynamic Query

Your first step in implementing your custom finder method in your `-FinderImpl` class is to open a new Hibernate session. Since your `-FinderImpl` class extends `BasePersistenceImpl<Entity>`, and `BasePersistenceImpl<Entity>` contains a session factory object and an `openSession` method, you can simply invoke the `openSession` method of your `-FinderImpl`'s parent class to open a new Hibernate session. The basic structure of your finder method should look like this:

```
public List<Entity> findBy-(...) {

    Session session = null;
    try {
            /*
            Try to open a new Hibernate session and create a dynamic
            query to retrieve and return the desired list of entity
            objects
            */
    }
    catch (Exception e) {
            // Exception handling
    }
    finally {
            closeSession(session);
    }

    return null;
    /*
    Return null only if there was an error returning the
    desired list of entity objects in the try block
    */

}
```

For example, in the case of the Guestbook application, you could write the following finder method to retrieve a list of Guestbook entries that have a specific name and that also belong to a Guestbook of a specific name:

```
public List<Event> findByEntryNameGuestbookName(String entryName, String guestbookName) {

    Session session = null;
    try {
        session = openSession();

        ClassLoader classLoader = getClass().getClassLoader();

        DynamicQuery guestbookQuery = DynamicQueryFactoryUtil.forClass(Guestbook.class, classLoader)
            .add(RestrictionsFactoryUtil.eq("name", guestbookName))
            .setProjection(ProjectionFactoryUtil.property("guestbookId"));

        Order order = OrderFactoryUtil.desc("modifiedDate");

        DynamicQuery entryQuery = DynamicQueryFactoryUtil.forClass(Entry.class, classLoader)
            .add(RestrictionsFactoryUtil.eq("name", entryName))
            .add(PropertyFactoryUtil.forName("guestbookId").in(guestbookQuery))
            .addOrder(order);

        List<Event> entries = EventLocalServiceUtil.dynamicQuery(entryQuery);

        return entries;
    }
    catch (Exception e) {
        try {
            throw new SystemException(e);
        }
        catch (SystemException se) {
            se.printStackTrace();
        }
    }
    finally {
        closeSession(session);
    }
```

Notice that in Liferay, you don't create criteria objects directly from the Hibernate session. Instead, you create dynamic query objects using Liferay's DynamicQueryFactoryUtil service. Thus, instead of

```
Criteria entryCriteria = session.createCriteria(Entry.class);
```

you use

```
DynamicQuery entryQuery = DynamicQueryFactoryUtil.forClass(Entry.class,
    classLoader);
```

Most features of Hibernate's Criteria API, including restrictions, projections, and orders, can be used on Liferay's dynamic query objects. Restrictions in Hibernate's Criteria API roughly correspond to the `where` clause of an SQL query: they offer a variety of ways to limit the results returned by the query. You can use restrictions, for example, to cause a query to return only results where a certain field has a particular value, or a value in a certain range, or a non-null value, etc.

Projections in Hibernate's Criteria API let you modify the kind of results returned by a query. For example, if you don't want your query to return a list of entity objects (the default), you can set a projection on a query so that only a list of the values of a certain entity field, or fields, is returned. You can also use projections on a query to return the maximum or minimum value of an entity field, or the sum of all the values of a field, or the average, etc. For more information on restrictions and projections, please refer to Hibernate's documentation.

Orders, another feature of Hibernate's Criteria API, let you control the order of the elements in the list returned by a query. You can choose the property or properties to which an order should be applied and you can choose for the properties to appear in ascending or descending order in the list.

Like Hibernate criteria, Liferay's dynamic queries are *chainable*. This means that you can add criteria to, set projections on, and add orders to Liferay's dynamic query objects just by appending the appropriate method calls to the query object. For example, the following snippet demonstrates chaining the addition of a restriction criterion and a projection to a dynamic query object declaration:

```
DynamicQuery guestbookQuery = DynamicQueryFactoryUtil.forClass(Guestbook.class, classLoader)
    .add(RestrictionsFactoryUtil.eq("name", guestbookName))
    .setProjection(ProjectionFactoryUtil.property("guestbookId"));
```

When you need to add restrictions to a dynamic query in Liferay, don't call Hibernate's `Restrictions` class directly. Instead, use the methods of Liferay's RestrictionsFactoryUtil service. You'll find the same methods in Liferay's `RestrictionsFactoryUtil` service class that you're used to from Hibernate's `Restrictions` class: `in`, `between`, `like`, `eq`, `ne`, `gt`, `ge`, `lt`, `le`, etc.

Thus, instead of

```
entryCriteria.add(Restrictions.eq("name", guestbookName));
```

to specify that a guestbook must have a certain name, you use

```
entryQuery.add(RestrictionsFactoryUtil.eq("name", guestbookName));
```

Similarly, to set projections, you create properties via Liferay's PropertyFactoryUtil service instead of through Hibernate's `Property` class. Thus, instead of

```
entryCriteria.setProjection(Property.forName("guestbookId"));
```

you use

```
entryQuery.setProjection(PropertyFactoryUtil.forName("guestbookId"));
```

Notice that in the custom `findByGuestbookNameEntryName` finder method, there are two distinct dynamic queries. The first query retrieves a list of guestbook IDs corresponding to guestbook names that match the `guestbookName` parameter of the finder method. The second query retrieves a list of guestbook entries with entry names that match the `entryName` parameter and have `guestbookId` foreign keys belonging to the list returned by the first query.

Here's the first query:

```
DynamicQuery guestbookQuery = DynamicQueryFactoryUtil.forClass(Guestbook.class, classLoader)
    .add(RestrictionsFactoryUtil.eq("name", guestbookName))
    .setProjection(ProjectionFactoryUtil.property("guestbookId"));
```

By default, `DynamicQueryFactoryUtil.forClass(Guestbook.class, classLoader)` returns a query that retrieves a list of all guestbook entities. Adding the `.add(RestrictionsFactoryUtil.eq("name", guestbookName))` restriction limits the results to only those guestbooks whose guestbook names match the `guestbookName` parameter. The `.setProjection(ProjectionFactoryUtil.property("guestbookId"))` projection changes the result set from a list of guestbook entries to a list of guestbook IDs. This is useful since guestbook IDs are much less expensive to retrieve than full guestbook entities and the guestbook IDs are all that the guestbook entry query requires.

Next is an order which applies to the list of entries returned by the `findByEntryNameGuestbookName` finder method:

```
Order order = OrderFactoryUtil.desc("modifiedDate");
```

When this order is applied to a query, the list of results returned by the query are arranged in descending order of the query entity's `modifiedDate` attribute. Thus the most recently modified entities (guestbook entries, in our example) appear first and the least recently modified entities appear last.

Here's the second query:

```
DynamicQuery eventQuery = DynamicQueryFactoryUtil.forClass(Entry.class, classLoader)
    .add(RestrictionsFactoryUtil.eq("name", entryName))
    .add(PropertyFactoryUtil.forName("guestbookId").in(guestbookQuery))
    .addOrder(order);

List<Event> entries = _eventLocalService.dynamicQuery(entryQuery);
```

By default, `DynamicQueryFactoryUtil.forClass(Entry.class, classLoader)` returns of list of all guestbook entry entities. The `.add(RestrictionsFactoryUtil.eq("name", entryName))` restriction limits the results to only those guestbook entries whose names match the `entryName` parameter of the finder method. PropertyFactoryUtil is a Liferay utility class with the method `forName(String propertyName)`, which returns the specified property. This property can be passed to another Liferay dynamic query. This is exactly what happens in the following line of our example:

```
.add(PropertyFactoryUtil.forName("guestbookId").in(guestbookQuery))
```

Here, the code makes sure that the guestbook IDs (foreign keys) of the entry entities in the `entityQuery` must belong to the list of guestbook IDs returned by the `guestbookQuery`. Declaring that an entity property in one query must belong to the result list of another query is a way to use Liferay's dynamic query API to create complex queries, similar to SQL joins.

Lastly, you apply the order defined earlier to the entries returned by the `findByEntryNameGuestbookName` finder method:

```
.addOrder(order);
```

This orders the list of guestbook entities by the `modifiedDate` attribute, from most recent to least recent.

Note: Service Builder not only generates a `public List dynamicQuery(DynamicQuery dynamicQuery)` method in `-LocalServiceBaseImpl` but it also generates `public List dynamicQuery(DynamicQuery dynamicQuery, int start, int end)` and `public List dynamicQuery(DynamicQuery dynamicQuery, int start, int end, OrderByComparator orderByComparator)` methods. You can go back to step 1 and either modify your custom finder method or create overloaded versions of your custom finder method to take advantage of these extra methods and their parameters. The `int start` and `int end` parameters are useful when paginating a result list. `start` is the lower bound of the range of model entity instances and `end` is the upper bound. The `OrderByComparator orderByComparator` is the comparator by which to order the results.

To use the overloaded `dynamicQuery` methods of your `-LocalServiceBaseImpl` class in the (optionally overloaded) custom finders of your `-FinderImpl` class, just choose the appropriate methods for running the dynamic queries: `EventLocalService.dynamicQuery(eventQuery)`, or `EventLocalService.dynamicQuery(eventQuery, start, end)` or `EventLocalService.dynamicQuery(eventQuery, start, end, orderByComparator)`.

Great! You've now created a custom finder method using Liferay's Dynamic Query API. Your last step is to add a service method that calls your finder.

Step 3: Accessing Your Custom Finder Method from the Service Layer

So far, you've created a `-FinderImpl` class, defined a custom `findBy-` finder method in that class, and implemented the custom finder method using Dynamic Query. Now how do you call your custom finder method from the service layer?

When you ran Service Builder after defining your custom finder method, the `-Finder` interface was generated (e.g., `GuestbookFinder`). Your portlet class, however, should not call the `-Finder` interface: only a local or remote service implementation (i.e., `-LocalServiceImpl` or `-ServiceImpl`) in your service module should invoke the `-Finder` class. This encourages a proper separation of concerns: the portlet classes in your application's web module invoke the business logic of the services published from your application's service module. The services, in turn, access the data model using the persistence layer's finder classes.

Note: In previous versions of Liferay Portal, your finder methods were accessible via `-FinderUtil` utility classes. Finder methods are now injected into your app's local services, removing the need to call finder utilities.

So you'll add a method in the `-LocalServiceImpl` class that invokes the finder method implementation via the `-Finder` class. Then you'll rebuild your application's service layer so that the portlet classes and JSPs in your web module can access the services.

For example, for the Guestbook application, you'd add the following method to the `EntryLocalServiceImpl` class:

```
public List<Entry> findByEntryNameGuestbookName(String entryName,
    String guestbookName) throws SystemException {

    return entryFinder.findByEntryNameGuestbookName(String entryName,
        String guestbookName);
}
```

After you've added your `findBy-` method to your `-LocalServiceImpl` class, run Service Builder to generate the interface and make the finder method available in the `EntryLocalService` class.

Now you can indirectly call the finder method from your portlet class or from a JSP by calling `EntryLocalService.findByEntryNameGuestbookName(...)`!

Congratulations on following the three step process of developing a dynamic query in a custom finder and exposing it as a service for your portlet!

Actionable Dynamic Queries

Suppose you have over a million users on your portal, and you want to perform some kind of mass update to a large portion of them. One approach might be to use a dynamic query to retrieve the list of users in question. Once loaded into memory, you could loop through the list and update each user. However, with over a million users, the memory cost of such an operation would be too great. In general, if you have very large numbers of service builder entities, it can be too expensive in terms of memory and speed to run a dynamic query to retrieve a list of such entities in order to do some processing on them.

Liferay provides actionable dynamic queries to solve this kind of situation. An actionable dynamic query does not return a list of service builder entities like a regular dynamic query. Instead, it uses a pagination strategy to load only small numbers of service builder entities into memory at a time and performs some processing (i.e., performs an *action*) on each entity. So instead of trying to use a dynamic query to load a million users into memory and then perform some processing on each of them, a much better strategy is to use an actionable dynamic query to process them. This way, only small numbers of users are loaded into memory at a time, but you still process all the users.

When you run Service Builder, it includes actionable dynamic query support in the generated API and service modules. For example, consider the API module of the BLADE service builder example project.

The FooLocalService interface in the API module contains these methods:

```
@Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
public ActionableDynamicQuery getActionableDynamicQuery();

public DynamicQuery dynamicQuery();

@Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
public ExportActionableDynamicQuery getExportActionableDynamicQuery(
    PortletDataContext portletDataContext);

@Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
public IndexableActionableDynamicQuery getIndexableActionableDynamicQuery();
```

The FooLocalServiceBaseImpl class in the service module implements each of these methods. See here and here for details.

The implementation of FooLocalService.dynamicQuery() uses DynamicQueryFactoryUtil to obtain new (regular) dynamic query instance for the Foo entity. This is the pattern shown earlier.

You can use FooLocalService.getActionableDynamicQuery() to obtain a new actionable dynamic query instance for the Foo entity. Once you have the instance, you can use chaining to build up the query using any of the techniques described above for regular dynamic queries, such as restrictions or projections. But the point of using an *actionable* dynamic query is to specify an action to perform on each entity that results from the query. To specify such an action, use the ActionableDynamicQuery.setPerformActionMethod<PerformActionMethod<?> method. Once the query has been defined and an action has been specified, use the ActionableDynamicQuery.performActions to perform the action on each entity that results from the query.

Here's an example from a test for Liferay DXP's Bookmarks application:

```
ActionableDynamicQuery actionableDynamicQuery = BookmarksEntryLocalServiceUtil.getActionableDynamicQuery();

actionableDynamicQuery.setPerformActionMethod(new ActionableDynamicQuery.PerformActionMethod<BookmarksEntry>() {
        @Override
        public void performAction(BookmarksEntry bookmarksEntry) {
            Assert.assertNotNull(bookmarksEntry);

            count.increment();
        }
    });

actionableDynamicQuery.performActions();
```

You can see the full context here.

Consider the FooLocalService from the BLADE service builder API project again. For most of your actionable dynamic query use cases, the actionable query returned by FooLocalService.getActionableDynamicQuery will suffice. This actionable dynamic query is an instance of the concrete class DefaultActionableDynamicQuery. However, in addition to FooLocalService.getActionableDynamicQuery, there are two additional methods related to actionable dynamic queries: FooLocalService.getExportActionableDynamicQuery and FooLocalService.getIndexableActionableDynamicQuery. These methods return instances of concrete classes (either IndexableActionableDynamicQuery or ExportActionableDynamicQuery) that extend DefaultActionableDynamicQuery. IndexableActionableDynamicQuery contains methods designed to facilitate processing that involves search indexing and ExportActionableDynamicQuery contains methods designed to facilitate processing that involves export / import functionality.

To see examples of configuring indexer and export actionable dynamic queries, see the Bookmarks application here and here. To see an example invocation of an indexable actionable dynamic query, see the `reindexEntries` method of the Bookmarks application's indexer here.

Related Topics

Service Builder Web Services
Creating Local Service
Invoking Local Services

81.8 Configuring service.properties

In this tutorial, you'll learn how to use and edit the `service.properties` file. You'll also learn about the properties included in this file and how to set them to fit your needs.

Service Builder generates a `service.properties` file in your `*-service` module's `src/main/resources` folder. Liferay DXP uses the properties in this file to alter your service's database schema. You should not modify this file, but rather make any necessary overrides in a `service-ext.properties` file in that same folder.

Here are some of the properties included in the `service.properties` file:

- `build.namespace`: This is the namespace you defined in your `service.xml`. Liferay distinguishes different plugins from each other using their namespaces.
- `build.number`: Liferay distinguishes different builds of your plugin. Each time a distinct build of your plugin is deployed to Liferay, Liferay increments this number.
- `build.date`: This is the time of the latest build of your plugin.
- `include-and-override`: The default value of this property defines `service-ext.properties` as an override file for `service.properties`.

Note: The `build.auto.upgrade` property is available for WAR-style Service Builder applications. This property determines whether or not Liferay should automatically apply changes to the database model when a new version of the plugin is deployed. This is `true` by default. This property is not necessary for module-style applications.

It's sometimes useful to override the `build.auto.upgrade` property in legacy projects from `service.properties`. Setting `build.auto.upgrade=false` in your `service-ext.properties` file prevents Liferay from trying automatically to apply any changes to the database model when a new version of the plugin is deployed. This is needed in projects to manually manage the changes to the database (recommended) or in which the SQL schema has intentionally been modified manually after generation by Service Builder.

Awesome! You now have all the tools necessary to set up your own `service-ext.properties` file.

Related Topics

What is Service Builder?
Creating Local Services

81.9 Connecting Service Builder to External Data Sources

Sometimes you want to use a data source other than Liferay DXP's. To do this, the data source must be defined in `portal-ext.properties` or configured as a JNDI data source on Liferay DXP's app server. This tutorial shows how to connect Service Builder to a data source.

Note: All entities defined in a Service Builder module's `service.xml` file are bound to the same data source. Binding different entities to different data sources requires defining the entities in separate Service Builder modules and configuring each of the modules to use a different data source.

Here are the steps:

1. If Liferay DXP's application server defines the data source using JNDI, skip this step. Otherwise, specify the data source in a `portal-ext.properties` file. Distinguish it from Liferay DXP's default data source by giving it a prefix other than `jdbc.default.`. This example uses prefix `jdbc.ext.`:

```
jdbc.ext.driverClassName=org.mariadb.jdbc.Driver
jdbc.ext.password=userpassword
jdbc.ext.url=jdbc:mariadb://localhost/external?useUnicode=true&characterEncoding=UTF-8&useFastDateParsing=false
jdbc.ext.username=yourusername
```

2. Create a Spring bean that points to the data source. To do this, create an `ext-spring.xml` file in your Service Builder module's `src/main/resources/META-INF/spring` folder or in your traditional portlet's `WEB-INF/src/META-INF` folder. Define the following elements:

   - A data source factory Spring bean for the data source. It's different based on the type:

     - **JNDI**: Specify an arbitrary property prefix and prepend the prefix to a JNDI name property key. Here's an example:

```
<bean class="com.liferay.portal.dao.jdbc.spring.DataSourceFactoryBean"
    id="liferayDataSourceFactory">
    <property name="propertyPrefix" value="custom." />
    <property name="properties">
        <props>
            <prop key="custom.jndi.name">jdbc/externalDataSource</prop>
        </props>
    </property>
</bean>
```

     - **Portal Properties**: Specify a property prefix that matches the prefix (e.g., `jdbc.ext.`) you used in `portal-ext.properties`.

```
<bean class="com.liferay.portal.dao.jdbc.spring.DataSourceFactoryBean"
    id="liferayDataSourceFactory">
    <property name="propertyPrefix" value="jdbc.ext." />
</bean>
```

   - A Liferay DXP data source bean that refers to the data source factory Spring bean.
   - An alias for the Liferay DXP data source bean.

   Here's an example `ext-spring.xml` that points to a JNDI data source:

```
<?xml version="1.0"?>

<beans default-destroy-method="destroy" default-init-method="afterPropertiesSet"
   xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd">

   <!-- To define an external data source, the liferayDataSource Spring bean
       must be overridden. Other default Spring beans like liferaySessionFactory
       and liferayTransactionManager may optionally be overridden.

       liferayDataSourceFactory refers to the data source configured on the
       application server. -->
   <bean class="com.liferay.portal.dao.jdbc.spring.DataSourceFactoryBean"
       id="liferayDataSourceFactory">
       <property name="propertyPrefix" value="custom." />
       <property name="properties">
           <props>
               <prop key="custom.jndi.name">jdbc/externalDataSource</prop>
           </props>
       </property>
   </bean>

   <!-- The data source bean refers to the factory to access the data source.
   -->
   <bean
       class="org.springframework.jdbc.datasource.LazyConnectionDataSourceProxy"
       id="liferayDataSource">
       <property name="targetDataSource" ref="liferayDataSourceFactory" />
   </bean>

   <!-- In service.xml, we associated our entity with the extDataSource. To
       associate the extDataSource with our overridden liferayDataSource, we define
       this alias. -->
   <alias alias="extDataSource" name="liferayDataSource" />
</beans>
```

The `liferayDataSourceFactory` above refers to a JNDI data source named `jdbc/externalDataSource`. If the data source was specified via data source properties in a `portal-ext.properties` file, the bean would require only a `propertyPrefix` property that matches the data source property prefix.

The data source bean `liferayDataSource` is overridden with one that refers to the `liferayDataSourceFactory` bean. The override affects this bundle (module or Web Application Bundle) only.

The alias `extDataSource` refers to the `liferayDataSource` data source bean.

```
**Note**: To use an external data source in multiple Service Builder
bundles, you must override the `liferayDataSource` bean in each bundle.
```

---

2. In your Service Builder module's `service.xml` file, set your entity's data source to the `liferayDataSource` alias you specified in your `ext-spring.xml` file. Here's an example:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.0.0//EN"
    "http://www.liferay.com/dtd/liferay-service-builder_7_0_0.dtd">

<service-builder package-path="com.liferay.example" >
    <namespace>TestDB</namespace>
    <entity local-service="true" name="Foo" table="testdata" data-source="extDataSource"
```

```
            remote-service="false" uuid="false">
           <column name="id" db-name="id" primary="true" type="long" />
           <column name="foo" db-name="foo" type="String" />
           <column name="bar" db-name="bar" type="long" />
    </entity>
</service-builder>
```

Note the example's `<entity>` tag attributes:

- `data-source`: The `liferayDataSource` alias `ext-spring.xml` specifies.
- `table`: Your entity's database table.

Also note that your entity's `<column>`s must have a `db-name` attribute set to the column name.

3. Run Service Builder.

Now your Service Builder services use the data source. You can use the services in your business logic as you always have regardless of the underlying data source.

Related Topics

Connecting to JNDI Data Sources
Service Builder
Running Service Builder and Understanding the Generated Code
Business Logic with Service Builder

Chapter 82

Business Logic with Service Builder

Once you've defined your application's entities and ran Service Builder to generate your service and persistence layers, you can begin adding business logic. Each entity generated by Service Builder contains a model implementation, local service implementation, and optionally a remote service implementation class. Your application's business logic can be implemented in these classes. The generated service layer contains default methods for CRUD operations, but often times it's necessary to implement additional methods. Once you've added your business logic, running Service Builder again regenerates your application's interfaces and makes your new logic available for invocation.

In this section of tutorials, you'll learn about creating and invoking your application's local services, finding and invoking Liferay's services, and customizing Liferay services.

82.1 Creating Local Services

The heart of your service is its `*LocalServiceImpl` class. This class is your entity's local service extension point. Local services can be invoked within your application or by other Liferay applications running on the same Liferay instance as your application. Remote services differ from local services in that remote services can be invoked from any application that can access your Liferay instance (e.g., over the Internet) and has permission to do so. All of your application's core business logic for working with your entity model (or models) should be added as methods of your `*LocalServiceImpl` class. Before adding any custom service methods, however, you should review the initial service classes that Service Builder generated during its initial run.

Best Practice: If your application needs both local and remote services, determine the service methods that your application needs for working with your entity model. Add these service methods to `*LocalServiceImpl`. Then create corresponding remote services methods in `*ServiceImpl`. Add permission checks to the remote service methods and make the remote service methods invoke the local service methods. The remote service methods can have the same names as the local service methods that they call. Within your application, only call the remote services. This ensures that your service methods are secured and that you don't have to duplicate permissions code.

Note that Service Builder creates a `*LocalService` class which is the interface for the local service. It contains the signatures of every method in `*LocalServiceBaseImpl` and `*LocalServiceImpl`. `*LocalServiceBaseImpl` contains some automatically generated methods that provide functionality that's common to all local services. Since the `*LocalService` class is generated, you should never modify it. If you do, your changes will be overwritten the next time you run Service Builder. All custom code should be placed in *LocalServiceImpl, where it will not be overwritten.

For example, the Bookmarks application's BookmarksEntryLocalServiceImpl class demonstrates the kinds of service methods that applications commonly need for working with an entity model. Click on the class's link to view some of its local service methods.

In order to add an entity to the database, you need an ID for the entity. Liferay provides a counter service which you call to obtain a unique ID for each new entity. It's possible to use the increment method of Liferay's CounterLocalService class, but Service Builder already makes a CounterLocalService instance available to your app's *LocalServiceBaseImpl. The CounterLocalService instance is injected into a module as an OSGi service:

```
@ServiceReference(type=com.liferay.counter.kernel.service.CounterLocalService.class)
protected com.liferay.counter.kernel.service.CounterLocalService counterLocalService;
```

If you're creating local services in Liferay's core, the CounterLocalService instance is injected as a Spring bean:

```
@BeanReference(type=com.liferay.counter.kernel.service.CounterLocalService.class)
protected com.liferay.counter.kernel.service.CounterLocalService counterLocalService;
```

Since your *LocalServiceImpl class extends *LocalServiceBaseImpl, you can access this CounterLocalService instance. See your app's *LocalServiceBaseImpl for a list of all the Spring beans/OSGi services you have available for use.

You can use either the injected class's increment method or you can call Liferay's CounterLocalService's increment method directly. For example, a bookmarks entry is assigned a unique ID like this:

```
long entryId = counterLocalService.increment();
```

The Bookmarks application uses the generated entryId as the ID for the new BookmarksEntry:

```
BookmarksEntry entry = bookmarksEntryPersistence.create(entryId);
```

bookmarksEntryPersistence is one of the OSGi services injected into EventLocalServiceBaseImpl by Service Builder.

Next, the Bookmarks application sets the attribute fields that were specified for the BookmarksEntry entity in the service.xml. These attributes include the groupId, userId, name, url, serviceContext, etc. Lastly, a Bookmarks folder ID must be associated to the entry.

It's also important to assign values to the audit fields. In the Bookmarks application, the group of the entity is set first. An entity's group determines its scope. In this example, the group is the site. The company and user are specified after the group is set. The company represents the portal instance and the user is the user who created the bookmark. The Bookmarks application sets the createDate and modifiedDate of the Event to the current time. After that, the generated addEntry method of BookmarksEntryLocalServiceBaseImpl is called to add the bookmark to the database. Lastly, the bookmark is added as a resource so that permissions can be applied to it later. To view the addEntry method in its entirety, see the BookmarksEntryLocalServiceImpl class.

The Bookmarks application creates local services for BookmarksFolder entities as well as for BookmarksEntry entities. Take a look at the custom service methods available in the BookmarksFolderLocalServiceImpl class for a better understanding of services available for bookmark folders.

Before you can use any custom methods that you added to your *LocalServiceImpl class, you must run Service Builder again. Running Service Builder again adds the method signatures of your custom service methods to your *LocalService interface and updates your *LocalServiceUtil class. For more information on running Service Builder see the Running Service Builder and Understanding the Generated Code tutorial.

Service Builder looks through your *LocalServiceImpl class and automatically copies the signatures of each method into the corresponding *LocalService interface. After running Service Builder, you can test that your services are working as intended by invoking one of the methods that Service Builder added to your *LocalService class. For example, if you were developing the Bookmarks application, you could make the following service invocation to make sure that your service was working as intended:

```
BookmarksEntryLocalService.addBookmarksEntry(bookmarksEntry);
```

In addition to all of the Java classes and interfaces, Service Builder also generates a service.properties file. To learn about the service.properties file and how to configure it, please refer to the Configuring service.properties tutorial. To learn how to invoke local services, please refer to the Invoking Local Services tutorial.

Related Topics

Running Service Builder and Understanding the Generated Code
Invoking Local Services
Creating Remote Services

82.2 Invoking Local Services

In this tutorial, you'll learn about the differences between local and remote services, and when you should invoke local services rather than their remote service counterparts.

Once Service Builder has generated your module project's services, you can call them from anywhere in your application. When invoking a local service, you should call one of the service methods of a *LocalService class. You should never call the service methods of an *Impl class directly. Local services in your project are generated automatically when using Service Builder. To do this, set the local-service attribute to true for an entity in the service.xml file. Service Builder generates methods that call existing services, but you can create new methods in the *LocalServiceImpl class that can be generated into new exposed methods in your module's services (*LocalService, *LocalServiceUtil, etc.).

Many of Liferay's module applications use their generated remote services for important calls in their controller layer, for example, because they offer conveniences like configured permissions checking. Remote services perform a permission check and then invoke the corresponding local service. However, there are many services you'd like to expose only to your local project, and do not want other applications to have access to.

In the Bookmarks application, for example, the BookmarksEntryLocalService interface provides the openEntry method, which opens a bookmark for viewing. The remote service interface BookmarksEntryService, however, does not provide this method. Why could this be?

There are many services that you don't want to expose to other applications in Liferay. In the example mentioned above, Bookmarks are configured to only open locally, meaning that other apps do not have access to open and view a bookmark entry. This should only be done by the Bookmarks application. Therefore, in certain cases, you'll need to invoke local services instead of remote services. For more information on invoking remote services, see the Invoking Remote Services tutorial.

To see how you could call a local service from a portlet action class, you'll examine the EditOrganizationMVCActionCommand class. Notice that this class has a private instance variable called _dlAppLocalService. The _dlAppLocalService instance variable of type DLAppLocalService gets an instance of DLAppLocalService at runtime via dependency injection. The instance variable is set like this:

```
@Reference(unbind = "-")
protected void setDLAppLocalService(DLAppLocalService dlAppLocalService) {
    _dlAppLocalService = dlAppLocalService;
}
```

This tutorial demonstrated how you can call the local services generated by Service Builder in your project. To learn how to call Liferay services, see the Service Security Layers and Finding and Invoking Liferay Services tutorials.

Related Topics

Creating Local Services
Creating Remote Services
Invoking Remote Services

82.3 Finding and Invoking Liferay Services

In this tutorial, you'll learn how to search for portal services and portlet services. You can find Liferay's services by searching for them in the Javadocs: @platform-ref@/7.0-latest/javadocs/.

First, you'll learn how to find a portal service using Liferay's Javadocs.

Finding Liferay Portal Services

Searching for Liferay Portal services is easy and intuitive. The first two options, `portal-impl` and `portal-kernel`, are the most popular options when searching for Liferay's Javadocs. In summary, the `portal-kernel` directory provides interfaces and utils, and the `portal-impl` directory provides service implementations that implement those interfaces. The remaining options are miscellaneous util and test classes that are used in Liferay DXP.

Liferay's Javadocs are easy to browse and well-organized. Here's how to find the *Organization* services, for example:

1. In your browser, open up the Javadocs: @platform-ref@/7.0-latest/javadocs/ You're offered several options, which were discussed earlier. Select *portal-kernel*.
2. Under *Portal Kernel*, click on the link for the `com.liferay.portal.kernel.service` package, since the services for the Organization entity belong to the *Portal* scope.
3. Find and click on the `*LocalService` class (in this case, `OrganizationLocalService`) in the *Class Summary* table or the *Classes* list at the bottom of the page.

That was easy! What if you want to find module services?

Finding Liferay Module Services

Searching for Liferay module services is also easy. The Javadocs for modules are hosted on Liferay's Nexus repository, and can be viewed by downloading and extracting the module's `*javadoc.jar` file. You can learn move about how a module's Java API is organized by reading the Java APIs section.

Here's an example of how to find services for a bookmarks entry:

1. Navigate to Liferay's Nexus repository and select `com.liferay.bookmarks.api`. Then select the appropriate version.

2. Select the com.liferay.bookmarks.api-[VERSION]-javadoc.jar link, which downloads that JAR file. Extract the JAR file, once downloaded.

3. Open the extracted contents and select the index.html file.

4. Select the com.liferay.bookmarks.service package from the main view, and then select the BookmarksEntryLocalService class in the *Class Summary* table or the *Classes* list.

Awesome! You've successfully located the bookmark entry's services.

Another easy way to search for services in module projects is by importing them into your IDE. For Liferay @ide@, you can right-click in the Package Explorer and navigate to *Import → Liferay Module Project(s)*. Then browse for your module, select the build type, and click *Finish*. Now you can peruse your module services from Liferay @ide@.

Now you're ready to invoke Liferay services.

Invoking Liferay Services Locally

Every Liferay service provides a local interface to clients running in the same JVM as Liferay Portal. Many local services (e.g., *LocalService classes) are called by remote services (e.g., *Service classes). The remote classes mask the complexity of the local service implementations and include permission checks. The core Liferay services that are provided as part of Liferay Portal were generated by the same Service Builder tool that you can use in your applications. You can invoke a remote Liferay service by calling the appropriate *LocalService or *Service class. The following code found in the journal-content-web module demonstrates how to retrieve the portal instance's group by calling Liferay's GroupLocalService:

```
Group group = _groupLocalService.getCompanyGroup(companyId);
```

By Liferay convention, the _groupLocalService instance variable is created and set for usage in the class it's called from:

```
private GroupLocalService _groupLocalService;

@Reference(unbind = "-")
protected void setGroupLocalService(GroupLocalService groupLocalService) {
    _groupLocalService = groupLocalService;
}
```

The @Reference(unbind="-") annotation retrieves a reference to a service of type GroupLocalService, and ignores how this service has been published and who published it.

Besides the services Service Builder made available for your application, you can also access any service published within the OSGi Registry. This means the following services are available:

- Beans defined in Liferay's core
- Beans created in other module app contexts
- Services declared using Declarative Service
- Services registered using the OSGi low level API

Some types of portlets don't have access to the OSGi Registry using Declarative Services (e.g., Spring MVC and JSF). You can call OSGi services in these portlets by using Service Trackers.

You'll learn more about referencing OSGi services next.

Referencing OSGi Services

All the services created within your Service Builder application are wired using an internal Spring Application Context. This uses AOP proxies to give your services the ability to deal with transactions, indexing, and security.

In many cases, however, you'll need to reference an external service (i.e., something that is not defined within your Spring Application Context). Liferay has included the ability to reference OSGi services from your Spring beans using the `@ServiceReference` annotation.

You'll step through a simple example next.

Suppose you're building an application with a simple entity defined in your `service.xml` file. The application needs to send an SMS every time a new entity is created, and the `SMSService` is provided by a module installed in the system.

How would you get a reference from the `-LocalServiceImpl` (Spring bean) to an *external* service? You can do this by using the `@ServiceReference` annotation:

```
@ServiceReference
private SMSService _smsService;
```

Using this annotation lets you retrieve a reference to an OSGi service from a regular Spring bean. This provides some nice benefits. If the `SMSService` is not available, the whole Spring context is not created until the service is available. Likewise, if the `SMSService` suddenly disappears, the whole Spring Application Context is destroyed. This makes our Spring apps much more robust and versatile.

Fortunately, Service Builder generates this kind of code every time you have a reference to an entity which is not defined in your `service.xml` file. For example, imagine that your entity has a reference provided in your `service.xml` file, to the Group entity:

```
<reference entity="Group" package-path="com.liferay.portal" />
```

The generated code for this entity would look like the following:

```
@ServiceReference(type = com.liferay.portal.kernel.service.GroupLocalService.class)
protected com.liferay.portal.kernel.service.GroupLocalService groupLocalService;
```

Great! You know how to find Liferay's core and module services, and can invoke them from your application. You also learned about referencing OSGi services.

Related Topics

Invoking Local Services

Invoking Remote Services

JSON Web Services Invoker

Service Trackers

Chapter 83

Data Access

Liferay DXP's data can be retrieved using its APIs. It's important, however, to understand how data is used amid all of Liferay DXP's constructs. At a basic level, all the data is represented by an object model. This is retrieved from the database and automatically mapped from SQL to the model by Service Builder. Using Model Listeners, you can listen for events on these models and take action when they are stored or retrieved.

All data in Liferay DXP is scoped to a context: a site, a page, or global. When storing data in your application, you should therefore take advantage of scope so your application integrates well with the system and users can add your application in whatever scope they need it.

Read on to learn how to do both of these things.

83.1 Data Scopes

Apps in Liferay DXP can restrict their data to specific scopes. Scopes provide a context for the application's data. For example, a site-scoped app can display its data across a single site. For a detailed explanation of scopes, see the user guide article Application Scope. To give your applications scope, you must manually add support for it. This tutorial shows you how.

Scoping Your Entities

In your service layer, your entities must have a `companyId` attribute of type `long` to enable scoping by portal instance and a `groupId` attribute of type `long` to enable scoping by site. Using Service Builder is the simplest way to do this. For instructions on this, see the tutorial series Service Builder Persistence and Business Logic with Service Builder.

Enabling Scoping

To enable scoping in your app, set the property `"com.liferay.portlet.scopeable=true"` in your portlet class's `@Component` annotation. For example, this property is set to `true` in the `@Component` annotation of Web Content Display Portlet's portlet class:

```
@Component(
    immediate = true,
    property = {
        ...
        "com.liferay.portlet.scopeable=true",
        ...,
    },
    service = Portlet.class
)
public class JournalContentPortlet extends MVCPortlet {...
```

That's it! You can now access your app's scope in your code. The next section shows you how.

Accessing Your App's Scope

Users can typically set an app's scope to a page, a site, or the entire portal. To handle your app's data, you must access it in its current scope. Liferay DXP gives you techniques to do this. Your app's scope is available:

1. Via the `scopeGroupId` variable that is injected in your JSPs whenever you use the `<liferay-theme:defineObjects />` tag. This variable contains your app's current scope. For example, Liferay's Bookmarks app uses `scopeGroupId` in its `view.jsp` to retrieve the bookmarks and total number of bookmarks in the current scope:

   ```
   ...
   total = BookmarksEntryServiceUtil.getGroupEntriesCount(scopeGroupId, groupEntriesUserId);

   bookmarksSearchContainer.setTotal(total);
   bookmarksSearchContainer.setResults(BookmarksEntryServiceUtil.getGroupEntries(scopeGroupId, groupEntriesUserId, bookmarksSearchContainer.getStart()
   ...
   ```

2. By calling the `getScopeGroupId()` method on the request's `ThemeDisplay` instance. This method returns your app's current scope. For example, the `EditEntryMVCActionCommand` class in Liferay's Blogs app does this in its `subscribe` and `unsubscribe` methods:

   ```
   protected void subscribe(ActionRequest actionRequest) throws Exception {
       ThemeDisplay themeDisplay = (ThemeDisplay)actionRequest.getAttribute(
           WebKeys.THEME_DISPLAY);

       _blogsEntryService.subscribe(themeDisplay.getScopeGroupId());
   }

   protected void unsubscribe(ActionRequest actionRequest) throws Exception {
       ThemeDisplay themeDisplay = (ThemeDisplay)actionRequest.getAttribute(
           WebKeys.THEME_DISPLAY);

       _blogsEntryService.unsubscribe(themeDisplay.getScopeGroupId());
   }
   ```

   If you know your app always needs the portal instance ID, use `themeDisplay.getCompanyId()`.

3. By calling the `getScopeGroupId()` method on a `ServiceContext` object. You can find more information on this, and an example, in the tutorial Understanding `ServiceContext`. If you know your app always needs the portal instance ID, use the `getCompanyId()` method on a `ServiceContext` object.

Awesome! Now you know how to get your app's scope. Next, you'll learn about a special use case: getting the site scope for entities that belong to a different app.

Accessing the Site Scope Across Apps

There may be times when you need to access a different app's site-scoped data, even though your app may be scoped to a page or the portal. For example, web content articles can be created in the page, site, or portal scope. Structures and Templates for such articles, however, can only exist in the site scope. If you use the above techniques to retrieve the app's current scope, you won't always get the site scope required for Structures and Templates. You might get the page or portal scope instead, if that's how the user configured your app. What a pickle! Never fear, the `ThemeDisplay` method `getSiteGroupId()` is here! This method always gets the site scope, no matter your app's current scope. For example, the Web Content app's `edit_feed.jsp` uses this method to get the site ID needed to retrieve Structures:

...

```
ddmStructure = DDMStructureLocalServiceUtil.fetchStructure(themeDisplay.getSiteGroupId(),
    PortalUtil.getClassNameId(JournalArticle.class), ddmStructureKey, true);
```

...

Great! Now you know how to scope your apps, access their scope, and even get the site scope of entities that belong to other apps. Now that's minty-fresh breath!

Related Topics

Application Scope

What is Service Builder?

Service Builder Persistence

Business Logic with Service Builder

Chapter 84

Web Services

Generating and invoking web services is a central part of the Liferay development experience. After all, what good is it if you can't generate remote services for an app you're developing, or call Liferay's built-in remote services? It's no good at all! Liferay without web services would be like a luxury car without wheels. Fortunately, Liferay comes with full set of JSON and SOAP web services that you can invoke until your heart's content. Liferay also provides Service Builder to generate local and remote services for your apps.

The tutorials that follow show you how to work with web services in Liferay. You'll learn how to use Service Builder to generate remote services for your apps. You'll also learn how to invoke those services, and any other Liferay web services.

Chapter 85

Service Builder Web Services

Liferay's Service Builder can generate local and remote services for your Liferay apps. The section of tutorials on Service Builder gives a general introduction to Service Builder, as well as instructions on generating your app's local services. But what if you want to use Service Builder to generate your app's remote services? And how should you invoke remote services generated by Service Builder? No sweat! That's exactly what the tutorials here cover. This section shows you how to use Service Builder to create JSON and SOAP web services, and how to invoke those services. Because Liferay's own developers use Service Builder to generate JSON and SOAP web services, knowing how to invoke these services is especially important. This section includes the following tutorials:

- Creating Remote Services: Use Service Builder to generate your app's JSON and SOAP web services.
- Invoking Remote Services: Learn the basics of invoking JSON and SOAP web services in Liferay.
- Service Security Layers: Learn how Liferay secures web services, and how to invoke them with proper authentication.
- Registering JSON Web Services: Learn some of the details behind how Service Builder generates JSON web services, and how you can tailor this process to your needs.
- Invoking JSON Web Services: Learn how to invoke Liferay's JSON web services API via URL. This includes information on passing URL parameters, troubleshooting, and more.
- JSON Web Services Invoker: Learn how to use Liferay's JSON Web Services Invoker to optimize your JSON web service calls.
- JSON Web Services Invocation Examples: See examples of how to invoke Liferay's JSON web services via JavaScript, URL, and cURL.
- Configuring JSON Web Services: Learn which properties you can use to control how JSON web services behave in your Liferay instance.
- SOAP Web Services: Learn how SOAP web services work in Liferay.

85.1 Creating Remote Services

Many default Liferay DXP services are available as web services. Liferay DXP exposes its web services via JSON and SOAP web services. If you're running the portal locally on port 8080, visit the following URL to browse Liferay DXP's default JSON web services:

```
http://localhost:8080/api/jsonws/
```

To browse Liferay DXP's default SOAP web services, visit this URL:

```
http://localhost:8080/api/axis
```

These web services APIs can be accessed by many different kinds of clients, including non-portlet and even non-Java clients. You can use Service Builder to generate similar remote services for your projects' custom entities. When you run Service Builder with the `remote-service` attribute set to `true` for an entity, all the classes, interfaces, and files required to support both SOAP and JSON web services are generated for that entity. Service Builder generates methods that call existing services, but it's up to you to implement the methods that are exposed remotely. In this tutorial, you'll learn how to generate remote services for your application. When you're done, your application's remote service methods can be called remotely via JSON and SOAP web services.

Using Service Builder to Generate Remote Services

Remember that you should implement your application's local service methods in `*LocalServiceImpl`. You should implement your application's remote service methods in `*ServiceImpl`.

Best Practice: If your application needs both local and remote services, determine the service methods that your application needs for working with your entity model. Add these service methods to `*LocalServiceImpl`. Then create corresponding remote services methods in `*ServiceImpl`. Add permission checks to the remote service methods and make the remote service methods invoke the local service methods. The remote service methods can have the same names as the local service methods that they call. Within your application, only call the remote services. This ensures that your service methods are secured and that you don't have to duplicate permissions code.

As an example, consider Liferay DXP's Blogs app. Articles are represented by the `JournalArticle` entity. This entity is declared in the `journal-service` module's `service.xml` file with the `remote-service` attribute set to `true`. Service Builder therefore generates the remote service class `JournalArticleServiceImpl` to hold the remote service method implementations. If you were developing this app from scratch, this class would initially be empty; you must use it to implement the entity's remote service methods. Also, note that the remote service method implementations in `JournalArticleServiceImpl` follow best practice by checking permissions and calling the corresponding local service method. For example, each `addArticle` method in `JournalArticleServiceImpl` checks permissions via the custom permissions class `JournalFolderPermission` and then calls the local service's matching `addArticle` method:

```
@Override
public JournalArticle addArticle(...)
    throws PortalException {

    JournalFolderPermission.check(
        getPermissionChecker(), groupId, folderId, ActionKeys.ADD_ARTICLE);

    return journalArticleLocalService.addArticle(...);
}
```

You'll also need to develop custom permissions classes for each entity you need to perform permissions checks on. Also note that the local service is called via the `journalArticleLocalService` field. This is a Spring bean of type `JournalArticleLocalServiceImpl` that's injected into `JournalArticleServiceImpl` by Service Builder. See the class `JournalArticleServiceBaseImpl` for a complete list of Spring beans available in `JournalArticleServiceImpl`.

After you've finished adding remote service methods to your `*ServiceImpl` class, save it and run Service Builder again. After running Service Builder, deploy your project and check the Liferay DXP JSON web services URL http://localhost:8080/api/jsonws/ to make sure that your remote services appear when you select your application's context path.

Nice work! You've successfully used Service Builder to generate your app's remote services. To make these services available via SOAP, however, you must build and deploy your app's Web Service Deployment Descriptor (WSDD). The next section shows you how to do this. If you don't need to generate SOAP web services, you can move on to the tutorial Invoking Remote Services.

Generating Your App's WSDD

Liferay DXP uses Apache Axis to make SOAP web services available. Since Axis requires a WSDD to make an app's remote services available via SOAP, you must build and deploy a WSDD for your app. To create your WSDD, you must install Liferay's WSDD Builder Gradle plugin in your app's project. How you do this, however, depends on what kind of project you have. For multi-module projects like a Service Builder project in a Liferay Workspace, you'll install the plugin via the workspace's `settings.gradle` file. This applies the WSDD Builder plugin to every module in the workspace that uses Service Builder (typically the `*-api` and `*-service` modules). If you have a standalone `*-service` module that uses Service Builder, however, you'll install the WSDD Builder plugin in the module's `build.gradle` file.

The next section shows you how to install the WSDD builder in a multi-module project. If you have a standalone module project, skip ahead to the section *Installing the WSDD Builder Plugin in a Standalone Module Project*.

Installing the WSDD Builder Plugin in a Multi-module Project

To install the WSDD Builder plugin in a multi-module project like a Service Builder project in a Liferay Workspace, do the following in the workspace's `settings.gradle` file:

1. Add the `ServiceBuilderPlugin` and `WSDDBuilderPlugin` imports to the top of the file:

   ```
   import com.liferay.gradle.plugins.service.builder.ServiceBuilderPlugin
   import com.liferay.gradle.plugins.wsdd.builder.WSDDBuilderPlugin
   ```

2. In the `repositories` block, add the Liferay CDN repository via Maven:

   ```
   repositories {
       maven {
           url "https://repository-cdn.liferay.com/nexus/content/groups/public"
       }
   }
   ```

   This repository hosts the WSDD Builder library, its transitive dependencies, and other Liferay libraries. Note that if you created your Service Builder project with the `service-builder` template in Blade CLI or Liferay @ide@, then your `settings.gradle` file should already contain this.

3. Add this code to the end of the file:

```
gradle.beforeProject {
    project ->

    project.plugins.withType(ServiceBuilderPlugin) {
        project.apply plugin: WSDDBuilderPlugin
    }
}
```

This is the code that applies the WSDD Builder plugin in every module in the Liferay Workspace that uses Service Builder. Your `settings.gradle` file should now look something like this:

```
import com.liferay.gradle.plugins.service.builder.ServiceBuilderPlugin
import com.liferay.gradle.plugins.wsdd.builder.WSDDBuilderPlugin

buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.workspace", version: "1.2.0"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.workspace"

gradle.beforeProject {
    project ->

    project.plugins.withType(ServiceBuilderPlugin) {
        project.apply plugin: WSDDBuilderPlugin
    }
}
```

4. Refresh the Liferay Workspace's Gradle project. Close and restart Liferay @ide@ if you're using it.

Now that you've installed the WSDD Builder plugin, you're ready to build and deploy the WSDD. For instructions on this, proceed to the section *Building and Deploying the WSDD*.

Installing the WSDD Builder Plugin in a Standalone Module Project

To install the WSDD Builder plugin in a standalone `*-service` module that uses Service Builder, do the following in the module's `build.gradle` file:

1. Add the plugin as a dependency in your `buildscript`.
2. Add the Liferay CDN repository via Maven.
3. Apply the plugin to your project.

For example, the following part of an example `build.gradle` file in a standalone `*-service` module includes the WSDD Builder plugin and applies it to the project:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.wsdd.builder", version: "1.0.9"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.portal.tools.wsdd.builder"
```

Now you're ready to build and deploy the WSDD. The next section shows you how to do this.

Building and Deploying the WSDD

To build the WSDD, you must run the `buildWSDD` Gradle task in your `*-service` module. Exactly how you do this depends on your development tools:

- **Liferay @ide@:** From the Liferay Workspace perspective's *Gradle Tasks* pane (typically on the right), open your `*-service` module's *build* folder and double-click *buildWSDD*.
- **Command Line:** Navigate to your `*-service` module and run `../../../gradlew buildWSDD`. Note that the exact location of the Gradle wrapper (`gradlew`) may vary. For Liferay Workspace projects, it's typically in the root workspace folder.

So what should you do if `buildWSDD` fails? A common cause of `buildWSDD` failures is failing to satisfy the dependencies needed by the WSDD Builder for your `*-service` module. Note that these dependencies vary depending on your project's code–there's no standard set. That said, the following are often required for portlet development:

```
compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
compileOnly group: "com.liferay", name: "com.liferay.registry.api", version: "1.0.0"
```

Click here for more information on finding and configuring dependencies for your apps.

In your `*-service` project's `build/libs` folder, the `buildWSDD` task generated a `*-service-wsdd-[version].jar` file that contains your WSDD. Deploy this JAR to your Liferay DXP instance. Your SOAP web services are then available at a URL that uses the following pattern:

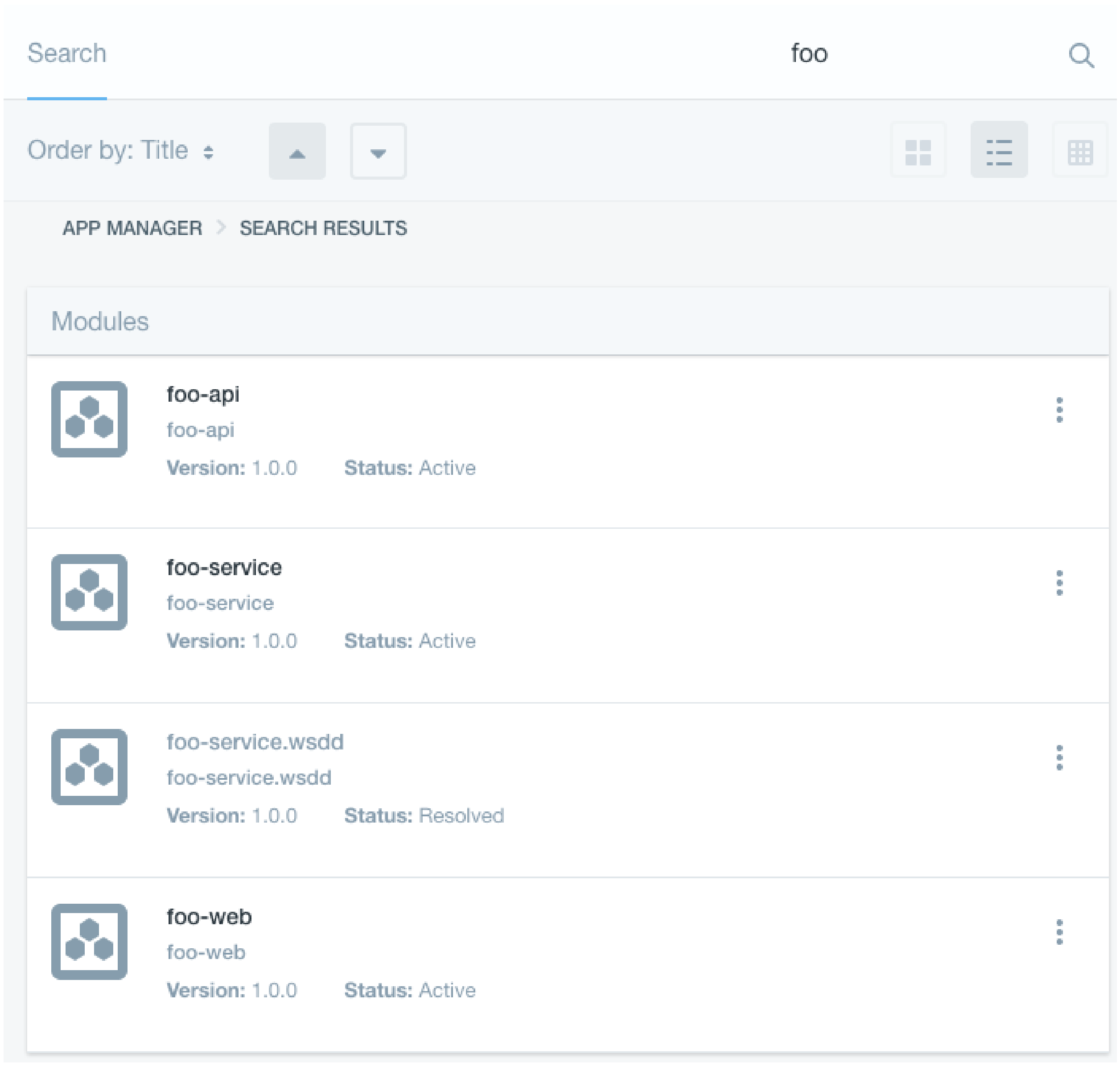
```
yourportaladdress/o/your.apps.service.module.context/api/axis
```

For example, if an app called *Foo* consists of the modules `foo-api`, `foo-service`, and `foo-web`, then the app's service module context is `foo-service`. If this app is deployed to a local Liferay DXP instance running at `http://localhost:8080`, you could access its SOAP services at:

```
http://localhost:8080/o/foo-service/api/axis
```

If you don't know an app's `*-service` module context, you can find it by searching for the app in the App Manager of the Liferay DXP instance in which the app is running. For example, the following screenshot shows the Foo app's modules in the App Manager. The name of the `*-service` module in the App Manager, `foo-service`, is also its context. Also note that the app's WSDD module is grayed out and listed as Resolved instead of Active. This is normal. WSDD modules are OSGi fragments, which can't be activated. They still work as intended, though.

Next, you'll learn how to build the WSDD module for Liferay DXP's built-in apps that don't include a WSDD by default. If you don't need to do this, you can move on to the tutorial Invoking Remote Services.


Search foo

Order by: Title

APP MANAGER > SEARCH RESULTS

Modules

foo-api
foo-api
Version: 1.0.0 Status: Active

foo-service
foo-service
Version: 1.0.0 Status: Active

foo-service.wsdd
foo-service.wsdd
Version: 1.0.0 Status: Resolved

foo-web
foo-web
Version: 1.0.0 Status: Active

Figure 85.1: To find your app's modules, including its WSDD module, search for your app in the App Manager. The `*-service` module's name in the App Manager is also the module's context.

Building the WSDD for Built-in Liferay DXP Apps

Liferay DXP doesn't provide WSDD modules for built-in apps that exist outside of the portal context. This means that by default you can't access SOAP web services for apps like Bookmarks or Blogs. To make SOAP web services available for such an app, you must build and deploy its WSDD from the `liferay-portal` GitHub repository. The apps are in the `liferay-portal/modules/apps` folder. Note that to build WSDDs for these apps, you must first download the `liferay-portal` source code to your machine. You'll run the WSDD build from your local `liferay-portal` copy.

When you build an app's WSDD, make sure to use `gradlew` in `liferay-portal` instead of the `gradle` on your machine. After building, you can find the WSDD JAR in the `tools/sdk/dist` folder of your local `liferay-portal` copy. Otherwise, building an app's WSDD is the same as in the preceding section.

For example, to build the WSDD for the Bookmarks app, first navigate to the `liferay-portal/modules/apps/collaboration/` `service` folder in your terminal. Then run the following command:

```
../../../../../gradlew buildWSDD
```

Next, deploy the `liferay-portal/tools/sdk/dist/com.liferay.bookmarks.service-wsdd-[version].jar` to your Liferay DXP instance. If your instance is running locally on `localhost:8080`, you should then be able to view the Bookmarks app's SOAP services at http://localhost:8080/o/com.liferay.bookmarks.service/api/axis.

Fantastic! Once you've created remote web services, you'll want to know how to invoke them. To learn how, see the tutorial Invoking Remote Services.

Related Topics

Invoking Remote Services
Invoking JSON Web Services
JSON Web Services Invoker
JSON Web Services Invocation Examples
What is Service Builder?

85.2 Invoking Remote Services

You can invoke the remote services of any installed Liferay application the same way that you invoke your local services. Doing so could be described as "invoking remote services locally." One reason to invoke a remote service instead of the corresponding local service might be to take advantage of the remote service's permission checks. Consider the following common scenario:

- Both a local service implementation and a remote service implementation have been created for a particular service.
- The remote service performs a permission check and then invokes the corresponding local service.

In the above scenario, it's a best practice to invoke the remote service instead of the local service. Doing so ensures that you don't need to duplicate permission checking code. This is the practice followed by the services in Liferay's Blogs app.

Of course, the main reason for creating remote services is to invoke them remotely. Service Builder can expose your project's remote web services both via a JSON API and via SOAP. By default, running Service Builder with `remote-service` set to `true` for your entities generates a JSON web services API for your project. You can access your project's JSON-based RESTful services via a convenient web interface.

Invoking Liferay Services Remotely

Many default Liferay services are available as web services. Liferay exposes its web services via SOAP and JSON web services. If you're running Liferay locally on port 8080, visit the following URL to browse Liferay's default SOAP web services:

```
http://localhost:8080/api/axis
```

To browse Liferay's default JSON web services, visit this URL:

```
http://localhost:8080/api/jsonws/
```

By default, the context path is set to / which means that core Liferay services are listed. By default, the *http://localhost:8080/api/jsonws/* page shows the JSON web services in the `portal` context. You can select a different context in the *Context Name* selector menu. For example, selecting `journal` in *Context Name* shows you the JSON web services in Liferay's Web Content app (this app's entities all begin with `Journal*`). You can also access a context's JSON web services via a direct URL. For example, the URL for the Web Content app's JSON web services is http://localhost:8080/api/jsonws?contextName=journal.

Important: To invoke Liferay services remotely, your Liferay instance must be configured to allow remote web service access. Please see the Understanding Liferay's Service Security Model tutorial for details.

Each entity's available service methods are listed in the left column of the JSON web services page. To view details about a service method, click it. The full package path to the service's `*Impl` class is displayed along with the method's parameters, return type, and possible exceptions. You can also invoke the service from this page. For example, in the `portal` context click the `AnnouncementsEntry` entity's `get-entry` method. This brings up that service method's details page, where you can also invoke the service:

The only parameter required to invoke the `get-entry` method is an `entryId`. To invoke this web service, you could enter an announcement entry's ID in the `entryId` field and then click *Invoke*. Liferay returns feedback from each invocation that indicates, for example, whether the service invocation succeeded or failed. Invoking remote services in this manner is a great way to test your app's remote services.

Service Builder can also make your project's web services available via SOAP using Apache Axis. After you've built your `*-service` project's WSDD (web service deployment descriptor) and deployed your project's modules, its services are available on your Liferay server. You can view your Liferay instance's and app's SOAP services in a browser as described in the tutorial Creating Remote Services.

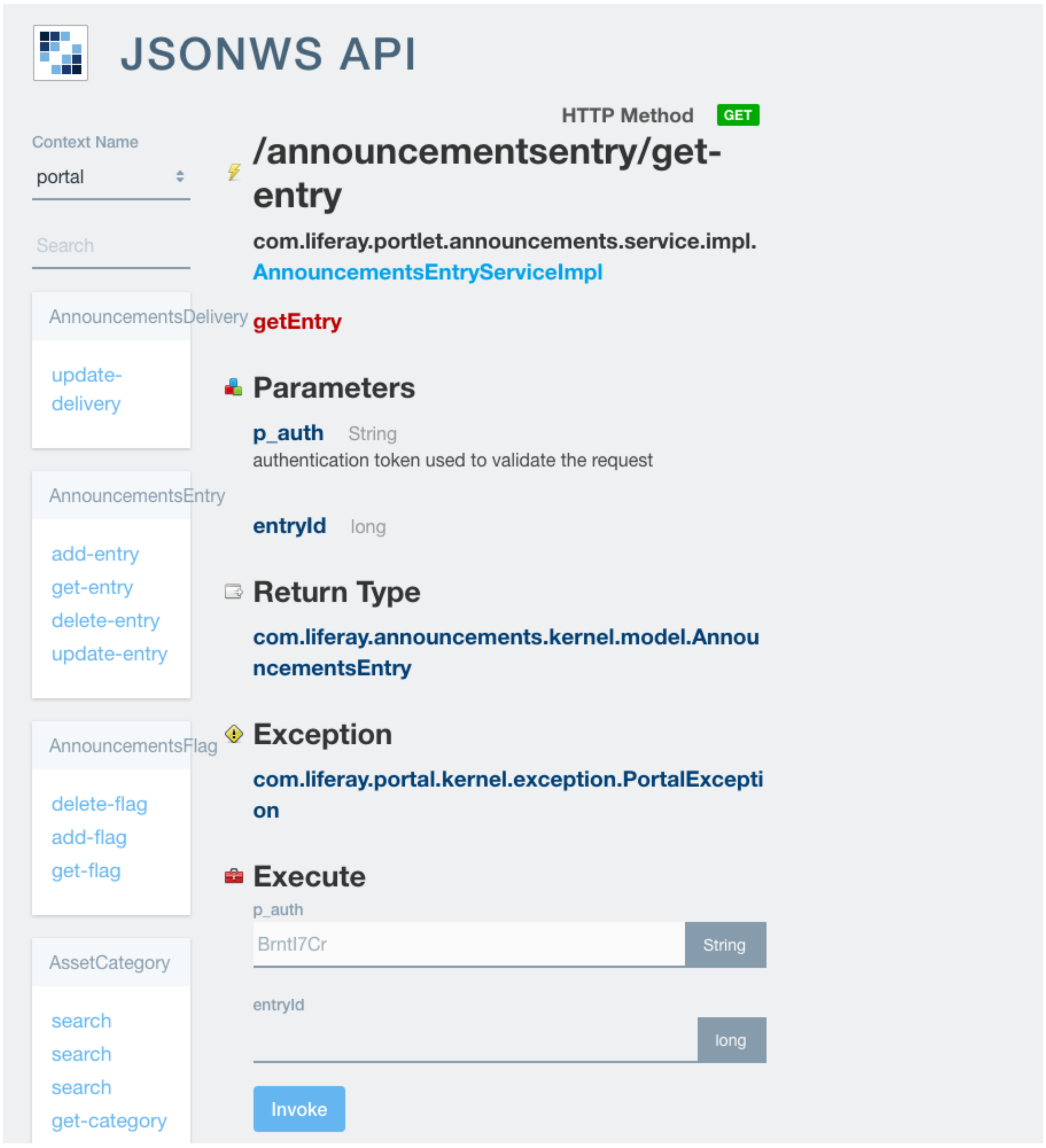
When viewing your SOAP services in a browser, Liferay lists the services available for all your entities and provides links to their WSDL documents. For example, clicking on the WSDL link for the User service takes you to the following URL:

```
http://localhost:8080/api/axis/Portal_UserService?wsdl
```

This WSDL document lists the entity's SOAP web services. Once the web service's WSDL is available, any SOAP web service client can access it. To see examples of SOAP web service client implementations, see the tutorial SOAP Web Services.

Liferay web services are designed to be invoked by client applications. Liferay's web services APIs can be accessed by many different kinds of clients, including non-portlet and even non-Java clients. For information on how to develop client applications that can access Liferay's JSON web services, please see the Invoking JSON Web Services tutorial. For information on how to develop client applications that access Liferay's SOAP web services, please see the SOAP Web Services tutorial. To learn how to create remote web services for your own application, please refer to the Creating Remote Services tutorial.

For more information on Liferay services, see the Liferay Portal CE Javadocs at @platform-ref@/7.0-latest/javadocs/.


JSONWS API

Context Name: portal

Search

AnnouncementsDelivery
- update-delivery

AnnouncementsEntry
- add-entry
- get-entry
- delete-entry
- update-entry

AnnouncementsFlag
- delete-flag
- add-flag
- get-flag

AssetCategory
- search
- search
- search
- get-category

HTTP Method GET

/announcementsentry/get-entry

com.liferay.portlet.announcements.service.impl.AnnouncementsEntryServiceImpl

getEntry

Parameters

p_auth String
authentication token used to validate the request

entryId long

Return Type

com.liferay.announcements.kernel.model.AnnouncementsEntry

Exception

com.liferay.portal.kernel.exception.PortalException

Execute

p_auth: BrntI7Cr (String)

entryId: (long)

Figure 85.2: The JSON web services page for an entity's remote service method also lets you invoke that service.

Related Topics

Invoking JSON Web Services
  JSON Web Services Invoker
  JSON Web Services Invocation Examples
  SOAP Web Services
  Creating Remote Services

85.3 Service Security Layers

Liferay's remote services are secured by default. They sit behind a layer of security that allows only local connections. To invoke Liferay services from a remote client, you must take deliberate steps to enable such access. Liferay's core web services require user authentication and verification. Regardless of whether you call the remote service from the same machine or via a web service, Liferay's standard security model performs its function. The user invoking a web service must have the proper permissions (as defined by Liferay's permissions system) for the remote service invocation to complete successfully. This tutorial explains these processes.

The first layer of security that a client encounters when calling a remote service is called *invoker IP filtering*. Imagine that you have have a batch job that runs on another machine in your network. This job polls a shared folder on your network and uses Liferay's web services to upload documents to your Liferay site's *Documents and Media* app on a regular basis. To get your batch job through the IP filter, you must grant web service access to the machine the batch job is running on. For example, if your batch job uses Liferay's SOAP web services to upload the documents, you must add the IP address of the machine where the batch job runs to the `axis.servlet.hosts.allowed` property. A typical entry might look like this:

```
axis.servlet.hosts.allowed=192.168.100.100, 127.0.0.1, [SERVER_IP]
```

If the IP address of the machine where the batch job runs is listed as an authorized host for the service, the machine is allowed to connect to Liferay's web services, pass in the appropriate user credentials, and upload the documents.

Note: The `portal.properties` file resides on the Liferay instance's host machine and is controlled by the instance administrator. Instance administrators can configure security settings for the Axis Servlet, the Liferay Tunnel Servlet, the Spring Remoting Servlet, the JSON Servlet, the JSON Web Service Servlet, and the WebDAV Servlet. The `portal.properties` file (online version is available at @platform-ref@/7.0-latest/propertiesdoc/portal.properties.html) describes these properties.

Next, if you invoke the remote service via web services (e.g., JSON WS, old JSON, Axis, REST, etc.), a two step process of authentication and authentication verification takes place. Each call to a Liferay web service must be accompanied by a user authentication token: `p_auth`. It's up to the web service caller to produce the token (e.g., through Liferay's utilities or through some third-party software). Liferay verifies that there is a Liferay user matching the token. If the credentials are invalid, the web service invocation is aborted. Otherwise, processing enters Liferay's user permission layer.

Liferay's user permission layer is the last Liferay security layer triggered when services are invoked remotely. It's used for every object in the Liferay instance, regardless of whether a local or remote service is involved. The user ID associated with a web service invocation must possess the proper permission to operate on the objects it's trying to access. A remote exception is thrown if the user ID doesn't possess the required permissions. An instance administrator can grant users access to these resources. For example, suppose you created a Documents and Media Library folder called *Documents* in a site, created a role called *Document Uploaders*, and granted this role the rights to add documents to your new folder. If your batch job accesses Liferay's web services to upload documents into the folder, you must call the web service using a user ID of a member of this role (or using the user ID of a user with individual rights to add documents to this folder, such as an instance administrator). If you don't, Liferay denies you access to the web service.

When invoking remote Liferay services from a non-browser client, you can specify the user credentials using HTTP basic authentication. For security reasons, you must be logged in and supply a valid `p_auth` authentication token to invoke a Liferay web service via a browser. Since you should never pass credentials over the network unencrypted, we recommend using HTTPS whenever accessing Liferay services on an untrusted network. Most HTTP clients (e.g., cURL) let you specify the basic authentication credentials in the URL–this is very handy for testing.

Important: To invoke a Liferay web service via your browser, you must be logged in to Liferay. You must also supply an authentication token (the `p_auth` parameter). If you navigate to your Liferay instance's JSON web services API page (localhost:8080/api/jsonws, by default) and click on a remote service method, you'll see the `p_auth` token for your browser session. This token is automatically supplied when you invoke a Liferay web service via the JSON web services API page or via JavaScript using `Liferay.Service(...)`.

Use the following syntax to call the Axis web service using credentials.

```
http://" + emailAddressOrScreenNameOrUserIdAsString + ":" + password + "@[server.com]:\
[port]/api/axis/" + serviceName
```

The `emailAddressOrScreenNameOrUserIdAsString` should be the user's email address, screen name, or user ID. The Liferay instance's authentication type setting determines which one to use. A user can find his or her ID by logging in as the user and accessing *My Account → Account Settings* from the User Menu. On this interface, the user ID appears below the user's profile picture and above the birthday field.

Suppose that your Liferay instance is set to authenticate by user ID, and that there's a user with an ID of `2` and a password of `test`. You can access Liferay's remote Organization service with the following URL:

```
http://2:test@localhost:8080/api/axis/Portal_OrganizationService
```

Note that if an email address appears in the URL path, it must be URL-encoded (e.g. `test@example.com` becomes `test%40liferay.com`).

Suppose that your Liferay instance is now set to authenticate by email address. To call the same web service for the same user, change the URL to this:

```
http://test%40liferay.com:test@localhost:8080/api/axis/Portal_OrganizationService
```

As mentioned, the authentication type specified for your Liferay instance dictates the authentication type you'll use to access your web service. The instance administrator can set the instance's authentication type to email address, screen name, or user ID.

You can set the authentication type via the Control Panel or via the `portal-ext.properties` file. To set the authentication type via the Control Panel, navigate to *Control Panel → Configuration → Instance Settings*, and select the *General* tab under *Authentication*. Choose your authentication type in the *How do users authenticate?* menu. To set the authentication type via properties file, add the following lines to your Liferay instance's `portal-ext.properties` file and uncomment the line for the appropriate authentication type:

```
#company.security.auth.type=emailAddress
#company.security.auth.type=screenName
#company.security.auth.type=userId
```

You should also review your Liferay instance's password policies, since they'll be enforced on your administrative user as well. If the instance enforces password policies on its users (e.g., requires them to change their passwords on a periodic basis), an administrative user accessing Liferay's web services in a batch job will have his or her password expire too.

To prevent a password from expiring, an instance administrator can add a new password policy that doesn't enforce password expiration, and then add a specific administrative user to the policy. Then your batch job can run as many times as you need it to, without your administrative user's password expiring.

To summarize, accessing Liferay remotely requires you to pass the following layers of security checks:

- *IP permission layer*: The IP address must be pre-configured in the server's portal properties.
- *Authentication/verification layer (web services only)*: Liferay verifies that the caller's authorization token can be associated with an instance user.

- *User permission layer*: The user needs permission to access the related resources.

If you'd like to develop client applications that can invoke Liferay's web services, make sure that your Liferay instance's web service security settings have been configured to allow access.

Related Topics

Configuring JSON Web Services

Invoking Remote Services

Invoking JSON Web Services

JSON Web Services Invoker

JSON Web Services Invocation Examples

SOAP Web Services

85.4 Registering JSON Web Services

Liferay's developers use a tool called *Service Builder* to build services. When you build services with Service Builder, all remote-enabled services (i.e., `service.xml` entities with the property `remote-service="true"`) are exposed as JSON web services. When each `*Service.java` interface is created for a remote-enabled service, the `@JSONWebService` annotation is added to that interface at the class level. All of the public methods of that interface become registered and available as JSON web services.

The `*Service.java` interface source file should never be modified by the user. If you need more control over its methods (e.g., if you need to hide some methods while exposing others), you can configure the `*ServiceImpl` class. When the service implementation class (`*ServiceImpl`) is annotated with the `@JSONWebService` annotation, the service interface is ignored and the service implementation class is used for configuration in its place. In other words, `@JSONWebService` annotations in the service implementation override any JSON web service configuration in the service interface.

That's it! Liferay scans all OSGi bundles registered with the `@Component` annotation or in a `*BundleActivator` class for remote services. Each class that uses the `@JSONWebService` annotation is examined and its methods become exposed via the JSON web services API. As explained previously, the `*ServiceImpl` configuration overrides the `*Service` interface configuration during registration.

Note: Liferay's developers use *Service Builder* to expose their services via JSON automatically. If you haven't used Service Builder before, please see the Service Builder section of tutorials.

Next, you'll see how you can register your application's remote services as JSON web services. Keep in mind that Liferay uses this same mechanism. This is why Liferay's remote services are exposed as JSON web services out-of-the-box.

Registering an App's JSON Web Services

As an example, say you have an app named `SupraSurf` that has some services, and you decide to expose them as remote services. After enabling the `remote-service` attribute on its `SurfBoard` entity, you rebuild the services. Service Builder regenerates the `SurfBoardService` interface, adding the `@JSONWebService` annotation to it. This annotation tells Liferay that the interface's public methods are to be exposed as JSON web services, making them a part of the app's JSON API. Start up your Liferay instance if it isn't running, and then deploy your app to Liferay.

To get some feedback from your Liferay instance on registering your application's services, configure the instance to log the application's informational messages (i.e., its `INFO ...` messages). See the tutorials on Liferay's logging system for details.

To test Liferay's JSON web service registration process, add a simple method to your app's services. Edit your `*ServiceImpl` class and add the following method:

```
public String helloWorld(String worldName) {
    return "Hello world: " + worldName;
}
```

Rebuild the services and re-deploy your app's modules. You can now invoke this service method via JSON. For instructions on doing this, see the JSON invocation tutorials listed in this section of tutorials.

This same mechanism registers Liferay's own services. They're conveniently enabled by default, so you don't have to configure them.

Next, you'll learn how to form a mapped URL for the remote service so you can access it.

Mapping and Naming Conventions

You can form the mapped URL of an exposed service by following the naming convention below:

```
http://[server]:[port]/api/jsonws/[context-path].[service-class-name]/[service-method-name]
```

Look at the last three bracketed items more closely:

- `context-name` is the app's context name (e.g., `suprasurf` in the previous example). Its value is specified via the `json.web.service.context.path` property in the `@OSGiBeanProperties` annotation. For example, for Liferay web content articles, Liferay's `JournalArticleService` class includes the following annotation (among others):

  ```
  @OSGiBeanProperties(property =  {
      "json.web.service.context.name=journal", "json.web.service.context.path=JournalArticle"}, service = JournalArticleService.class)
  ```

- `service-class-name` is generated from the service's class name in lower case, minus its `Service` or `ServiceImpl` suffix. For example, specify `surfboard` as the `app-context-name` for the `SurfBoardService` class.

- `service-method-name` is generated from the service's method name by converting its camel case to lower case and using dashes (-) to separate words.

The following example demonstrates these naming conventions by mapping a service method's URL using the naming conventions both on a custom service and on a Liferay service.

For the custom service method, the URL looks like:

```
http://localhost:8080/api/jsonws/suprasurf.surfboard/hello-world
```

Note the context name part of the URL. For Liferay, it's similar. Here's a Liferay service method:

```
@JSONWebService
public interface UserService {
    public com.liferay.portal.model.User getUserById(long userId) {...}
```

Here's that Liferay service method's URL:

```
http://localhost:8080/api/jsonws/user/get-user-by-id
```

Each service method is bound to one HTTP method type. Any method with a name starting with `get`, `is`, or has is assumed to be a read-only method and is mapped as a *GET HTTP* method by default. All other methods are mapped as *POST HTTP* methods.

Recall that you can see a list of your Liferay instance's JSON web services at http://localhost:8080/api/jsonws. When you select a method on this page, the part of its HTTP method URL that follows `http://[server]:[port]/api/jsonws` is listed at the top of the screen.

Conveniently, remote service requests can leverage the authentication credentials associated with the user's current Liferay session. Next, you'll learn how to prevent a method from being exposed as a service.

Ignoring a Method

To keep a method from being exposed as a service, annotate the method with the following option:

```
@JSONWebService(mode = JSONWebServiceMode.IGNORE)
```

Methods with this annotation don't become part of the JSON Web Service API. Next, you'll learn how to define custom HTTP method and URL names.

HTTP Method and URL Names

At the method level, you can define custom HTTP method names and URL names. Just use an annotation like this one:

```
@JSONWebService(value = "add-board-wow", method = "PUT")
public boolean addBoard(
```

In this example, the application's service method `addBoard` is mapped to URL method name `add-board-wow`. Its complete URL is now `http://localhost:8080/api/jsonws/suprasurf.surfboard/add-board-wow` and can be accessed using the HTTP PUT method.

If the URL method name in a JSON web service annotation starts with a slash character (`/`), only the method name is used to form the service URL; the class name is ignored:

```
@JSONWebService("/add-something-very-specific")
public boolean addBoard(
```

Similarly, you can change the class name part of the URL, by setting the value in a class-level annotation:

```
@JSONWebService("sbs")
public class SurfBoardServiceImpl extends SurfBoardServiceBaseImpl {
```

This maps all of the service's methods to a URL class name `sbs` instead of the default class name `surfboard`.

Next, you'll learn a different approach to exposing your methods via manual registration.

Manual Registration Mode

Up to now, it's assumed that you want to expose most of your service methods, while hiding some specific methods (the *blacklist* approach). Sometimes, however, you want the opposite: to explicitly specify only the methods you want to expose (the *whitelist* approach). This is possible by specifying *manual mode* on the class-level annotation. Then it's up to you to annotate only those methods you want to expose. For example:

```
@JSONWebService(mode = JSONWebServiceMode.MANUAL)
public class SurfBoardServiceImpl extends SurfBoardServiceBaseImpl{
    ...
    @JSONWebService
    public boolean addBoard(
```

Now only the `addBoard` method and any other method annotated with `@JSONWebService` are part of the JSON Web Service API; all of this service's other methods are excluded from the API.

Related Topics

Invoking JSON Web Services
JSON Web Services Invoker
JSON Web Services Invocation Examples

85.5 Invoking JSON Web Services

If you know the URL and are connected to the internet, invoke Liferay's JSON web service API in any language you want or directly with the URL or cURL. Additionally, Liferay provides a handy JSON web services page that allows you to browse and invoke service methods.

If you're running Liferay locally on port 8080, you can find the JSON web services page at http://localhost:8080/api/jsonws. You can use this page to generate example code for invoking web services. When you invoke a service on this page as described in the tutorial Invoking Remote Services, the JSON result of your service invocation appears. Click on the *JavaScript Example*, *curl Example*, or *URL Example* tabs to see different ways of invoking the web service.

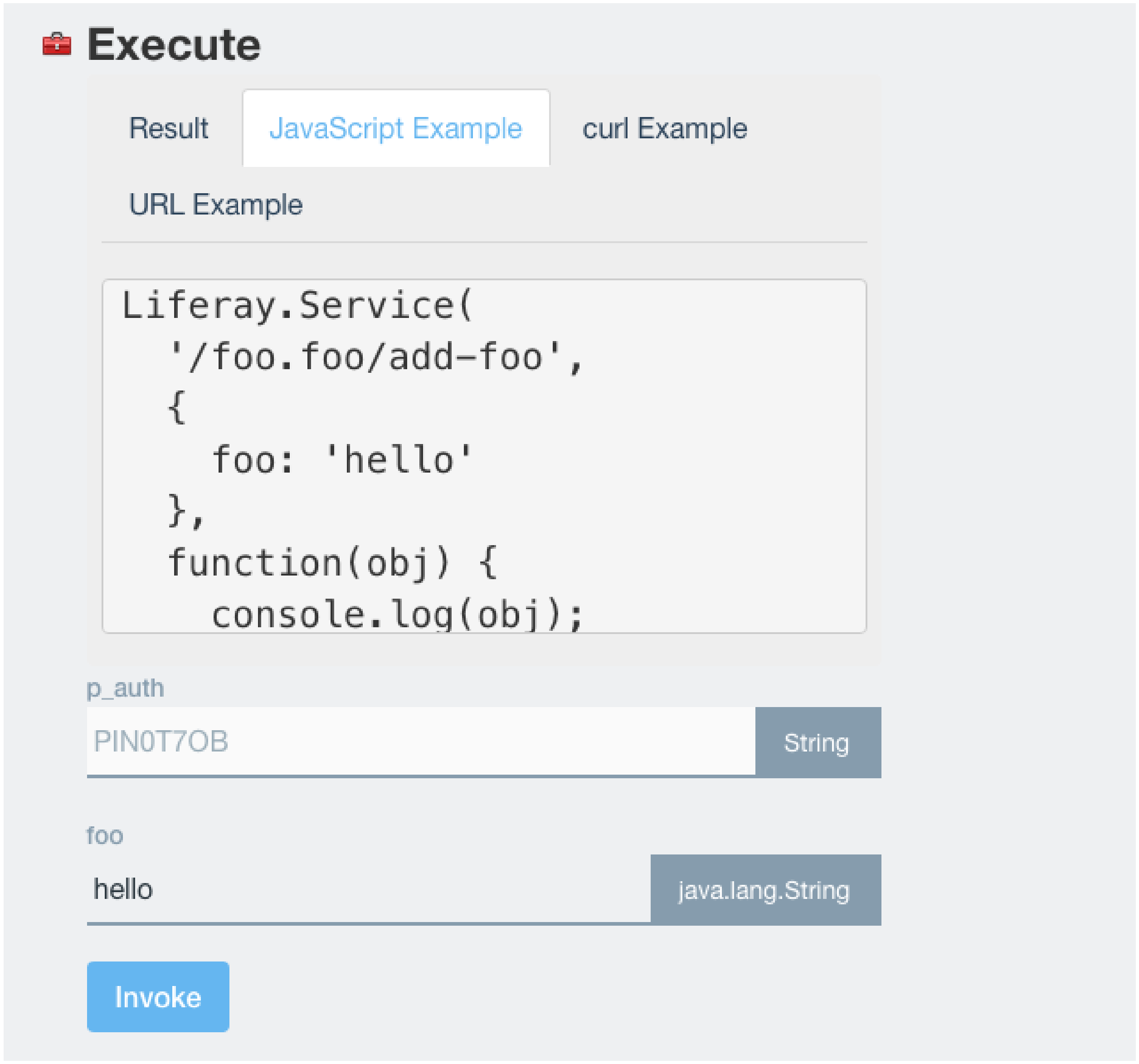
This tutorial explains general techniques for working with JSON web services and includes details about invoking them via URL. For examples of invoking Liferay's JSON web services via JavaScript, URL, and cURL, see the JSON Web Services Invocation Examples tutorial.

There are multiple ways to invoke a JSON web service since there are different ways to supply parameters. In this tutorial, you'll learn how to include parameters in web service invocations. First, you must understand how your invocation is matched to a method, especially in the case of overloaded service methods.

The general rule is that you provide the service method's name and *all* the service method's parameters–even if you only provide `null` values. It's important to provide all parameters, but it doesn't matter *how* you do it (e.g., as part of the URL line, as request parameters, etc.). The order of the parameters doesn't matter either.

Note: An authentication related token (p_auth) must accompany each Liferay web service invocation. For details, see the Service Security Layers tutorial. Also, see the note in the following section to learn how to find the `p_auth` token value that corresponds to your Liferay session.

Exceptions abound in life, and there's an exception to the rule that *all* parameters are required. When using numeric *hints* to match methods, not all of the parameters are required. You'll learn to use hints next.



```
Liferay.Service(
  '/foo.foo/add-foo',
  {
    foo: 'hello'
  },
  function(obj) {
    console.log(obj);
```

p_auth: PIN0T7OB (String)

foo: hello (java.lang.String)

Figure 85.3: When you invoke a service from Liferay's JSON web services page, you can view the result of your service invocation as well as example code for invoking the service via JavaScript, curl, or URL.

Using Hints When Invoking a Service via URL

Adding numeric hints lets you specify how many method arguments a service has. If you don't specify an argument for a parameter, it's automatically passed in as `null`. Syntactically, you can add hints as numbers separated by a dot in the method name. Here's an example:

```
/foo/get-bar.2/param1/123/-param2
```

Here, the `.2` is a numeric hint specifying that only service methods with two arguments are matched; others will be ignored for matching.

There's an important distinction to make between matching with hints and matching without hints. When a hint is specified, you don't have to specify all of the parameters. Any missing arguments are treated as `null`. The previous example may be called like this:

```
/foo/get-bar.2/param1/123
```

In this example, `param2` will automatically be set to `null`.

Here's a real Liferay example:

```
http://localhost:8080/api/jsonws/bookmarks.bookmarksfolder/add-folder.4/parent-folder-id/0/name/News?p_auth=[value]
```

In this example, the hint number is `4` because there are four parameters: `parentFolderId`, `name`, `description`, and `p_auth`. Since the `description` parameter is omitted, its value is assumed to be `null`. If you try to invoke this web service with another hint number such as `3` or `5`, you'll get an exception since there is no `bookmarks/add-folder` method that takes that number of parameters. The authentication parameter `p_auth` is associated with your Liferay session. See below for more information.

Important: When invoking a Liferay web service by entering a URL into your browser, you must be logged into Liferay with an account that has permission to invoke the web service. You must also supply an authentication token as a URL parameter. This authentication token is associated with your browser session and is called `p_auth`. Using this authentication token helps prevent CSRF attacks.

Here are two easy ways to find the `p_auth` token:

1. Go to Liferay's JSON web services page and click on any service method. The value of the `p_auth` token appears under the Execute heading.

2. If you're working from a JavaScript context and have access to the `Liferay` object, invoking `Liferay.authToken` provides the value of the `p_auth` parameter.

For example, if your `p_auth` parameter's value is `n35K1pb2`, you could invoke the preceding URL examples like this:

```
http://localhost:8080/api/jsonws/bookmarks.bookmarksfolder/add-folder.4/parent-folder-id/0/name/News?p_auth=n35K1pb2
```

For simplicity, the remainder of this tutorial omits the `p_auth` parameter from the example URLs for invoking web services. Remember that you must include it if you want to invoke services from your browser!

Next, you'll learn how to pass parameters as part of the URL path.

Passing Parameters as Part of a URL Path

To pass method parameters as part of the URL path, specify them in name-value pairs after the service URL. Parameter names must be formed from method argument names by converting them from camel case to names that use all lower case, dash-separated words. For example, this returns all top-level bookmark folders from the specified site:

```
http://localhost:8080/api/jsonws/bookmarks.bookmarksfolder/get-folders/group-id/20181/parent-folder-id/0
```

You can pass parameters in any order; it's not necessary to follow the order in which the arguments are specified in the method signatures.

When a method name is overloaded, the *best match* will be used. The method that contains the least number of undefined arguments is chosen and invoked for you.

You can also pass parameters in a URL query. The next section shows you how to do this.

Passing Parameters as a URL Query

To pass in parameters as request parameters, specify them as-is (camel case) and set them equal to their argument value. For example:

```
http://localhost:8080/api/jsonws/bookmarks.bookmarksfolder/add-folder?parentFolderId=0&name=News&description=news
```

As with passing parameters as part of a URL path, the parameter order is not important and the best match rule applies for overloaded methods.

Now you know a few different ways to pass parameters. It's also possible to pass URL parameters in a mixed way. For example, some can be part of the URL path while others can be specified as request parameters.

Parameter values are sent as strings using the HTTP protocol. Before a matching Java service method is invoked, each parameter value is converted from a `String` to its target Java type. Liferay uses a third party open source library to convert each object to its appropriate common type. Although it's possible to add or change the conversion for certain types, this tutorial only covers the standard conversion process.

Conversion for common types (e.g., `long`, `String`, `boolean`) is straightforward. Dates can be given in milliseconds. Locales can be passed as locale names (e.g. `en` and `en_US`). To pass in an array of numbers, send a string of comma-separated numbers (e.g. the string `4,8,15,16,23,42` can be converted to `long[]` type). You get the picture!

In addition to the common types, arguments can be of type `List` or `Map`. To pass a `List` argument, send a JSON array. To pass a `Map` argument, send a JSON object. These types of conversions are performed in two steps:

- *Step 1–JSON deserialization*: JSON arrays are converted into `List<String>`, and JSON objects are converted to `Map<String, String>`. For security reasons, it's forbidden to instantiate any type within JSON deserialization.
- *Step 2–Generification*: Each `String` element of the `List` and `Map` is converted to its target type (the argument's generic Java type specified in the method signature). This step is only executed if the Java argument type uses generics.

As an example, consider the conversion of a `String` array `[en,fr]` as JSON web service parameters for a `List<Locale>` Java method argument type:

- *Step 1–JSON deserialization*: The JSON array is deserialized to a `List<String>` containing `String`s `en` and `fr`.

- *Step 2–Generification*: Each `String` is converted to the `Locale` (the generic type), resulting in the `List<Locale>` Java argument type.

Next, you'll learn how to specify an argument as `null`.

Sending Null Values

To pass a `null` value for an argument, prefix the parameter name with a dash. Here's an example:

```
http://localhost:8080/api/jsonws/bookmarks.bookmarksfolder/add-folder/parent-folder-id/0/name/News/-description
```

Here's the equivalent example using URL query parameters instead of URL path parameters:

```
http://localhost:8080/api/jsonws/bookmarks.bookmarksfolder/add-folder?parentFolderId=0&name=News&-description
```

The `description` parameter is interpreted as `null`. Note that this parameter doesn't have to be last in the URL.

Null parameters don't have specified values. When a null parameter is passed as a request parameter, its value is ignored and `null` is used instead:

```
<input type="hidden" name="-description" value=""/>
```

When using JSON-RPC (see the JSON-RPC section below), you can send null values explicitly, even without a prefix. Here's an example:

```
"description":null
```

Next, you'll learn about encoding parameters.

Encoding Parameters

There's a difference between URL encoding and query (i.e., request parameters) encoding. The difference lies in how the space character is encoded. When the space character is part of the URL path, it's encoded as `%20`; when it's part of the query it's encoded as a plus sign (`+`).

All these encoding rules apply to ASCII and international (non-ASCII) characters. Since Liferay works in UTF-8 mode, parameter values must be encoded as UTF-8 values. Liferay doesn't decode request URLs and request parameter values to UTF-8 itself; it relies on the web server layer. When accessing services through JSON-RPC, encoding parameters to UTF-8 isn't enough–you need to send the encoding type in a Content-Type header (e.g. `Content-Type : "text/plain; charset=utf-8"`).

For example, suppose you want to pass the value "Супер" ("Super" in Cyrillic) to a JSON web service method. This name first has to be converted to UTF-8 (resulting in an array of 10 bytes) and then encoded for URLs or request parameters. The resulting value is the string `%D0%A1%D1%83%D0%BF%D0%B5%D1%80` that can be passed to your service method. When received, this value is first translated to an array of 10 bytes (URL decoded), and then converted to a UTF-8 string of the 5 original characters.

Next, you'll learn how to send files as arguments.

Sending Files as Arguments

Files can be uploaded using multi-part forms and requests. Here's an example:

```
<form
 action="http://localhost:8080/api/jsonws/dlapp/add-file-entry"
 method="POST"
 enctype="multipart/form-data">
    <input type="hidden" name="repositoryId" value="10172"/>
    <input type="hidden" name="folderId" value="0"/>
    <input type="hidden" name="title" value="test.jpg"/>
    <input type="hidden" name="description" value="File upload example"/>
    <input type="hidden" name="changeLog" value="v1"/>
    <input type="file" name="file"/>
    <input type="submit" value="addFileEntry(file)"/>
</form>
```

This is a common upload form that invokes the `DLAppService` class's `addFileEntry` method.
Now you'll learn how to invoke JSON web services using JSON-RPC.

JSON-RPC

You can invoke JSON Web Service using JSON-RPC. Most of the JSON-RPC 2.0 specification is supported in Liferay JSON web services. One important limitation is that parameters must be passed in as *named* parameters. Positional parameters aren't supported, as there are too many overloaded methods for convenient use of positional parameters.

Here's an example of invoking a JSON web service using JSON-RPC:

```
POST http://localhost:8080/api/jsonws/dlapp
{
    "method":"get-folders",
    "params":{"repositoryId":10172, "parentFolderId":0},
    "id":123,
    "jsonrpc":"2.0"
}
```

Next, you'll learn about parameters that are made available to secure JSON web services by default.

Default Parameters

When accessing secure JSON web services (i.e., services for which the user must be authenticated), some parameters are made available to the web services by default. All of Liferay's web services are secured by default. Unless you want to change the available parameters' values to something other than their defaults, you don't have to specify them explicitly.

Here are the available default parameters:

- `userId`: The primary key of the authenticated user
- `user`: The full user object
- `companyId`: The primary key of the user's company
- `serviceContext`: The empty service context object

Next, you'll learn about object parameters.

Object Parameters

Most services accept simple parameters like numbers and strings. However, sometimes you might need to provide an object (a non-simple type) as a service parameter.

To create an instance of an object parameter, prefix the parameter with a plus sign, + and don't assign it any other parameter value. This is similar to specifying a null parameter by prefixing the parameter with a dash symbol, -.

Here's an example:

```
/jsonws/foo/get-bar/zap-id/10172/start/0/end/1/+foo
```

To create an instance of an object parameter as a request parameter, make sure you encode the + symbol:

```
/jsonws/foo/get-bar?zapId=10172&start=0&end=1&%2Bfoo
```

Here's an alternative syntax:

```
<input type="hidden" name="+foo" value=""/>
```

If a parameter is an abstract class or an interface, it can't be instantiated as such. Instead, a concrete implementation class must be specified to create the argument value. You can do this by specifying the + prefix before the parameter name, followed by specifying the concrete implementation class. Here's an example:

```
/jsonws/foo/get-bar/zap-id/10172/start/0/end/1/+foo:com.liferay.impl.FooBean
```

Here's another way of doing it:

```
<input type="hidden" name="+foo:com.liferay.impl.FooBean" value=""/>
```

The examples above specify that a `com.liferay.impl.FooBean` object, presumed to implement the class of the parameter named `foo`, is created.

You can also set a concrete implementation as a value. Here's an example:

```
<input type="hidden" name="+foo" value="com.liferay.impl.FooBean"/>
```

In JSON-RPC, here's what it looks like:

```
"+foo" : "com.liferay.impl.FooBean"
```

All the preceding examples specify a concrete implementation for the `foo` service method parameter.

Once you pass in an object parameter, you might want to populate the object. Find out how next.

Inner Parameters

When you pass in an object parameter, you'll often need to populate its inner parameters (i.e., fields). Consider a default parameter `serviceContext` of type `ServiceContext`. To make an appropriate call to JSONWS, you might need to set the `serviceContext` parameter's `addGroupPermissions` and `scopeGroupId` fields.

You can pass inner parameters by using dot notation to specify them. Just append the name of the parameter with a dot (i.e., a period, .), followed by the inner parameter's name. For the `ServiceContext` inner parameters mentioned previously, you'll specify `serviceContext.addGroupPermissions` and `serviceContext.scopeGroupId`. These are recognized as inner parameters and their values are injected into existing parameters before the API service method is executed.

Inner parameters aren't counted as regular parameters for matching methods and are ignored during matching.

Tip: Use inner parameters with object parameters to set inner contents of created object parameter instances!

Next, let's see what values are returned when a JSON web service is invoked.

Returned Values

No matter how a JSON web service is invoked, it returns a JSON string that represents the service method result. Returned objects are *loosely* serialized to a JSON string and returned to the caller.

Now you'll look at some values returned from service calls. You'll create a `UserGroup` as in the SOAP web service client examples. To make it easy, you'll use the test form provided with the JSON web service in our browser.

1. Sign in to a local Liferay instance as an administrator and then point your browser to the JSON web service method that adds a `BookmarksFolder`:

   ```
   http://localhost:8080/api/jsonws?contextName=bookmarks&signature=%2Fbookmarks.bookmarksfolder%2Fadd-folder-4-parentFolderId-
   name-description-serviceContext
   ```

   Alternatively, navigate to it by starting at `http://localhost:8080/api/jsonws` and then scrolling down to the section for *BookmarksFolder*. Then click *add-folder*.

2. In the `parentFolderId` field, enter `0`. Top-level bookmarks folders have a `parentFolderId` value of `0`. Set the name to an arbitrary value like *News*. Set the description to something like *Created via JSON WS*.

3. Click *Invoke* and you'll get a result similar to the following:

   ```
   {
     "companyId": "20202",
     "createDate": 1459969296960,
     "description": "Created via JSON WS",
     "folderId": "31001",
     "groupId": "20233",
     "lastPublishDate": null,
     "modifiedDate": 1459969297005,
     "name": "News",
     "parentFolderId": "0",
     "resourceBlockId": "1",
     "status": 0,
     "statusByUserId": "0",
     "statusByUserName": "",
   ```

```
  "statusDate": null,
  "treePath": "/31001/",
  "userId": "20250",
  "userName": "Joe Bloggs",
  "uuid": "0682170c-f9d7-f295-aa67-26ceea37a6e5"
}
```

The returned `String` represents the `BookmarksFolder` object you just created, serialized into a JSON string. To find out more about JSON strings, go to json.org.

Common JSON Web Service Errors

While working with JSON web services, you may encounter errors. Some common errors are listed here:

- *Authenticated access required*

  If you see this error, it means you don't have permission to invoke the remote service. Double-check that you're signed in as a user with the appropriate permissions. If necessary, sign in as an administrator to invoke the remote service.

- *Missing value for parameter*

  If you see this error, you didn't pass a parameter value along with the parameter name in your URL path. The parameter value must follow the parameter name, like in this example:

  ```
  /api/jsonws/user/get-user-by-id/userId
  ```

  The path above specifies a parameter named `userId`, but doesn't specify the parameter's value. You can resolve this error by providing the parameter value after the parameter name:

  ```
  /api/jsonws/user/get-user-by-id/userId/173
  ```

- *No JSON web service action associated*

  This is error means no service method could be matched with the provided data (method name and argument names). This can be due to various reasons. For example, arguments may be misspelled, the method name may be formatted incorrectly, and so on. Since JSON web services reflect the underlying Java API, any changes in the respective Java API are automatically propagated to the JSON web services. For example, if a new argument is added to a method or an existing argument is removed from a method, the parameter data must match that of the new method signature.

- *Unmatched argument type*

  This error appears when you try to instantiate a method argument using an incompatible argument type.

Related Topics

JSON Web Services Invoker
JSON Web Services Invocation Examples
Service Security Layers
Invoking Remote Services

85.6 JSON Web Services Invoker

To use JSON web services, you send a request that defines a service method and parameters, and you receive the result as a JSON object. As straightforward as this seems, it can be improved. In this tutorial, you'll learn how to use JSON web services more efficiently and pragmatically.

Consider the following example. You're working with two related objects: a `User` and its corresponding `Contact`. With simple JSON web service calls, you first call the user service to get the user object, and then you use that object's contact ID to call the contact service. You end up sending two HTTP requests to get two JSON objects that aren't even bound together. There's no contact information in the user object (i.e. no `user.contact`). This approach is suboptimal with respect to performance (sending two HTTP calls) and usability (manually managing the relationship between two objects). It'd be nicer if you had a tool to address these inefficiencies. Fortunately, the JSON Web Service Invoker does just that!

Liferay's JSON Web Service Invoker helps optimize your JSON Web Services use. In the following sections, you'll learn how.

Simple Invoker Calls

The Invoker is accessible from the following fixed address:

```
http://[address]:[port]/api/jsonws/invoke
```

It only accepts a `cmd` request parameter–this is the Invoker's command. If the command request parameter is missing, the request body is used as the command. So you can specify the command by either using the request parameter `cmd` or the request body.

The Invoker command is a plain JSON map that describes how JSON web services are called and how the results are managed. Here's an example of how to call a simple service using the Invoker:

```
{
    "/user/get-user-by-id": {
        "userId": 123,
        "param1": null
    }
}
```

The service call is defined as a JSON map. The key specifies the service URL (i.e. the service method to be invoked) and the key's value specifies a map of service parameter names (i.e. `userId` and `param1`) and their values. In the example above, the retrieved user is returned as a JSON object. Since the command is a JSON string, null values can be specified either by explicitly using the `null` keyword or by placing a dash before the parameter name and leaving the value empty (e.g. `"-param1": ''`).

The example Invoker calls functions exactly the same way as the following standard JSON Web Service call:

```
/user/get-user-by-id?userId=123&-param1
```

Next, suppose that you're running Liferay locally on port `8080`. Consider the following example of a real Liferay JSON web service invoker call. Suppose that you're signed in to Liferay as the default admin user whose email address is `test@example.com` and whose user ID is `20127`. And suppose that the value of your `p_auth` authentication token is `htXjvt5d`. You can then invoke the following URL to obtain a JSON representation of your user object:

```
http://localhost:8080/api/jsonws/invoke?cmd={%22/user/get-user-by-id%22:{%22userId%22:20172}}&p_auth=htXjvt5d
```

This URL uses the following JSON map. Note that it's supplied in the URL by using the cmd URL parameter:

```
{
    "/user/get-user-by-id": {
        "userId": 20172
    }
}
```

Note in the URL that the double quotes are URL-encoded. Also, if you're not sure what your user ID is, you can find it in the User Menu under *My Account → Account Settings*. If you're not sure what the value of your p_auth authentication token is, navigate to Liferay's JSON web services API page and click on any method in the list. The value of your p_auth token appears under the Execute heading along with any other parameters of the selected API method.

You can use JSON syntax for supplying values for objects and arrays that you need to supply as parameters. To supply a value for an object, use curly brackets: { and }. To supply a value for an array, use square brackets: [and]. Suppose as before that you're signed in to Liferay as an admin user and that the value of your p_auth authentication token is htXjvt5d. Furthermore, suppose that two vocabularies have been created with vocabulary IDs of 20783 and 20784. Here's a Liferay JSON web service invoker example that demonstrates how to pass an array as a parameter:

```
http://localhost:8080/api/jsonws/invoke?cmd={%22/assetvocabulary/get-vocabularies%22:{%22vocabularyIds%22:[20783,20784]}}&p_auth=htXjvt5d
```

This URL uses the following JSON map:

```
{
    "/assetvocabulary/get-vocabularies": {
        "vocabularyIds": [20783,20784]
    }
}
```

As before, the double quotes in the URL are URL-encoded. Also, the vocabularyIds parameter is an array, so its value is supplied as a JSON array.

Finally, here's one more Liferay JSON web service invoker example that demonstrates how to pass an object containing an array as a parameter:

```
http://localhost:8080/api/jsonws/invoke?cmd={%22/user/add-user%22:{%22companyId%22:20127,%22autoPassword%22:false,%22password1%22:%22test%22,%22password2%22
```

This URL uses the following JSON map:

```
{
    "/user/add-user": {
        "companyId": 20127,
        "autoPassword": false,
        "password1": "test",
        "password2": "test",
        "autoScreenName": false,
        "screenName": "joe.bloggs",
        "emailAddress": "joe.bloggs@example.com",
        "facebookId": 0,
        "openId": "",
        "locale": "en_US",
        "firstName": "Joe",
        "middleName": "T",
        "lastName": "Bloggs",
        "prefixId": 0,
        "suffixId": 0,
        "male": true,
        "birthdayMonth": 1,
        "birthdayDay": 1,
        "birthdayYear": 1970,
        "jobTitle": "Tester",
        "groupIds": null,
        "organizationIds": null,
        "roleIds": null,
        "userGroupIds": null,
        "sendEmail": false,
        "serviceContext": {"assetTagNames":["test"]}
    }
}
```

The `serviceContext` is the object containing an array in this example. It contains the array `assetTagNames`.

Of course, the JSON Web Service Invoker handles calls to plugin methods as well:

```
{
    "/suprasurf/hello-world": {
        "worldName": "Mavericks"
    }
}
```

The code above calls the (fictitious) SupraSurf application's remote service.

You can use variables to reference objects returned from service calls. Variable names must start with a dollar sign, `$`. In the previous example, the service call returned a user object you can assign to a variable:

```
{
    "$user = /user/get-user-by-id": {
        "userId": 123,
    }
}
```

The `$user` variable holds the returned user object. You can reference the user's contact ID using the syntax `$user.contactId`.

Next, see how you can use nested service calls to join information from two related objects.

Nesting Service Calls

With nested service calls, you can bind information from related objects together in a JSON object. You can call other services within the same HTTP request and nest returned objects in a convenient way. Here's a nested service call in action:

```
{
    "$user = /user/get-user-by-id": {
        "userId": 123,
        "$contact = /contact/get-contact-by-id": {
            "@contactId": "$user.contactId"
        }
    }
}
```

This command defines two service calls: the contact object returned from the second service call is nested in (i.e. injected into) the user object, as a property named `contact`. Now you can bind the user and his or her contact information together!

Now you'll see what the Invoker does in the background when using a single HTTP request to make the preceding nested service call:

- First, the Invoker calls the Java service mapped to `/user/get-user-by-id`, passing in a value for the `userId` parameter.

- Next, the resulting user object is assigned to the variable `$user`.
- The nested service calls are invoked.
- The Invoker calls the Java service mapped to `/contact/get-contact-by-id` by using the `contactId` parameter, with the `$user.contactId` value from the object `$user`.
- The resulting contact object is assigned to the variable `$contact`.
- Lastly, the Invoker injects the contact object referenced by `$contact` into the user object's property named `contact`.

Note: You must flag parameters that take values from existing variables. To flag a parameter, insert the `@` prefix before the parameter name.

Next, you'll learn about filtering object properties so that only the properties you need are returned when you invoke a service.

Filtering Results

Many of Liferay's model objects are rich with properties. If you only need a handful of an object's properties for your business logic, making a web service invocation that returns all of an object's properties is a waste of network bandwidth. With the JSON Web Service Invoker, you can define a whitelist of properties: only the specific properties you request in the object are returned from your web service call. Here's how you whitelist the properties you need:

```
{
    "$user[firstName,emailAddress] = /user/get-user-by-id": {
        "userId": 123,
        "$contact = /contact/get-contact-by-id": {
            "@contactId": "$user.contactId"
        }
    }
}
```

In this example, the returned user object has only the `firstName` and `emailAddress` properties (it still has the `contact` property, too). To specify whitelist properties, you simply place the properties in square brackets (e.g., `[whiteList]`) immediately following the name of your variable.

Next, you'll learn about making calls in batch.

Making Batch Calls

When nesting service calls, the intent is to invoke multiple services with a single HTTP request. Using a single request for multiple service calls is helpful for gathering related information from the service call results, but it can also be advantageous to use a single request to invoke multiple unrelated service calls. The Invoker lets you batch service calls together to improve performance. It's simple: just pass in a JSON array of commands using the following format:

```
[
    {/* first command */},
    {/* second command */}
]
```

The result is a JSON array populated with results from each command. The commands are collectively invoked in a single HTTP request, one after another.

Great! Now you know how to use Liferay's JSON Web Service Invoker to simplify your JSON calls to Liferay.

Related Topics

Invoking Remote Services
    Invoking JSON Web Services
    JSON Web Services Invocation Examples

85.7 JSON Web Services Invocation Examples

This tutorial provides examples of invoking Liferay's JSON web services via JavaScript, URL, and cURL. To illustrate the differences between these, the same two use cases (getting a user and adding a user) are shown in each example. This tutorial also includes an example of using JavaScript to invoke Liferay's JSON web services from a portlet.

Loading AlloyUI

Liferay web pages use the AlloyUI JavaScript framework. Among the JavaScript objects created for each Liferay page is a `Liferay` object. This object includes a `Service` function that you can use to invoke Liferay's API. To invoke Liferay web services via `Liferay.Service(...)`, your JavaScript context must include the AlloyUI JavaScript framework. Liferay uses AlloyUI 3.0. If you're working in a JSP, you can load the AlloyUI taglib and wrap your JavaScript code in an `<aui:script>` tag. Here's the required import:

```
<%@ taglib uri="http://alloy.liferay.com/tld/aui" prefix="aui" %>
```

By default, the `<aui:script>` tag includes the base AUI module. To load specific AUI modules, specify them via the use attribute. For example, to use the AUI `node` and `event` modules, wrap your code like this:

```
<aui:script use="node, event">
    // Liferay service invocation here
</aui:script>
```

If you're not working in a JSP, you won't have access to taglibs. In this case, create an AUI context manually. For example, use the following HTML fragment to load the AUI seed and CSS files:

```
<script src="http://cdn.alloyui.com/3.0.0/aui/aui-min.js"></script>
<link href="http://cdn.alloyui.com/3.0.0/aui-css/css/bootstrap.min.css" rel="stylesheet"></link>
```

Then you can create an AUI context like this:

```
AUI().use('aui-base', function(A){
    // Liferay service invocation here
});
```

Now you're ready to invoke Liferay's JSON web services.

Get User JSON Web Service Invocation via JavaScript

First, examine the following JSON web service invocation, written in JavaScript:

```
Liferay.Service(
    '/user/get-user-by-email-address',
    {
        companyId: Liferay.ThemeDisplay.getCompanyId(),
        emailAddress: 'test@example.com`
    },
    function(obj) {
        console.log(obj);
    }
);
```

If you run this code, the test@example.com user (JSON object) is logged to the JavaScript console. The `Liferay.Service(...)` function takes three arguments:

1. A string representing the service to invoke
2. A parameters object
3. A callback function

The callback function takes the result of the service invocation as an argument.

Add User JSON Web Service Invocation via JavaScript

Here's an example JSON web service invocation, also written in JavaScript, that adds a new user. It requires many more parameters than the one for retrieving a user!

```
Liferay.Service(
    '/user/add-user',
    {
        companyId: Liferay.ThemeDisplay.getCompanyId(),
        autoPassword: false,
        password1: 'test',
        password2: 'test',
        autoScreenName: false,
        screenName: 'joe.bloggs',
        emailAddress: 'joe.bloggs@example.com',
        facebookId: 0,
        openId: '',
        locale: 'en_US',
        firstName: 'Joe',
        middleName: 'T',
        lastName: 'Bloggs',
        prefixId: 0,
        suffixId: 0,
        male: true,
        birthdayMonth: 1,
        birthdayDay: 1,
        birthdayYear: 1970,
        jobTitle: 'Tester',
        groupIds: null,
        organizationIds: null,
        roleIds: null,
        userGroupIds: null,
        sendEmail: false,
        serviceContext: {assetTagNames: ['test']}
    },
    function(obj) {
        console.log(obj);
    }
);
```

The `serviceContext` object assigns the `test` tag to the newly created user. Note that you can use JSON syntax to supply values for objects and arrays. For example, to supply a value for the `serviceContext` object, you use curly brackets: { and }. To supply a value for the `assetTagNames` array, you use square brackets: [and]. Thus, the line `serviceContext: {assetTagNames: ['test']}` indicates that `serviceContext` is an object containing an array named `assetTagNames`, which contains the string `test`.

Invoking JSON Web Services via JavaScript in a Application

You can adapt the example from the previous section for use in a Liferay app. For example, the JSP page below creates a form that lets the user specify a first name, middle name, last name, screen name, and email address. When the user clicks the *Add User* button, the app uses these values to create a new user.

```
<%@ taglib uri="http://alloy.liferay.com/tld/aui" prefix="aui" %>
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>

<portlet:defineObjects />

<portlet:renderURL var="successURL">
    <portlet:param name="mvcPath" value="/success.jsp"/>
</portlet:renderURL>

<portlet:renderURL var="failureURL">
    <portlet:param name="mvcPath" value="/failure.jsp"/>
</portlet:renderURL>

<aui:form method="GET" name="<portlet:namespace />fm">
    <aui:fieldset>
        <aui:input label="First Name" name="first-name"></aui:input>
        <aui:input label="Middle Name" name="middle-name"></aui:input>
        <aui:input label="Last Name" name="last-name"></aui:input>
        <aui:input label="Screen Name" name="screen-name"></aui:input>
        <aui:input label="Email Address" name="email-address"></aui:input>
    </aui:fieldset>

        <p>Click the button below to add a new user by invoking Liferay's JSON web services.</p>

        <aui:button-row>
            <aui:button id="add-user" value="Add User">
            </aui:button>
        </aui:button-row>
</aui:form>

<aui:script use="node, event">
var addUserButton = A.one('#add-user');

var firstNameNode = A.one('#<portlet:namespace />first-name');
var middleNameNode = A.one('#<portlet:namespace />middle-name');
var lastNameNode = A.one('#<portlet:namespace />last-name');
var screenNameNode = A.one('#<portlet:namespace />screen-name');
var emailAddressNode = A.one('#<portlet:namespace />email-address');

addUserButton.on('click', function(event) {
        var firstName = firstNameNode.get('value');
        var middleName = middleNameNode.get('value');
        var lastName = lastNameNode.get('value');
        var screenName = screenNameNode.get('value');
        var emailAddress = emailAddressNode.get('value');

    var user = Liferay.Service(
        '/user/add-user',
        {
            companyId: Liferay.ThemeDisplay.getCompanyId(),
            autoPassword: false,
            password1: 'test',
            password2: 'test',
            autoScreenName: false,
            screenName: screenName,
            emailAddress: emailAddress,
            facebookId: 0,
            openId: '',
            locale: 'en_US',
            firstName: firstName,
            middleName: middleName,
            lastName: lastName,
            prefixId: 0,
            suffixId: 0,
            male: true,
            birthdayMonth: 1,
            birthdayDay: 1,
            birthdayYear: 1970,
            jobTitle: 'Tester',
            groupIds: null,
            organizationIds: null,
            roleIds: null,
            userGroupIds: null,
            sendEmail: false,
            serviceContext: {assetTagNames: ['test']}
        },
        function(obj) {
            console.log(obj);

            if (obj.hasOwnProperty('createDate')) {
                window.open('<%= successURL %>', '_self');
            }
            else {
                window.open('<%= failureURL %>', '_self');
            }
        }
    );
});
</aui:script>
```

In this example, it's assumed that the JSP page is part of a web module with a portlet class that extends Liferay's `MVCPortlet` class. This is required since the code uses the `mvcPath` URL parameter. It's also assumed that the JSP code is in a file named `view.jsp`, and that there are also `success.jsp` and `failure.jsp` files in the same directory.

Get User JSON Web Service Invocation via URL

Here's a simple JSON web service invocation via URL that returns the user with the specified email address:

```
http://localhost:8080/api/jsonws/user/get-user-by-email-address/company-id/20154/email-address/test%40liferay.com?p_auth=[value]
```

This web service invocation returns the test@example.com user. After invoking a service via Liferay's JSONWS API page, the URL provided when you click on the *URL Example* tab omits the `p_auth` URL query parameter. It's assumed that you'll add this parameter yourself. Remember that you must be logged in as a user with the required permission to invoke a web service. To find the `p_auth` token that corresponds to your session, see the Invoking JSON Web Services tutorial.

If you read that tutorial, you know that you can supply parameters as either URL path parameters or URL query parameters. In the preceding example, the company ID and email address are supplied as URL path parameters. Here's an equivalent example using URL query parameters:

```
http://localhost:8080/api/jsonws/user/get-user-by-email-address?companyId=20154&emailAddress=test@example.com&p_auth=[value]
```

Next, you'll consider an example that requires many more parameters!

Add User JSON Web Service Invocation via URL

Here's an example JSON web service invocation via URL that adds a new user with the specified attributes:

```
http://localhost:8080/api/jsonws/user/add-user/company-id/20154/auto-password/false/password1/test/password2/test/auto-screen-
name/false/screen-name/joe.bloggs/email-address/joe.bloggs%40liferay.com/facebook-id/0/-open-id/locale/en_US/first-name/Joe/middle-
name/T/last-name/Bloggs/prefix-id/0/suffix-id/0/male/true/birthday-month/1/birthday-day/1/birthday-year/1970/job-title/Tester/-group-
ids/-organization-ids/-role-ids/-user-group-ids/send-email/false?p_auth=[value]
```

And here's the same example using URL query parameters instead of URL path parameters:

```
http://localhost:8080/api/jsonws/user/add-user?companyId=20154&autoPassword=false&password1=test&password2=test&autoScreenName=false&screenName=joe.bloggs&e
openId&locale=en_US&firstName=Joe&middleName=T&lastName=Bloggs&prefixId=0&suffixId=0&male=true&birthdayMonth=1&birthdayDay=1&birthdayYear=1970&jobTitle=Test
groupIds&-organizationIds&-roleIds&-userGroupIds&sendEmail=false&p_auth=[value]
```

Get User JSON Web Service Invocation via cURL

Here's an example JSON web service invocation via the cURL tool that returns the user with the specified email address:

```
curl http://localhost:8080/api/jsonws/user/get-user-by-email-address \
  -u test@example.com:test \
  -d companyId=20154 \
  -d emailAddress='test@example.com'
```

Note that cURL is a command line tool. You can execute this command from a terminal or command prompt.

Add User JSON Web Service Invocation via cURL

Here's an example JSON web service invocation via the cURL tool that adds the user with the specified attributes:

```
curl http://localhost:8080/api/jsonws/user/add-user \
  -u test@example.com:test \
  -d companyId=20154 \
  -d autoPassword=false \
  -d password1='test' \
  -d password2='test' \
  -d autoScreenName=false \
  -d screenName='joe.bloggs' \
  -d emailAddress='joe.bloggs@example.com' \
  -d facebookId=0 \
  -d openId='0' \
  -d locale=en_US \
  -d firstName='Joe' \
  -d middleName='T' \
  -d lastName='Bloggs' \
  -d prefixId=0 \
  -d suffixId=0 \
  -d male=true \
  -d birthdayMonth=1 \
  -d birthdayDay=1 \
  -d birthdayYear=1970 \
  -d jobTitle='Tester' \
  -d groupIds= \
  -d organizationIds= \
  -d roleIds= \
  -d userGroupIds= \
  -d sendEmail=false
```

Great! Now you've seen how to invoke Liferay's JSON web services from JavaScript, URL, and cURL.

Related Topics

Invoking JSON Web Services
    JSON Web Services Invoker
    Invoking Remote Services

85.8 Configuring JSON Web Services

JSON web services are enabled in Liferay by default. If you need to disable them, specify this portal property setting in a `portal-ext.properties` file:

```
json.web.service.enabled=false
```

This tutorial presents other such properties that you can use to fine-tune exactly how JSON web services work in your Liferay instance. You can find these, and other properties, in the portal properties reference documentation. As with the preceding property, you should set portal properties in a `portal-ext.properties` file.

First, you'll learn about setting whether JSON web services are discoverable via the API page.

Discoverability

By default, JSON web services are discoverable via the API page at `http://[address]:[port]/api/jsonws`. To disable this, set the following property:

```
jsonws.web.service.api.discoverable=false
```

Next, you'll learn how to disable HTTP methods.

Disabling HTTP Methods

When strict HTTP method mode is enabled, you can filter web service access based on HTTP methods used by the services. For example, you can set your Liferay instance's JSON web services to work in read-only mode by disabling HTTP methods other than `GET`. For example:

```
jsonws.web.service.invalid.http.methods=DELETE,POST,PUT
```

With this setting, all requests that use `DELETE`, `POST`, or `PUT` HTTP methods are ignored.

Next, you'll learn how to restrict public access to exposed JSON APIs.

Strict HTTP Methods

All JSON web services are mapped to either `GET` or `POST` HTTP methods. If a service method name starts with `get`, `is` or `has`, the service is assumed to be read-only and is bound to the `GET` method. Otherwise, it's bound to `POST`.

By default, Liferay doesn't check `HTTP` methods when invoking a service call; it works in non-strict http method mode, where services may be invoked using any HTTP method. If you need the strict mode, you can set it as follows:

```
jsonws.web.service.strict.http.method=true
```

When using strict mode, you must use the correct HTTP methods to calll service methods. When strict HTTP mode is enabled, you still might need to disable HTTP methods. You'll learn how next.

Controlling Public Access

Each service method knows whether a given user has permission to invoke the chosen action. If you're concerned about security, you can restrict access to exposed JSON APIs by explicitly permitting or restricting certain JSON web service paths.

The property `jsonws.web.service.paths.includes` denotes patterns for JSON web service action paths that are allowed. Set a blank pattern to allow any service action path.

The property `jsonws.web.service.paths.excludes` denotes patterns for JSON web service action paths that aren't allowed even if they match one of the patterns set in `jsonws.web.service.paths.includes`.

Note that these properties support wildcards. For example, if you set `jsonws.web.service.paths.includes=get*,has*,is*`, Liferay makes all read-only JSON methods publicly accessible. All other JSON methods are secured. To disable access to all exposed methods, you can leave the right side of the = symbol empty. To enable access to all exposed methods, specify *. Remember that if a path matches both the `jsonws.web.service.paths.includes` and `jsonws.web.service.paths.excludes` properties, the `jsonws.web.service.paths.excludes` property takes precedence.

Related Topics

Registering JSON Web Services
Creating Remote Services
Invoking Remote Services

85.9 SOAP Web Services

You can access Liferay's web services via Simple Object Access Protocol (SOAP) over HTTP. The packaging protocol is SOAP, and the transport protocol is HTTP.

Note: An authentication token must accompany each Liferay web service invocation. For details, see the tutorial on Service Security Layers.

As an example, consider some example SOAP web service clients for Liferay's `Company`, `User`, and `UserGroup` services that perform these tasks:

1. List each user group the user with the screenname *test* belongs to.
2. Add a new user group named *MyGroup*.
3. Add your Liferay instance's administrative user to the new user group. For demonstration purposes, you'll use an administrative user whose email address is `test@example.com`.

You'll use these SOAP related classes:

```
import com.liferay.portal.kernel.model.CompanySoap;
import com.liferay.portal.kernel.model.CompanySoap;
import com.liferay.portal.kernel.model.UserGroupSoap;
import com.liferay.portal.kernel.model.UserGroupSoap;
import com.liferay.portal.service.http.CompanyServiceSoap;
import com.liferay.portal.service.http.CompanyServiceSoapServiceLocator;
import com.liferay.portal.service.http.UserGroupServiceSoap;
import com.liferay.portal.service.http.UserGroupServiceSoapServiceLocator;
import com.liferay.portal.service.http.UserServiceSoap;
import com.liferay.portal.service.http.UserServiceSoapServiceLocator;
```

Can you see the naming convention for SOAP related classes? These classes have either `-ServiceSoapServiceLocator`, `-ServiceSoap`, or `-Soap` as suffixes. The `-ServiceSoapServiceLocator` class finds the `-ServiceSoap` class via the service's URL you provide. The `-ServiceSoap` class is the interface to the services specified in the Web Services Definition Language (WSDL) file for each service. The `-Soap` classes are the serializable implementations of the models.

So how do you determine the URLs for these services? This is a most excellent question! You can see a list of the services deployed on your Liferay instance by opening your browser to the following URL:

```
http://[host]:[port]/api/axis
```

Note that this URL only lists services in the portal context. To learn how to find services in other contexts in your Liferay instance, see the SOAP sections in the tutorial Creating Remote Services.

Regardless of the context you're viewing SOAP services in, each web service is listed with its name, operations, and a link to its WSDL file. For example, here's the list of secure web services listed for `UserGroup`:

- `Portal_UserGroupService` (wsdl)
  - `addGroupUserGroups`
  - `addTeamUserGroups`
  - `addUserGroup`
  - `deleteUserGroup`
  - `fetchUserGroup`
  - `getUserGroup`
  - `getUserGroups`
  - `getUserUserGroups`
  - `unsetGroupUserGroups`
  - `unsetTeamUserGroups`
  - `updateUserGroup`

Note that some of these methods are overloaded.

Liferay uses Service Builder to automatically generate JSON and SOAP web service interfaces. If you haven't used Service Builder before, see its introductory tutorial.

The WSDL file is written in XML and provides a model for describing and locating the web service. Here's a WSDL excerpt of the `addUserGroup` operation of `UserGroup`:

```
<wsdl:operation name="addUserGroup" parameterOrder="name description">
    <wsdl:input message="intf:addUserGroupRequest" name="addUserGroupRequest"/>
    <wsdl:output message="intf:addUserGroupResponse" name="addUserGroupResponse"/>
</wsdl:operation>
```

To use the service, you pass in the WSDL URL along with your login credentials to the SOAP service locator for your service. The next section shows you an example of this.

SOAP Java Client

Now you'll learn how to invoke Liferay's SOAP web services. As an example, you'll do this by setting up a Java web services client in Eclipse. You can use Eclipse's Web Service Client wizard to either create a new web service client project or add a client to an existing project. You must add a new web service client to your project for each service that you want to consume in your client code. For this example, you'll build a web service client to invoke Liferay's `Company`, `User`, and `UserGroup` services.

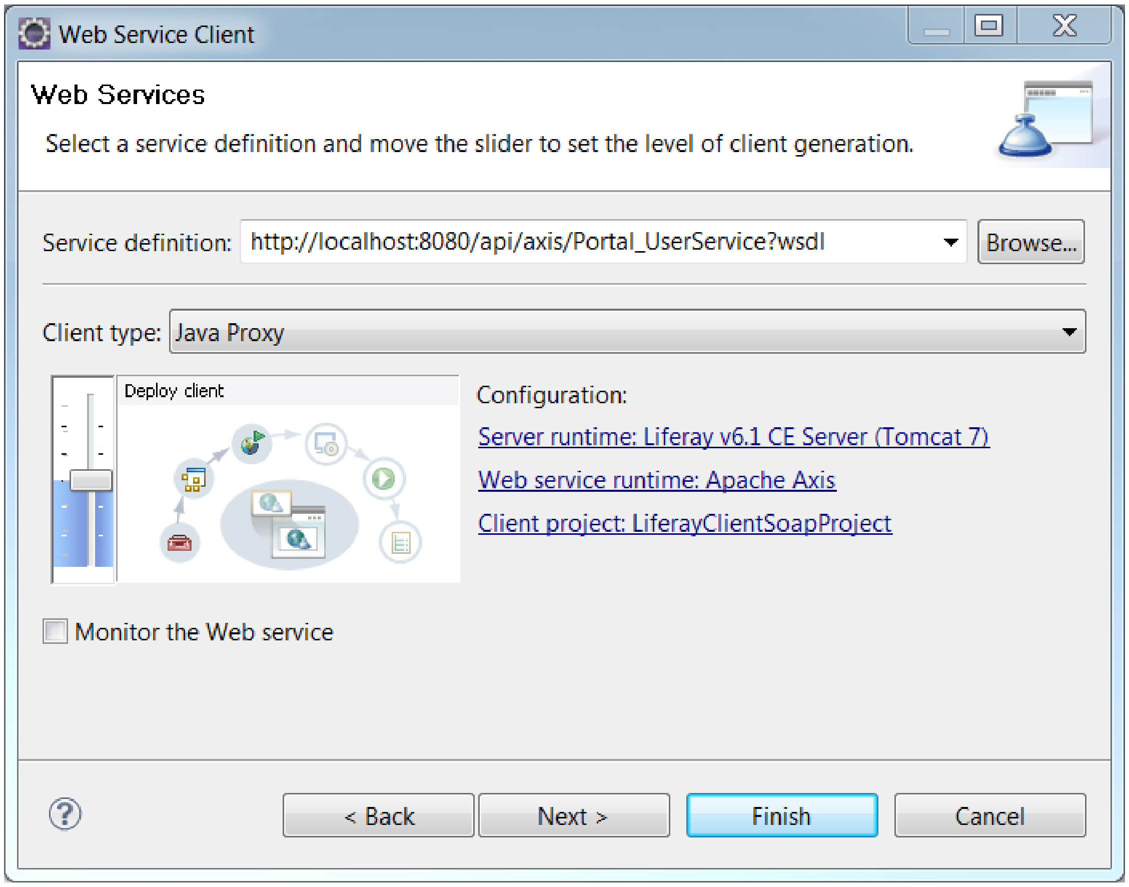
To create a new web service client project in Eclipse, click *File → New → Other...*, then expand the *Web Services* category. Select *Web Service Client*.

For each client you create, you're prompted to enter the service definition (WSDL) for the desired service. Since your example web service client needs Liferay's `Company`, `User`, and `UserGroup` services, enter the following WSDLs:

```
http://localhost:8080/api/axis/Portal_CompanyService?wsdl
```

```
http://localhost:8080/api/axis/Portal_UserService?wsdl
```

```
http://localhost:8080/api/axis/Portal_UserGroupService?wsdl
```


Web Service Client

Web Services

Select a service definition and move the slider to set the level of client generation.

Service definition: http://localhost:8080/api/axis/Portal_UserService?wsdl

Client type: Java Proxy

Configuration:

Server runtime: Liferay v6.1 CE Server (Tomcat 7)

Web service runtime: Apache Axis

Client project: LiferayClientSoapProject

Monitor the Web service

Figure 85.4: Service Definition

When you specify a WSDL, Eclipse automatically adds the auxiliary files and libraries required to consume that web service. After you've created your web service client project using one of the above WSDLs, you need to create additional clients in the project using the remaining WSDLs. To create an additional client in an existing project, right-click on the project and select *New → Other → Web Service Client*. Click *Next*, enter the WSDL, and complete the wizard.

The following code locates and invokes operations to create a new user group named MyUserGroup and add a user with the screen name *test* to it. Create a LiferaySoapClient.java file in your web service client project and add this code to it. If you create this class in a package other than the one that's specified in this code, replace the package with your package. To run the client from Eclipse, make sure that your Liferay server is running, right-click the LiferaySoapClient.java class, and select *Run as Java application*. Check your console to check that your service calls succeeded.

```
package com.liferay.test;

import java.net.URL;

import com.liferay.portal.kernel.model.CompanySoap;
import com.liferay.portal.kernel.model.UserGroupSoap;
import com.liferay.portal.service.http.CompanyServiceSoap;
import com.liferay.portal.service.http.CompanyServiceSoapServiceLocator;
import com.liferay.portal.service.http.UserGroupServiceSoap;
import com.liferay.portal.service.http.UserGroupServiceSoapServiceLocator;
import com.liferay.portal.service.http.UserServiceSoap;
import com.liferay.portal.service.http.UserServiceSoapServiceLocator;
```

```
public class LiferaySoapClient {

    public static void main(String[] args) {

        try {
            String remoteUser = "test";
            String password = "test";
            String virtualHost = "localhost";

            String groupName = "MyUserGroup";

            String serviceCompanyName = "Portal_CompanyService";
            String serviceUserName = "Portal_UserService";
            String serviceUserGroupName = "Portal_UserGroupService";

            long userId = 0;

            // Locate the Company
            CompanyServiceSoapServiceLocator locatorCompany =
                new CompanyServiceSoapServiceLocator();

            CompanyServiceSoap soapCompany =
                locatorCompany.getPortal_CompanyService(
                    _getURL(remoteUser, password, serviceCompanyName,
                            true));

            CompanySoap companySoap =
                soapCompany.getCompanyByVirtualHost(virtualHost);

            // Locate the User service
            UserServiceSoapServiceLocator locatorUser =
                new UserServiceSoapServiceLocator();
            UserServiceSoap userSoap = locatorUser.getPortal_UserService(
                _getURL(remoteUser, password, serviceUserName, true));

            // Get the ID of the remote user
            userId = userSoap.getUserIdByScreenName(
                companySoap.getCompanyId(), remoteUser);
            System.out.println("userId for user named " + remoteUser +
                    " is " + userId);

            // Locate the UserGroup service
            UserGroupServiceSoapServiceLocator locator =
                new UserGroupServiceSoapServiceLocator();
            UserGroupServiceSoap usergroupsoap =
                locator.getPortal_UserGroupService(
                    _getURL(remoteUser, password, serviceUserGroupName,
                            true));

            // Get the user's user groups
            UserGroupSoap[] usergroups = usergroupsoap.getUserUserGroups(
                    userId);

            System.out.println("User groups for userId " + userId + " ...");
            for (int i = 0; i < usergroups.length; i++) {
                System.out.println("\t" + usergroups[i].getName());
            }

            // Adds the user group if it does not already exist
            String groupDesc = "My new user group";
            UserGroupSoap newUserGroup = null;

            boolean userGroupAlreadyExists = false;
            try {
                newUserGroup = usergroupsoap.getUserGroup(groupName);
                if (newUserGroup ≠ null) {
                    System.out.println("User with userId " + userId +
                        " is already a member of UserGroup " +
                                newUserGroup.getName());
                    userGroupAlreadyExists = true;
                }
            } catch (Exception e) {
                // Print cause, but continue
                System.out.println(e.getLocalizedMessage());
            }

            if (!userGroupAlreadyExists) {
                newUserGroup = usergroupsoap.addUserGroup(
                        groupName, groupDesc);
                System.out.println("Added user group named " + groupName);

                long users[] = {userId};
                userSoap.addUserGroupUsers(newUserGroup.getUserGroupId(),
                        users);
            }

            // Get the user's user groups
            usergroups = usergroupsoap.getUserUserGroups(userId);

            System.out.println("User groups for userId " + userId + " ...");
            for (int i = 0; i < usergroups.length; i++) {
                System.out.println("\t" + usergroups[i].getName());
            }
        }
        catch (Exception e) {
            e.getLocalizedMessage();
        }
    }

        private static URL _getURL(String remoteUser, String password,
                        String serviceName, boolean authenticate)
                        throws Exception {

                // Unauthenticated url
                String url = "http://localhost:8080/api/axis/" + serviceName;

                // Authenticated url
                if (authenticate) {
                        url = "http://" + remoteUser + ":" + password
                                        + "@localhost:8080/api/axis/"
                                        + serviceName;
                }

                return new URL(url);
        }

}
```

Running this client should produce output like this:

```
userId for user named test is 10196
User groups for user 10196 ...
java.rmi.RemoteException: No UserGroup exists with the key {companyId=10154,
name=MyUserGroup}
Added user group named
Added user to user group named MyUserGroup
User groups for user 10196 ...
    MyUserGroup
```

The output tells you the user had no groups, but was added to the user group `MyUserGroup`.

You might be thinking, "But an error was thrown! Something is wrong!" Yes, an error is thrown (`java.rmi.RemoteException:`), but you can sit here as cool as an ice cream sandwich all the same. The exception is thrown because the `UserGroup` check is invoked before the `UserGroup` is created. Because the very next line of the output says `Added user group named...`, you're okay. The SOAP web service invocations worked!

Here are a few things to note about this example:

- Authentication is done using HTTP Basic Authentication, which isn't appropriate for a production environment since the password is unencrypted. It's simply used for convenience in this example. In production, you should use SSL. Refer to Liferay's `portal.properties` file and look up the `company.security.auth.requires.https` and `web.server.protocol` properties for more information.
- The screen name and password are passed in the URL as credentials.
- The name of the service (e.g. `Portal_UserGroupService`) is specified at the end of the URL. Remember that the service name can be found in the web service listing.

The operations `getCompanyByVirtualHost()`, `getUserIdByScreenName()`, `getUserUserGroups()`, `addUserGroup()` and `addUserGroupUsers()` are specified for the `-ServiceSOAP` classes `CompanyServiceSoap`, `UserServiceSoap` and `UserGroupServiceSoap` in the WSDL files. Information on parameter types, parameter order, request type, response type, and return type are conveniently specified in the WSDL for each Liferay web service. It's all there for you!

Next, you'll learn how to implement a web service client in PHP.

SOAP PHP Client

You can write your client in any language that supports web services invocation. The following example code invokes the same operations as before, but uses PHP and a PHP SOAP client instead of Java:

```
<?php
    $userGroupName = "MyUserGroup2";
    $userName = "test";
    $clientOptions = array('login' => $userName, 'password' => 'test');

    // Add user group
    $userGroupClient = new
        SoapClient(
            "http://localhost:8080/api/axis/Portal_UserGroupService?wsdl",
            $clientOptions);
    $userGroup = $userGroupClient->addUserGroup($userGroupName,
        "This user group was created by the PHP client! ");
    print ("User group ID is $userGroup->userGroupId ");

    // Add user to user group
    $companyClient = new SoapClient(
        "http://localhost:8080/api/axis/Portal_CompanyService?wsdl",
        $clientOptions);
    $company = $companyClient->getCompanyByVirtualHost("localhost");
    $userClient = new SoapClient(
        "http://localhost:8080/api/axis/Portal_UserService?wsdl",
        $clientOptions);
    $userId = $userClient->getUserIdByScreenName($company->companyId,
        $userName);
    print ("User ID for $userName is $userId ");
    $users = array($userId);
    $userClient->addUserGroupUsers($userGroup->userGroupId, $users);

    // Print the user groups to which the user belongs
    $userGroups = $userGroupClient->getUserUserGroups($userId);
    print ("User groups for user $userId ... ");
    foreach($userGroups as $ug)
        print ("$ug->name, $ug->userGroupId ")
?>
```

Remember, you can implement a web service client in any language that supports SOAP web services.

Related Topics

Service Security Layers
Creating Remote Services
Invoking Remote Services
What is Service Builder?

85.10 JAX-WS and JAX-RS

Liferay supports JAX-WS and JAX-RS via the Apache CXF implementation. Apps can publish JAX web services to the CXF endpoints defined in your Liferay instance. CXF endpoints are effectively context paths the JAX web services are deployed to and accessible from. To publish any kind of JAX web service, one or more CXF endpoints must be defined in your Liferay instance. To access JAX web services, an *extender* must also be configured in your Liferay instance. Extenders specify where the services are deployed and whether they are augmented with handlers, providers, and so on. There are two types of extenders:

1. **SOAP Extenders:** Required to publish JAX-WS web services. Each SOAP extender can deploy the services to one or more CXF endpoints and can use a set of JAX-WS handlers to augment the services.
2. **REST Extenders:** Required to publish JAX-RS web services. REST extenders for JAX-RS services are analogous to SOAP extenders for JAX-WS services. To create JAX-RS services that can work across different JAX-RS implementations, you must provide an implementation of `javax.ws.rs.core.Application` to the OSGi framework. You can do this by registering an instance of this implementation as an OSGi service via `BundleContext` or the Declarative Services `@Component` annotation. The JAX-RS application encompasses the services that represent JAX-RS endpoints and the services that represent JAX-RS providers. By specifying OSGi filters in a REST extender, you can also dynamically add endpoints or JAX-RS providers to a JAX-RS application.

SOAP extenders and REST extenders are subsystems that track the services the app developer registers in OSGi (those matching the provided OSGi filters), and deploy them under the specified CXF endpoints. For example, if you create the CXF endpoints `/soap` and `/rest`, you could later create a REST extender for `/rest` that publishes REST applications, and a SOAP extender for `/soap` that publishes SOAP services. Of course, this is only a rough example: you can fine tune things to your liking.

CXF endpoints and both types of extenders can be created programmatically or with Liferay's Control Panel. This tutorial shows you how to do both, and then shows you how to publish JAX-WS and JAX-RS web services. The following topics are covered:

- Configuring Endpoints and Extenders with the Control Panel
  - CXF Endpoints
  - SOAP Extenders
  - REST Extenders
- Configuring Endpoints and Extenders Programmatically
- Publishing JAX-WS Web Services
- Publishing JAX-RS Web Services

Configuring Endpoints and Extenders with the Control Panel

Liferay's Control Panel lets administrators configure endpoints and extenders for JAX web services. Note that you must be an administrator in your Liferay instance to access the settings here. First, you'll learn how to create CXF endpoints.

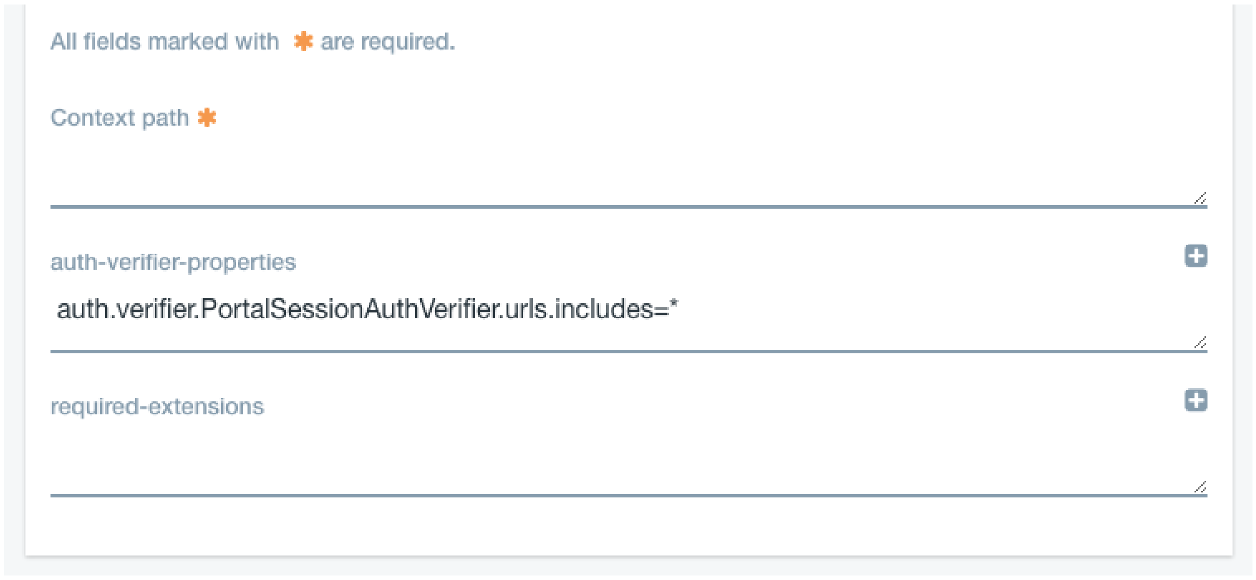
CXF Endpoints

To configure a CXF endpoint with the Control Panel, first go to *Control Panel* → *Configuration* → *System Settings* → *Foundation*. Then select *CXF Endpoints* from the table. If there are any existing CXF endpoints, they're shown here. To add a new one, select the *Add* button (+) in the lower right-hand corner. The form that appears lets you configure a new CXF endpoint by filling out these fields:

- **Context path:** The path the JAX web services are deployed to on the Liferay server. For example, if you define the context path `/web-services`, any services deployed there are available at `http://your-server:your-port/o/web-services`.

- **`AuthVerifier` properties:** Any properties defined here are passed as-is to the `AuthVerifier` filter. See the `AuthVerifier` documentation for more details.

- **Required extensions:** CXF normally loads its default extension classes, but in some cases you can override them to replace the default behavior. In most cases, you can leave this field blank: overriding extensions isn't common. By specifying custom extensions here via OSGi filters, Liferay waits until those extensions are registered in the OSGi framework before creating the CXF servlet and passing the extensions to the servlet.



All fields marked with * are required.

Context path *

auth-verifier-properties

auth.verifier.PortalSessionAuthVerifier.urls.includes=*

required-extensions

Figure 85.5: Fill out this form to create a CXF endpoint.

Next, you'll learn how to use the Control Panel to create SOAP extenders for JAX-WS web services.

SOAP Extenders

For an app to deploy JAX-WS web services, you must configure a SOAP extender. To configure a SOAP extender with the Control Panel, first go to *Control Panel* → *Configuration* → *System Settings* → *Foundation*. Then select *SOAP Extenders* from the table. If there are any existing SOAP extenders, they're shown here. To add a new one, select the *Add* button (+) in the lower right-hand corner. The form that appears lets you configure a new SOAP extender by filling out these fields:

- **Context paths:** Specify at least one CXF endpoint here. This is where the services affected by this extender are deployed. In the preceding CXF endpoint example, this would be `/web-services`. Note that you can specify more than one CXF endpoint here.

- **jax.ws.handler.filters:** Here you can specify a set of OSGi filters that select certain services registered in the OSGi framework. The selected services should implement JAX-WS handlers and augment the JAX-WS services specified in the *jax.ws.service.filters* property. These JAX-WS handlers apply to each service selected in this extender.

- **jax.ws.service.filters:** Here you can specify a set of OSGi filters that select the services registered in the OSGi framework that will be deployed to the CXF endpoints. These OSGi services must be proper JAX-WS services.

- **soap.descriptor.builder:** Leave this option empty to use JAX-WS annotations to describe the SOAP service. To use a different way to describe the SOAP service, you can provide an OSGi filter here that selects an implementation of `com.liferay.portal.remote.soap.extender.SoapDescriptorBuilder`.

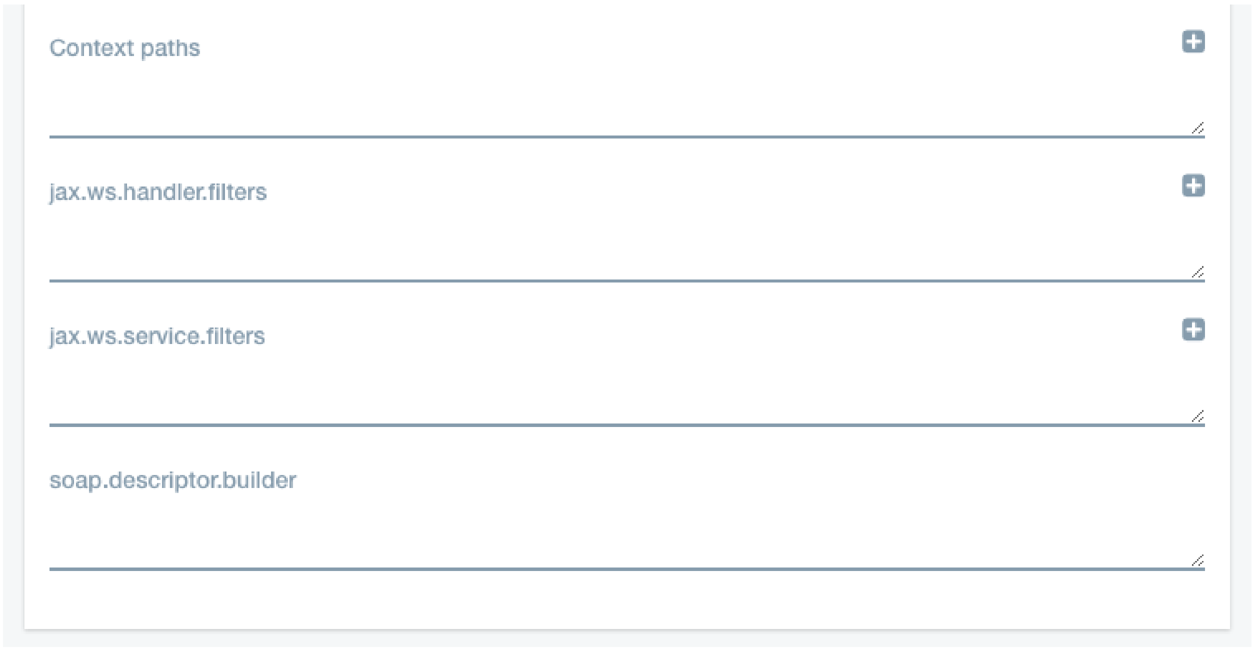


Figure 85.6: Fill out this form to create a SOAP extender.

Next, you'll learn how to use the Control Panel to create REST extenders for JAX-RS web services.

REST Extenders

To configure a REST extender with the Control Panel, first go to *Control Panel* → *Configuration* → *System Settings* → *Foundation*. Then select *REST Extender* from the table. If there are any existing REST extenders, they're shown here. To add a new one, select the *Add* button (+) in the lower right-hand corner. The form that appears lets you configure a new REST extender by filling out these fields:

- **Context paths:** Specify at least one CXF endpoint here. This is where the services affected by this extender are deployed. In the preceding CXF endpoint example, this would be `/web-services`. Note that you can specify more than one CXF endpoint here. This works the same way as the *Context paths* setting in SOAP Extenders.

- **jax.rs.application.filters:** Here you can specify a set of OSGi filters that select services that implement `javax.ws.rs.core.Application`. These JAX-RS applications are deployed to the CXF endpoints specified in the *Context paths* property.

- **jsx.rs.provider.filters:** Here you can specify a set of OSGi filters that select services registered in the OSGi framework. The selected services should implement any of the interfaces supported by JAX-RS for providers. These JAX-RS providers are added to the JAX-RS application as if they had been returned by the `getSingletons()` method of `javax.ws.rs.core.Application`. The following links list some of the supported JAX-RS providers:

  - JAX-RS Entity Providers
  - Filters and Interceptors

- **jax.rs.service.filters:** Here you can specify a set of OSGi filters that selects services registered in the OSGi framework that are valid JAX-RS endpoints. These endpoints are added to the JAX-RS application as if they had been returned by the `getSingletons()` method of `javax.ws.rs.core.Application`.

Next, you'll learn how to configure endpoints and extenders programmatically.

Configuring Endpoints and Extenders Programmatically

To configure endpoints or extenders programmatically, you must use Liferay's configurator extender. The configurator extender provides a way for OSGi modules to deploy default configuration values. Modules that use the configurator extender must provide a `ConfigurationPath` header that points to the configuration files' location inside the module. For example, the following configuration sets the `ConfigurationPath` to `src/main/resources/configuration`:

```
Bundle-Name: Liferay Export Import Service JAX-WS
Bundle-SymbolicName: com.liferay.exportimport.service.jaxws
Bundle-Version: 1.0.0
Liferay-Configuration-Path: /configuration
Include-Resource: configuration=src/main/resources/configuration
Liferay-Releng-Module-Group-Description:
Liferay-Releng-Module-Group-Title: Data Management
```

Note: If you're using any version before Liferay CE GA4 or Liferay DXP Fixpack 22, the `Liferay-Configuration-Path` directive above is `Configuration-Path`. As of LPS-62571, Liferay-specific Bnd instructions are prefixed with `Liferay` to avoid conflicts.

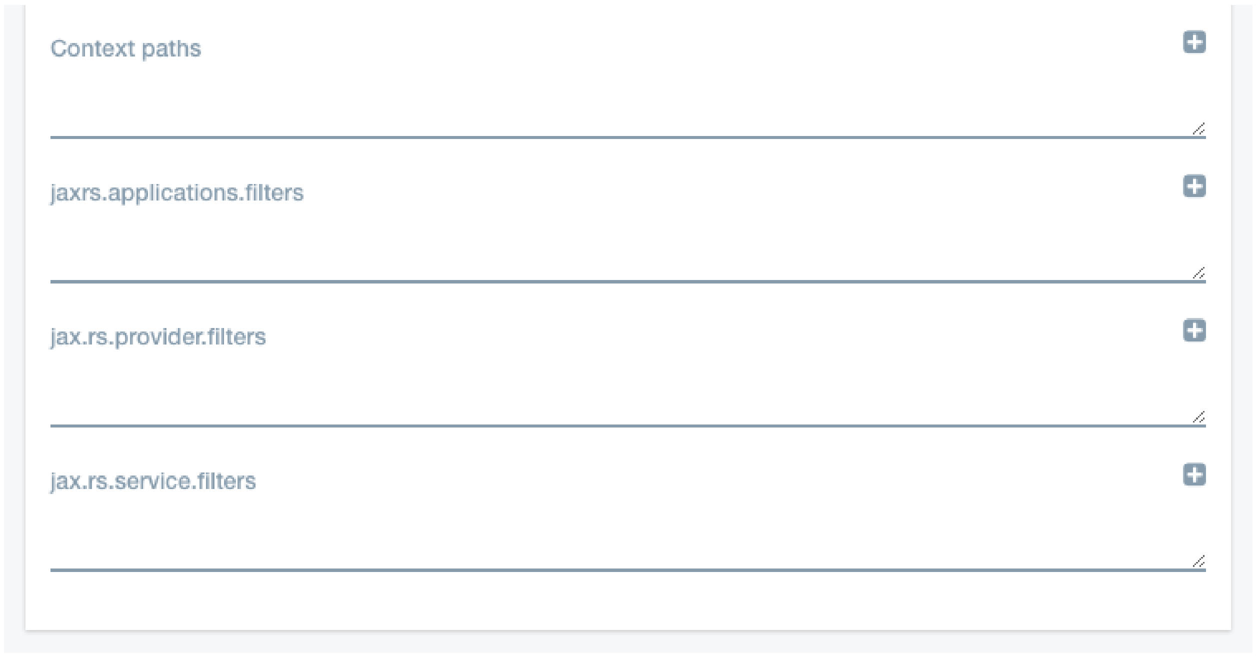


Figure 85.7: Fill out this form to create a REST extender.

There are two different configuration types in OSGi's `ConfigurationAdmin`: single, and factory. Factory configurations can have several configuration instances per factory name. Liferay DXP uses factory configurations. You must provide a factory configuration's default values in a `*.properties` file. In this properties file, use a suffix on the end of the PID (persistent identifier) and then provide your settings. For example, the following code uses the `-staging` suffix on the PID and creates a CXF endpoint at the context path `/staging-ws`:

```
com.liferay.portal.remote.cxf.common.configuration.CXFEndpointPublisherConfiguration-staging.properties:

contextPath=/staging-ws
```

As another example, the following code uses the suffix `-stagingjaxws` on the PID and creates a SOAP extender at the context path `/staging-ws`. This code also includes settings for the configuration fields `jaxWsHandlerFilterStrings` and `jaxWsServiceFilterStrings`:

```
com.liferay.portal.remote.soap.extender.configuration.SoapExtenderConfiguration-stagingjaxws.properties:

contextPaths=/staging-ws
jaxWsHandlerFilterStrings=(staging.jax.ws.handler=true)
jaxWsServiceFilterStrings=(staging.jax.ws.service=true)
```

You must then use these configuration fields in the configuration class. For example, the `SoapExtenderConfiguration` interface below contains the configuration fields `contextPaths`, `jaxWsHandlerFilterStrings`, and `jaxWsServiceFilterStrings`:

```
@ExtendedObjectClassDefinition(
    category = "foundation", factoryInstanceLabelAttribute = "contextPaths"
)
@Meta.OCD(
    factory = true,
    id = "com.liferay.portal.remote.soap.extender.configuration.SoapExtenderConfiguration",
    localization = "content/Language", name = "soap.extender.configuration.name"
)
public interface SoapExtenderConfiguration {

    @Meta.AD(required = false)
    public String[] contextPaths();

    @Meta.AD(name = "jax.ws.handler.filters", required = false)
    public String[] jaxWsHandlerFilterStrings();

    @Meta.AD(name = "jax.ws.service.filters", required = false)
    public String[] jaxWsServiceFilterStrings();

    @Meta.AD(name = "soap.descriptor.builder", required = false)
    public String soapDescriptorBuilderFilter();

}
```

You can use similar techniques to create REST extenders. For example, see the `RestExtenderConfiguration` interface in Liferay's source code.

Next, you'll learn how to publish JAX-WS web services.

Publishing JAX-WS Web Services

To publish JAX-WS web services via SOAP in a 7.0 module, annotate the class and its methods with standard JAX-WS annotations, and then register it as a service in the OSGi framework. For example, the following class uses the `@WebService` annotation for the class and `@WebMethod` annotations for its methods. You must also set the `jaxws` property to true in the OSGi `@Component` annotation:

```
import javax.jws.WebMethod;
import javax.jws.WebService;

import org.osgi.service.component.annotations.Component;

@Component(
    immediate = true, property = "jaxws=true", service = Calculator.class
)
@WebService
public class Calculator {

    @WebMethod
    public int divide(int a, int b) {
        return a / b;
    }

    @WebMethod
    public int multiply(int a, int b) {
        return a * b;
    }

    @WebMethod
    public int subtract(int a, int b) {
        return a - b;
    }

    @WebMethod
    public int sum(int a, int b) {
        return a + b;
    }

}
```

You should also make sure that you include `org.osgi.core` and `org.osgi.service.component.annotations` as dependencies to your project.

Next, you'll learn how to publish JAX-RS web services.

Publishing JAX-RS Web Services

You can publish JAX-RS web services in a Liferay module the same way you would outside of Liferay. You must also, however, register the class in the OSGi framework. Note that the services must match the OSGi filters provided in the respective extenders. This is how the instances that become services are selected. There's no classpath scanning or other automatic mechanism at work here: it's the developer's responsibility to register the services in the OSGi framework.

The following example registers an OSGi component that publishes a JAX-RS web service at `/application-path/hello`. Get requests to this web service return a simple *"Hello!"*:

```
import org.osgi.service.component.annotations.Component;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Application;
import java.util.Collections;
import java.util.Set;

@Component(immediate = true, service = Application.class)
@ApplicationPath("/application-path")
public class RestEndpoint extends Application {

    public Set<Object> getSingletons() {
        return Collections.<Object>singleton(this);
    }

    @GET
    @Path("/hello")
    @Produces("text/text")
    public String sayHello() {
        return "Hello!";
    }
}
```

Nice work! Now you know how JAX-WS and JAX-RS works in Liferay.

Related Topics

Service Builder Web Services

85.11 Liferay WebSocket Whiteboard

Modern web apps exchange large amounts of data with clients. The WebSockets specification lets this exchange occur over a full-duplex connection that remains open, therefore enabling real-time communication. This approach is more efficient than techniques like long polling, which open two connections to emulate a full-duplex connection. Click here for more information on WebSockets.

Since WebSockets are ubiquitous throughout the web and in all modern browsers, you need a way to register new WebSocket endpoints in Liferay DXP. The *Liferay WebSocket Whiteboard* lets you define new WebSocket endpoints as regular OSGi services. This tutorial shows you how to do this. Onward!

Configuring a Non-Liferay OSGi Container

There may be instances where you want to use a Liferay OSGi module in a non-Liferay OSGi container, and need to define a WebSocket endpoint. To do this, you must register a `javax.servlet.ServletContext` service with the property `websocket.active` set to `true`:

```
Dictionary<String, Object> servletContextProps = new Hashtable<String, Object>();
servletContextProps.put("websocket.active", Boolean.TRUE);

bundleContext.registerService(ServletContext.class, servletContext, servletContextProps);
```

You must also configure the `javax.websocket-api`'s `ServiceLoader`. You can do this by creating your own module as a `javax.websocket-api` fragment. Here's an example of a manifest for such a module:

```
Fragment-Host: javax.websocket-api

Require-Capability:\
    osgi.serviceloader;\

filter:='(osgi.serviceloader=javax.websocket.server.ServerEndpointConfig$Configurator)';\
        cardinality:=multiple,\
    osgi.extender;\
        filter:='(osgi.extender=osgi.serviceloader.processor)'
```

Next, you'll learn how to define a new WebSocket server endpoint in a Liferay OSGi container.

Configuring a Liferay OSGi Container

Defining a new WebSocket server endpoint in Liferay DXP is straightforward. Follow these steps:

1. If you're running Liferay Portal 7.0.2 GA3 or Liferay Digital Enterprise 7.0 Fix Pack 7 or earlier, add the following property to your `portal-ext.properties` file. Otherwise, you can skip this and move on to the next step.

   ```
   module.framework.system.packages.extra=\
       com.ibm.crypto.provider,\
       com.ibm.db2.jcc,\
       com.microsoft.sqlserver.jdbc,\
       com.mysql.jdbc,\
       com.p6spy.engine.spy,\
       com.sun.security.auth.module,\
       com.sybase.jdbc4.jdbc,\
       oracle.jdbc,\
       org.postgresql,\
       org.apache.naming.java,\
       org.hsqldb.jdbc,\
       org.mariadb.jdbc,\
       sun.misc,\
       sun.net.util,\
       sun.security.provider,\
       javax.websocket;version="1.1.0",\
       javax.websocket.server;version="1.1.0"
   ```

2. Deploy the Liferay WebSocket Whiteboard module (`com.liferay.websocket.whiteboard`) to your Liferay DXP instance. You can download this module from JCenter or Maven Central by clicking the respective link:

   - JCenter

- Maven Central

3. In your module's build file, add a dependency to the Liferay WebSocket Whiteboard:

   ```
   com.liferay:com.liferay.websocket.whiteboard:1.0.1
   ```

4. In your module, define a WebSocket server endpoint as you normally would. Note, however, that Liferay DXP doesn't currently support the annotation-driven approach; only the interface-driven approach is supported. To create a WebSocket server endpoint, register an OSGi Service for `javax.websocket.Endpoint.class` with the following properties:

   - `org.osgi.http.websocket.endpoint.path`: the WebSocket's path (required)
   - `org.osgi.http.websocket.endpoint.decoders`: the WebSocket's decoders (optional)
   - `org.osgi.http.websocket.endpoint.encoders`: the WebSocket's encoders (optional)
   - `org.osgi.http.websocket.endpoint.subprotocols`: the WebSocket's subprotocols (optional)

For example, the following steps show you how to define a WebSocket endpoint in a portlet. For the purposes of this example, the portlet also contains a client that communicates with the endpoint. This example portlet, Echo Portlet, uses WebSocket functionality to echo a simple message the client sends to the server.

Although the following steps show only code snippets, you can click here to see the complete example code.

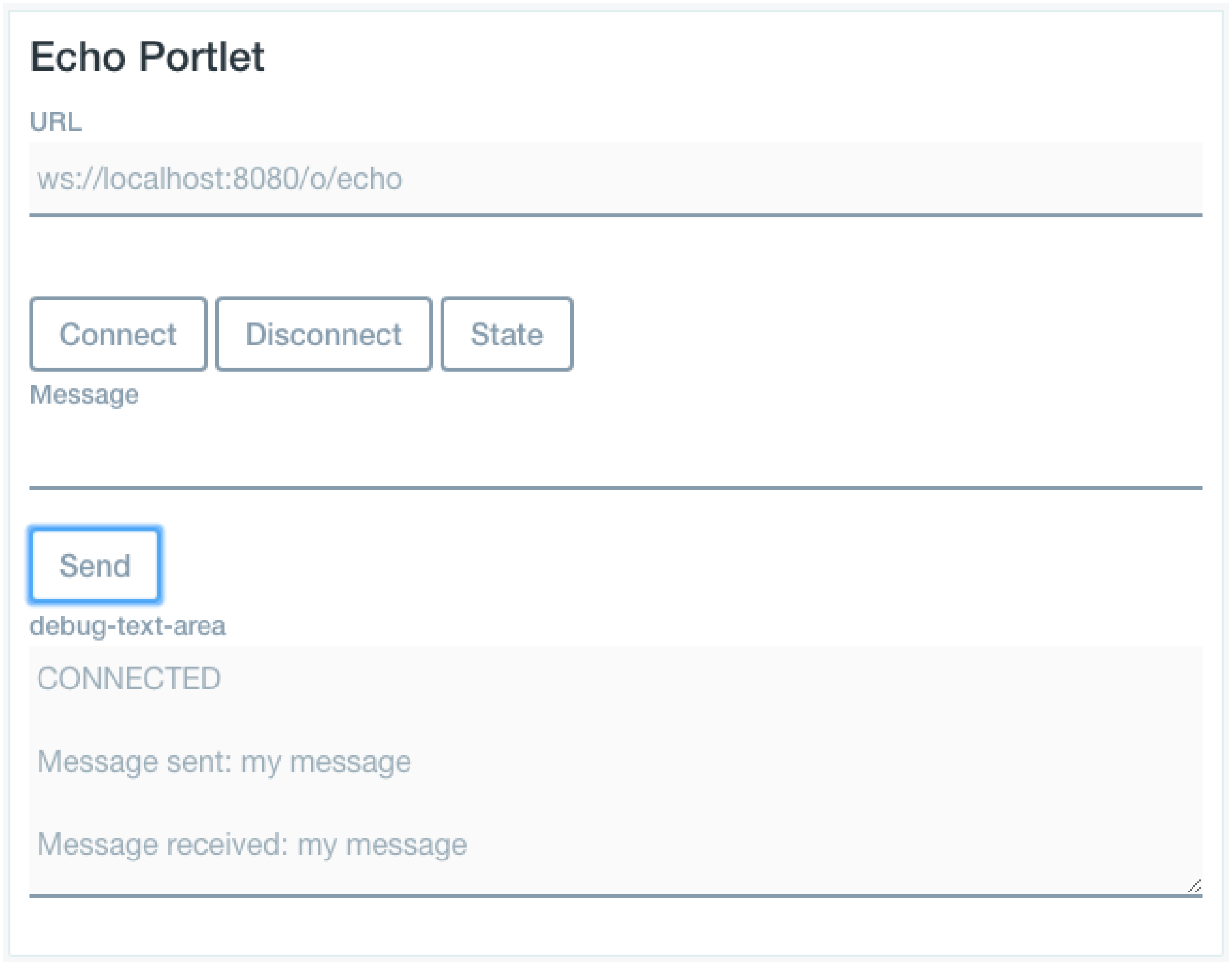
Use these steps to define a WebSocket endpoint:

1. Add the WebSocket dependency to your module's build file. For example, here's the dependency in a `build.gradle` file:

   ```
   javax.websocket:javax.websocket-api:1.1
   ```

2. Create the WebSocket endpoint. Note that the `@Component` annotation contains the required property `org.osgi.http.websocket.endpoint.path`, which defines the endpoint `/o/echo`. Also note that `service = Endpoint.class` in the `@Component` annotation registers this class as an `Endpoint` service in Liferay DXP's OSGi framework. Otherwise, there's nothing special about the `EchoWebSocketEndpoint` class's code; it resembles that of any other WebSocket endpoint:

   ```
   @Component(
       immediate = true,
       property = {"org.osgi.http.websocket.endpoint.path=/o/echo"},
       service = Endpoint.class
   )
   public class EchoWebSocketEndpoint extends Endpoint {

       @Override
       public void onOpen(final Session session, EndpointConfig endpointConfig) {
           session.addMessageHandler(
               new MessageHandler.Whole<String>() {
               @Override
               public void onMessage(String text) {
                   try {
                       RemoteEndpoint.Basic remoteEndpoint =
                           session.getBasicRemote();
                       remoteEndpoint.sendText(text);
                   }
   ```



CONNECTED

Message sent: my message

Message received: my message

Figure 85.8: The example Echo portlet sends and receives a simple message via a WebSocket endpoint.

```
                    catch (IOException ioe) {
                        throw new RuntimeException(ioe);
                    }
                }
            });
        }
    }
```

3. Write your client code. In this example, the Echo portlet's `view.jsp` defines a WebSocket client. Again, there's nothing special about this code; it resembles that of other WebSocket clients:

```
<%@ include file="/init.jsp" %>

<div id="content">
    <div id="left_col">
        <strong>Websocket URL:</strong>
        <br />
        <input id="urlInputText" type="text" readonly />
        </br>
        </br>
        <button onClick="initWebSocket();">Connect</button>
        <button onClick="stopWebSocket();">Disconnect</button>
        <button onClick="checkSocket();">State</button>
        <br />
        <br />
        <strong>Message:</strong>
        <br />
        <input id="inputText" onkeydown="if(event.keyCode==13)sendMessage();" type="text" />
         <button onClick="sendMessage();">Send</button>
     </div>

     <div id="right_col">
         <strong>Log:</strong>
         <br />
         <textarea id="debugTextArea" style="width:400px;height:200px;" readonly></textarea>
     </div>
</div>

<script type="text/javascript">
    var debugTextArea = document.getElementById("debugTextArea");

    var wsUri = "ws://localhost:8080/o/echo";

    urlInputText.value=wsUri;

    resizeUrlInputText(urlInputText.value);

    function resizeUrlInputText(message) {
        urlInputText.size=message.length;
    }

    function debug(message) {
        debugTextArea.value += message + "\n\n";
        debugTextArea.scrollTop = debugTextArea.scrollHeight;
    }

    function sendMessage() {
        var msg = document.getElementById("inputText").value;
        if ( websocket != null ) {
            document.getElementById("inputText").value = "";
            websocket.send(msg);
            debug("Message sent: " + msg);
            console.log( "string sent :", '"'+msg+'"' );
        }
        else {
            debug("Can't sent message, the connection is not open");
        }
    }

    var websocket = null;

    function initWebSocket() {
        try {
            if (typeof MozWebSocket == 'function')
                WebSocket = MozWebSocket;
            if ( websocket && websocket.readyState == 1 )
                websocket.close();
            websocket = new WebSocket( wsUri );
            websocket.onopen = function(evt) {
                debug("CONNECTED");
            };
            websocket.onclose = function(evt) {
                debug("DISCONNECTED");
            };
            websocket.onmessage = function(evt) {
                console.log( "Message received: ", evt.data );
                debug( "Message received: " + evt.data );
            };
            websocket.onerror = function(evt) {
                debug('ERROR: ' + evt.data);
            };
        }
        catch (exception) {
            debug('ERROR: ' + exception);
        }
    }

    function stopWebSocket() {
        if (websocket) {
            websocket.close();
        }
    }

    function checkSocket() {
        if (websocket != null) {
            var stateStr;
            switch (websocket.readyState) {
                case 0: {
                    stateStr = "CONNECTING";
                    break;
                }
                case 1: {
                    stateStr = "OPEN";
                break;
            }
            case 2: {
                stateStr = "CLOSING";
                break;
            }
            case 3: {
                stateStr = "CLOSED";
                break;
            }
            default: {
                stateStr = "UNKNOW";
                break;
            }
        }
        debug("WebSocket state = " + websocket.readyState + " ( " + stateStr + " )");
        } else {
            debug("WebSocket is null");
        }
    }
</script>
```

That's it! Now you know how to create WebSocket endpoints in Liferay DXP.

Related Topics

JAX-WS and JAX-RS

Service Builder Web Services

Chapter 86

Asset Framework

Liferay's asset framework is a system that allows you to add core Liferay features to your application. For example, if you've built an event management application that displays a list of upcoming events, you can use the asset framework to let users add tags, categories, or comments to make entries more self-descriptive.

Tags, categories, and comments are just a few of the features in Liferay's asset framework. You'll also find it easy to use: you'll be infusing your application with these features in no time.

As background, the term *asset* refers to any type of content in the portal. This could be text, a file, a URL, an image, documents, blog entries, bookmarks, wiki pages, or anything you create in your applications.

The asset framework tutorials assume that you've used Liferay's Service Builder to generate your persistence layer, that you've implemented permissions on the entities that you're persisting, and that you've enabled them for search and indexing. You can learn more about Liferay's Service Builder and how to use it in the Service Builder tutorial section.

If you've yet to do any of those things, you can see how each is done in respective tutorials Generating the Back-end, Using Resources and Permissions, and Leveraging Search. Lastly, the Learning Path Assets: Integrating with Liferay's Framework takes you through the fundamentals of enabling an example application's custom entities to use the asset framework. If you haven't traveled through that Learning Path, we recommend you do so before continuing with the tutorials in this section.

The tutorials that follow in this section explore the details of leveraging the asset framework's various features. Here are some features that you'll give your users as you implement them in your app:

- Extensively render your assets.
- Associate tags to custom content types. Users can create and assign new tags or use existing tags.
- Associate categories to custom content types.
- Manage tags from the Control Panel. Administrators can even merge tags.
- Manage categories from the Control Panel. This includes the ability to create category hierarchies.
- Relate assets to one another.

At this point, you might be saying, "Liferay's asset framework sounds great, but how do I leverage all of its awesome features?" Excellent question, and perfect timing!

Before diving head first into the tutorials, you must implement a way to let the framework know whenever any of your custom content entries is added, updated, or deleted. The next tutorial covers that. From that point onward, each tutorial shows you how to leverage a particular asset framework feature in your UI. It's time to start your asset framework training!

86.1 Related Topics

What is Service Builder

Service Builder Persistence

Configuration

86.2 Adding, Updating, and Deleting Assets for Custom Entities

To use Liferay's asset framework with an entity, you must inform the asset framework about each entity instance you create, modify, and delete. In this sense, it's similar to informing Liferay's permissions framework about a new resource. All you have to do is invoke a method of the asset framework that associates an `AssetEntry` with the entity so Liferay can keep track of the entity as an asset. When it's time to update the entity, you update the asset at the same time. To see how to asset-enable entities in a working example portlet, visit the tutorial Assets: Integrating with Liferay's Framework.

To leverage assets, you must also implement indexers for your portlet's entities. Liferay's asset framework uses indexers to manage assets. For instructions on creating an indexer in a working example portlet, see the tutorial Enabling Search and Indexing.

This tutorial shows you how to enable assets for your custom entities and implement indexes for them. It's time to get started!

Preparing Your Project for the Asset Framework

In your project's `service.xml` file, add an asset entry entity reference for your custom entity. Add the following `reference` tag before your custom entity's closing `</entity>` tag.

```
<reference package-path="com.liferay.portlet.asset" entity="AssetEntry" />
```

Then run Service Builder.

Now you're ready to implement adding and updating assets!

Adding and Updating Assets

Your `-LocalServiceImpl` Java class inherits from its parent base class an `AssetEntryLocalService` instance; it's assigned to the variable `assetEntryLocalService`. To add your custom entity as a Liferay asset, you must invoke the `assetEntryLocalService`'s `updateEntry` method.

Here's what the `updateEntry` method's signature looks like:

```
AssetEntry updateEntry(
    long userId, long groupId, Date createDate, Date modifiedDate,
    String className, long classPK, String classUuid, long classTypeId,
    long[] categoryIds, String[] tagNames, boolean listable,
    boolean visible, Date startDate, Date endDate, Date publishDate,
    Date expirationDate, String mimeType, String title,
    String description, String summary, String url, String layoutUuid,
    int height, int width, Double priority)
throws PortalException
```

Here are descriptions of each of the `updateEntry` method's parameters:

- `userId`: identifies the user updating the content.
- `groupId`: identifies the scope of the created content. If your content doesn't support scopes (extremely rare), pass `0` as the value.

- `createDate`: the date the entity was created.
- `modifiedDate`: the date of this change to the entity.
- `className`: identifies the entity's class. The recommended convention is to use the name of the Java class that represents your content type. For example, you can pass in the value returned from `[YourClassName].class.getName()`.
- `classPK`: identifies the specific entity instance, distinguishing it from other instances of the same type. It's usually the primary key of the table where the entity is stored.
- `classUuid`: serves as a secondary identifier that's guaranteed to be universally unique. It correlates entity instances across scopes. It's especially useful if your content is exported and imported across separate portals.
- `classTypeId`: identifies the particular variation of this class, if it has any variations. Otherwise, use `0`.
- `categoryIds`: represent the categories selected for the entity. The asset framework stores them for you.
- `tagNames`: represent the tags selected for the entity. The asset framework stores them for you.
- `listable`: specifies whether the entity can be shown in dynamic lists of content (such as asset publisher configured dynamically).
- `visible`: specifies whether the entity is approved.
- `startDate`: the entity's publish date. You can use it to specify when an Asset Publisher should show the entity's content.
- `endDate`: the date the entity is taken down. You can use it to specify when an Asset Publisher should stop showing the entity's content.
- `publishDate`: the date the entity will start to be shown.
- `expirationDate`: the date the entity will no longer be shown.
- `mimetype`: the Multi-Purpose Internet Mail Extensions type, such as ContentTypes.TEXT_HTML, used for the content.
- `title`: the entity's name.
- `description`: a `String`-based textual description of the entity.
- `summary`: a shortened or truncated sample of the entity's content.
- `url`: a URL to optionally associate with the entity.
- `layoutUuid`: the universally unique ID of the layout of the entry's default display page.
- `height`: this can be set to `0`.
- `width`: this can be set to `0`.
- `priority`: specifies how the entity is ranked among peer entity instances. Low numbers take priority over higher numbers.

The following code from Liferay's Wiki application's WikiPageLocalServiceImpl Java class demonstrates invoking the `updateEntry` method on the wiki page entity called `WikiPage`. In your `add-` method, you could invoke `updateEntry` after adding your entity's resources. Likewise, in your `update-` method, you could invoke `updateEntry` after calling the `super.update-` method. The code below is called in the `WikiPageLocalServiceImpl` class's `updateStatus(...)` method.

```
AssetEntry assetEntry = assetEntryLocalService.updateEntry(
    userId, page.getGroupId(), page.getCreateDate(),
    page.getModifiedDate(), WikiPage.class.getName(),
    page.getResourcePrimKey(), page.getUuid(), 0,
    assetCategoryIds, assetTagNames, true, true, null, null,
    page.getCreateDate(), null, ContentTypes.TEXT_HTML,
    page.getTitle(), null, null, null, null, 0, 0, null);

Indexer<JournalArticle> indexer = IndexerRegistryUtil.nullSafeGetIndexer(
    WikiPage.class);

indexer.reindex(page);
```

Immediately after invoking the `updateEntry` method, you must update the respective asset and index the entity instance. The above code calls the indexer to index (or re-index, if updating) the entity. It's that easy to update assets and indexes.

Tip: The current user's ID and the scope group ID are commonly made available in service context parameters. If the service context you use contains them, then you can access them in calls like these:
long userId = serviceContext.getUserId(); long groupId = serviceContext.getScopeGroupId();

Next, you'll learn what's needed to properly delete an entity that's associated with an asset.

Deleting Assets

When deleting your entities, you should delete the associated assets and indexes at the same time. This cleans up stored asset and index information, which keeps the Asset Publisher from showing information for the entities you've deleted.

In your `-LocalServiceImpl` Java class, open your `delete-` method. After the code that deletes the entity's resource, you must delete the entity instance's asset entry and index.

Here's some code which deletes an asset entry and an index associated with a portlet's entity.

```
assetEntryLocalService.deleteEntry(
    ENTITY.class.getName(), ENTITY.getInsultId());

Indexer indexer = IndexerRegistryUtil.nullSafeGetIndexer(ENTITY.class);
indexer.delete(ENTITY);
```

In your `-LocalServiceImpl` class, you can write similar code. Replace the *ENTITY* class name and variable with your entity's name.

Important: In order for Liferay's Asset Publisher application to show your entity, the entity must have an Asset Renderer. To learn how to implement an Asset Renderer for your custom entity, refer to the Rendering an Asset tutorial. Note also that an Asset Renderer is how you show a user the components of your entity in the Asset Publisher. On deploying your portlet with asset, indexer, and asset rendering implementations in place, an Asset Publisher can show your custom entities!

Great! Now you know how to add, update, and delete assets in your apps!

Related Topics

Relating Assets

What is Service Builder?

86.3 Implementing Asset Categorization and Tagging

In this tutorial, you'll allow content authors the ability to specify tags and categories for their entities in the UI. Liferay provides a set of JSP tags for showing category and tag inputs in your UI. Before beginning, your entities should be asset-enabled and you should have asset renderers enabled for them.

Now it's time to get started!

You can use the following tags in the JSPs you provide for adding/editing custom entities. Here's what the tags look like in the edit_entry.jsp for the Blogs portlet:

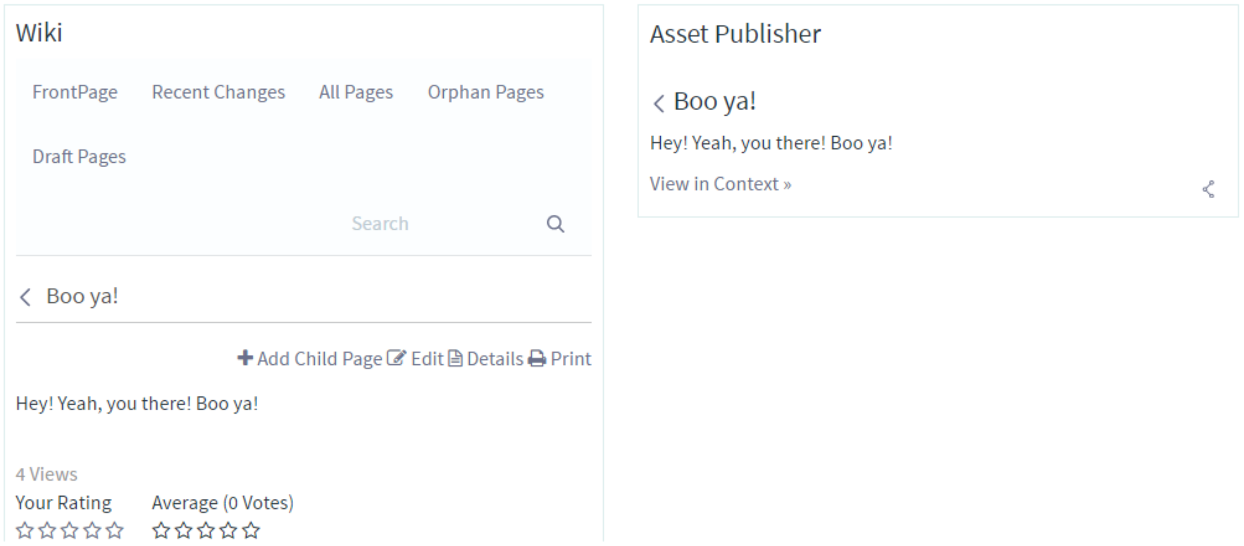


Figure 86.1: It can be useful to show custom entities, like this wiki page entity, in a JSP or in an Asset Publisher.

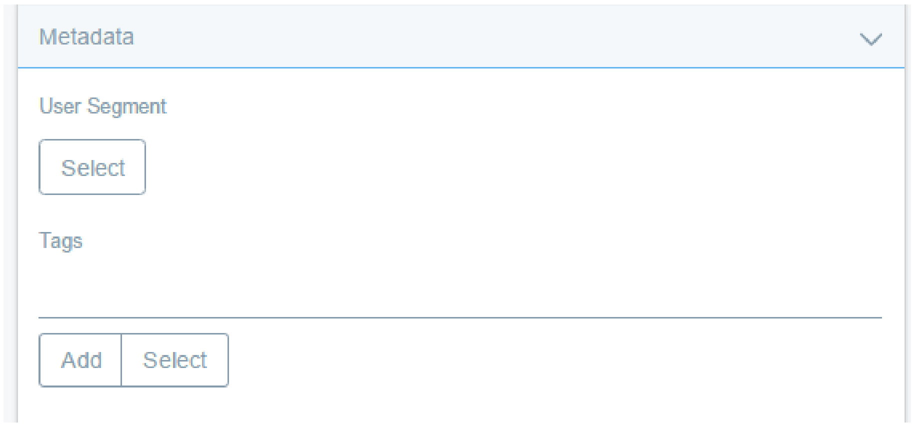


Figure 86.2: Adding category and tag input options lets authors aggregate and label custom entities.

```
<liferay-ui:asset-categories-error />
<liferay-ui:asset-tags-error />
...
<aui:fieldset-group markupView="lexicon">
    ...
    <aui:fieldset collapsed="<%= true %>" collapsible="<%= true %>" label="categorization">
        <aui:input name="categories" type="assetCategories" />

        <aui:input name="tags" type="assetTags" />
    </aui:fieldset>
    ...
</aui:fieldset-group>
```

These category and tag aui:input tags generate form controls that let users browse/select a categories for the entity, browse/select tags, and/or create new tags to associate with the entity.

The liferay-ui:asset-categories-error and liferay-ui:asset-tags-error tags show messages for errors occurring during the asset category or tag input process. The aui:fieldset tag uses a container that lets users hide or show the category and tag input options.

For styling purposes, the `aui:fieldset-group` tag is given the `lexicon` markdup view.

Once the tags and categories have been entered, you'll want to show them along with the content of the asset. Here's how to display the tags and categories:

```
<p><liferay-ui:message key="categories" />:</p>

<div class="entry-categories">
    <liferay-ui:asset-categories-summary
        className="<%= BlogsEntry.class.getName() %>"
        classPK="<%= entry.getEntryId() %>"
        portletURL="<%= renderResponse.createRenderURL() %>"
    />
</div>

...

<div class="entry-tags">
    <p><liferay-ui:message key="tags" />:</p>

    <liferay-ui:asset-tags-summary
        className="<%= BlogsEntry.class.getName() %>"
        classPK="<%= entry.getEntryId() %>"
        portletURL="<%= renderResponse.createRenderURL() %>"
    />
</div>
```

You'll notice the `portletURL` parameter is used for both tags and categories, which supports navigation amongst the two. Each tag that uses this parameter becomes a link containing the `portletURL` *and* `tag` or `categoryId` parameter value. To implement this, you need to implement the look-up functionality in your portlet code. Do this by reading the values of those two parameters and using `AssetEntryService` to query the database for entries based on the specified tag or category.

Deploy your changes and add/edit a custom entity in your UI. Your form shows the categorization and tag input options in a panel that the user can hide/show.

Great! Now you know how to make category and tag input options available to your app's content authors.

Related Topics

Relating Assets

Adding, Updating, and Deleting Assets for Custom Entities

What is Service Builder?

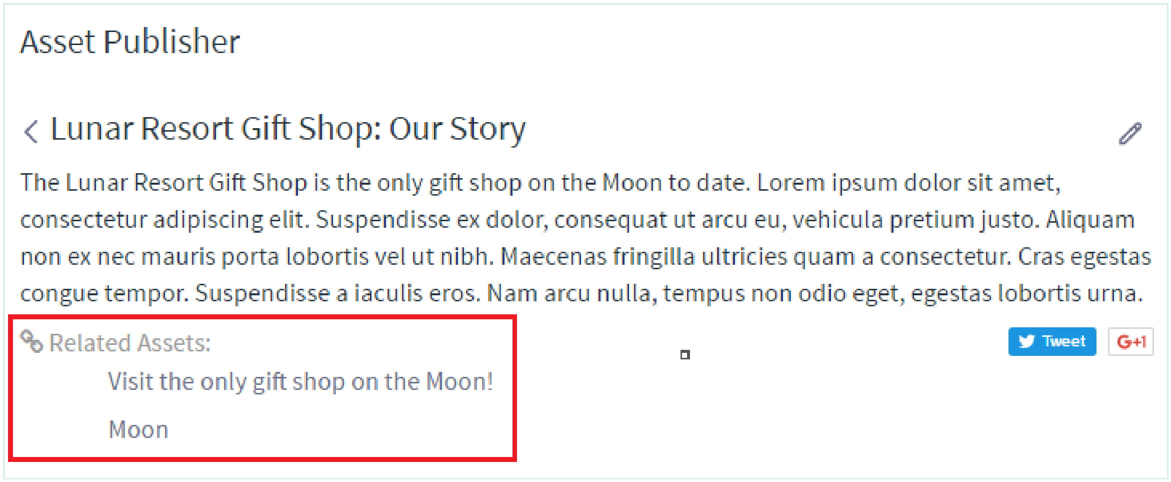
86.4 Relating Assets

The ability to relate assets is one of the most powerful features of Liferay's asset framework. By relating assets, you can connect individual pieces of content across your site or portal. This helps your users discover related content, particularly when there's an abundance of other available content. For example, assets related to a web content article appear alongside that entry in the Asset Publisher application.

This tutorial shows you how to provide a way for authors to relate content. This tutorial assumes that you've asset enabled your appliation. If you've already done this, go ahead and begin relating your assets!

Relating Assets in the Service Layer

First, you must make some modifications to your portlet's service layer. You must implement persisting your entity's asset relationships. In your portlet's `service.xml`, put the following line of code below any finder method elements and then run Service Builder:


Asset Publisher

Lunar Resort Gift Shop: Our Story

The Lunar Resort Gift Shop is the only gift shop on the Moon to date. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse ex dolor, consequat ut arcu eu, vehicula pretium justo. Aliquam non ex nec mauris porta lobortis vel ut nibh. Maecenas fringilla ultricies quam a consectetur. Cras egestas congue tempor. Suspendisse a iaculis eros. Nam arcu nulla, tempus non odio eget, egestas lobortis urna.

Related Assets:
Visit the only gift shop on the Moon!
Moon

Figure 86.3: You and your users can find it helpful to relate assets to entities, such as this blogs entry.

```
<reference package-path="com.liferay.portlet.asset" entity="AssetLink" />
```

Next, you need to modify the add-, delete-, and update- methods in your -LocalServiceImpl to persist the asset relationships. You'll use your -LocalServiceImpl's assetLinkLocalService instance variable to execute persistence actions.

For example, consider the Wiki application. When you update wiki assets and statuses, both methods utilize the updateLinks via your instance variable assetLinkLocalService. Here's the updateLinks invocation in the Wiki application's WikiPageLocalServiceImpl.updateStatus(...) method:

```
assetLinkLocalService.updateLinks(
    userId, assetEntry.getEntryId(), assetLinkEntryIds,
    AssetLinkConstants.TYPE_RELATED);
```

To call the updateLinks method, you need to pass in the current user's ID, the asset entry's ID, the ID's of the asset link entries, and the link type. You should invoke this method after creating the asset entry. You can assign to an AssetEntry variable (e.g., one called assetEntry) the value returned from invoking assetEntryLocalService.updateEntry. That way you can get the asset entry's ID for updating its asset links. Lastly, in order to specify the link type parameter, make sure to import com.liferay.portlet.asset.model.AssetLinkConstants.

In your -LocalServiceImpl class' delete- method, you must delete the asset's relationships before deleting the asset. For example, you could delete your existing asset link relationships by using the following code:

```
AssetEntry assetEntry = assetEntryLocalService.fetchEntry(
    ENTITY.class.getName(), ENTITYId);

assetLinkLocalService.deleteLinks(assetEntry.getEntryId());
```

Make sure to replace the *ENTITY* place holders for your custom -delete method.

Super! Now your portlet's service layer can handle related assets. Even so, there's still nothing in your portlet's UI that lets your users relate assets. You'll take care of that in the next step.

Relating Assets in the UI

You typically implement the UI for linking assets in the JSP that you provide users the ability to create and edit your entity, This way only content creators can relate other assets to the entity. Related assets are implemented in the JSP by using the Liferay UI tag `liferay-ui:input-asset-links` inside of a collapsible panel. This code is placed inside the `aui:fieldset` tags of the JSP. The panel and `liferay-ui:input-asset-links` tag are shown below for the Blogs application:

```
<aui:fieldset collapsed="<%= true %>" collapsible="<%= true %>" label="related-assets">
    <liferay-ui:input-asset-links
        className="<%= BlogsEntry.class.getName() %>"
        classPK="<%= entryId %>"
    />
```

Your content authors are able to relate assets once you add this code and redeploy your portlet.

The following screenshot shows the Related Assets menu for an appliation. Note that it is contained in a collapsible panel titled Related Assets.

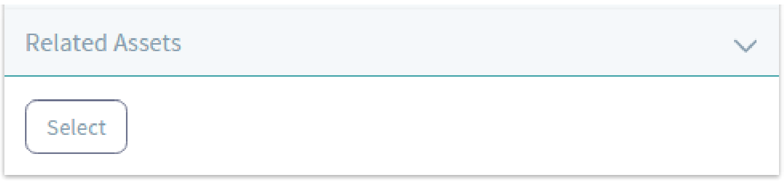


Figure 86.4: Your portlet's entity is now available in the Related Assets *Select* menu.

Even though you've provided a way for authors to assign related assets, the Related Assets menu shows your entity's fully qualified class name, instead of a more concise name. To replace the long fully qualified class name shown in the menu with a simplified name for your entity, add a language key that uses the fully qualified class name as the key's name and the new simplified name as the key's value. Put the language key in file `docroot/WEB-INF/src/content/Language.properties` in your portlet. You can refer to the Overriding Language Keys tutorial for more documentation on using language properties.

Upon redeploying your portlet, the value you assigned to the fully qualified class name in your `Language.properties` file shows in the Related Assets menu:

Awesome! Now content creators and editors can relate the assets of your application. The next thing you need to do is reveal any such related assets to the rest of your application's users. After all, you don't want to give everyone edit access just so they can view related assets!

Showing Related Assets

You can show related assets in your application's view of that entity or, if you've implemented asset rendering for your custom entity, you can show related assets in the full content view of your entity for users to view in an Asset Publisher portlet.

This section shows you how to access an entity's asset entry in your entity's view JSP and how to display links to its related assets. When you finish, users can click on the entity instances in your portlet to view any related assets.

In your entity's view JSP you can use `ParamUtil` to get the ID of the entity from the render request. Then you can create an entity object using your `-LocalServiceUtil` class. You can use an entity instance object to get the `AssetEntry` object associated with it.

```
<%
long insultId = ParamUtil.getLong(renderRequest, "insultId");
```

```
Insult ins = InsultLocalServiceUtil.getInsult(insultId);
AssetEntry assetEntry = AssetEntryLocalServiceUtil.getEntry(Insult.class.getName(), ins.getInsultId());
%>
```

To show the entity's related assets, you can use the `liferay-ui:asset-links` tag. For this tag, you should retrieve the entity's class name and the variable holding your instance object, so you can return its ID. The example code below uses the example entity class `Insult` and an instance object variable called `ins`:

```
<liferay-ui:asset-links
    assetEntryId="<%=(assetEntry != null) ? assetEntry.getEntryId() : 0%>"
    className="<%=Insult.class.getName()%>"
    classPK="<%=ins.getInsultId()%>" />
```

Go ahead and use a the `liferay-ui:asset-links` tag in your JSP. Great! Now you have the JSP that lets your users view related assets.

If you've already connected your portlet's view to the view JSP for your entity, you've completed the tutorial. You can otherwise follow the remainder of this tutorial to learn how to implement that connection.

Creating a URL to Your New JSP

Now that you've implemented showing off this asset feature, you must connect your application's main view JSP to your entity's view JSP. If your main view JSP uses a search container to list your entity instances, you can insert a `portlet:renderURL` tag just after the `liferay-ui:search-container-row` tag. For example, your `view.jsp` could look like this:

```
<liferay-ui:search-container-row
    className="com.sample.portlet.insults.model.Insult"
    keyProperty="insultId"
    modelVar="insult" escapedModel="<%= true %>"
>

<portlet:renderURL windowState="maximized" var="rowURL">
    <portlet:param name="mvcPath" value="/html/insult/view_insult.jsp" />
    <portlet:param name="insultId" value="<%= String.valueOf(insult.getInsultId()) %>" />
</portlet:renderURL>
```

Next, add to the first search container column an `href` attribute with the value of the URL you just created in the `portlet:renderURL` tag. For example, the value of `href` that corresponds with the render URL created above is `"<%=rowURL %>"`. Your `search-container-column-text` tag can look similar to this tag:

```
<liferay-ui:search-container-column-text
    name="Insult"
    value="<%= insult.getInsultString() %>"
    href="<%=rowURL %>"
/>
```

Now, redeploy your portlet and refresh the page so that your portlet's view JSP reloads. Each entity listed is a link. Click on one to view your entity's JSP that you made in the previous step of this tutorial.

Related assets, if you've created any yet, should be visible near the bottom of the page.

Excellent! Now you know how to implement related assets in your apps. Another thing you might want to do is investigate permissioning in the UI. For more information on this, see the tutorial Checking Permissions in the UI.

Related Topics

Adding, Updating, and Deleting Assets for Custom Entities
What is Service Builder?
Defining Content Relationships

86.5 Implementing Asset Priority

The Asset Publisher lets you order assets by priority. For this to work, however, users must be able to set the asset's priority when creating or editing the asset. For example, when creating or editing web content, users can assign a priority in the Metadata section's Priority field.

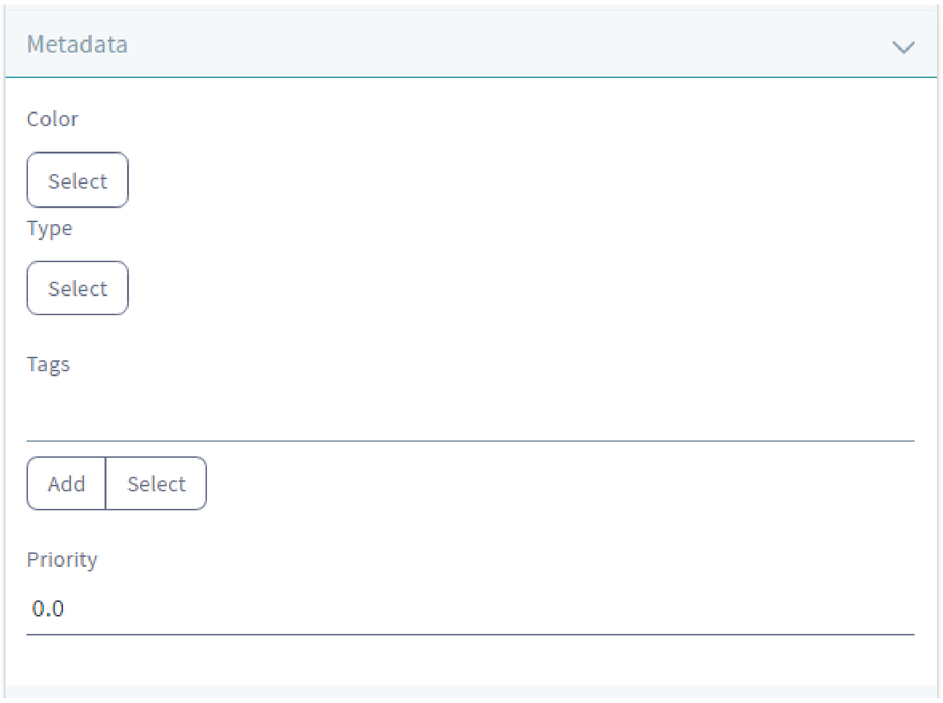


Figure 86.5: The Priority field lets users set an asset's priority.

This field isn't enabled by default for your custom assets. You must manually add support for it. Fortunately, this is very straightforward. This tutorial shows you how. Onwards!

Add the Priority Field to Your JSP

In the JSP for adding and editing your asset, add the following input field that lets users set the asset's priority. This example also validates the input to make sure the value the user sets is a number higher than zero:

```
<aui:input label="priority" name="assetPriority" type="text" value="<%= priority %>">
    <aui:validator name="number" />

    <aui:validator name="min">[0]</aui:validator>
</aui:input>
```

That's it for the view layer! Now when users create or edit your custom asset, they can enter its priority. Next, you'll learn how to use that value in your service layer.

Using the Priority Value in Your Service Layer

To make the priority value functional, you must retrieve it from the view and add it to the asset in your database. The priority value is automatically available in your service layer via the `ServiceContext` variable `serviceContext`. Retrieve it with `serviceContext.getAssetPriority()`, and then pass it as the last argument to the `assetEntryLocalService.updateEntry` call in your `-LocalServiceImpl`. You can see an example of this in the `BlogsEntryLocalServiceImpl` class of Liferay DXP's Blogs app. The `updateAsset` method takes a `priority` argument, which it passes as the last argument to its `assetEntryLocalService.updateEntry` call:

```
@Override
public void updateAsset(
        long userId, BlogsEntry entry, long[] assetCategoryIds,
        String[] assetTagNames, long[] assetLinkEntryIds, Double priority)
    throws PortalException {

    ...

    AssetEntry assetEntry = assetEntryLocalService.updateEntry(
        userId, entry.getGroupId(), entry.getCreateDate(),
        entry.getModifiedDate(), BlogsEntry.class.getName(),
        entry.getEntryId(), entry.getUuid(), 0, assetCategoryIds,
        assetTagNames, true, visible, null, null, null, null,
        ContentTypes.TEXT_HTML, entry.getTitle(), entry.getDescription(),
        summary, null, null, 0, 0, priority);

    ...
}
```

The `BlogsEntryLocalServiceImpl` class calls this `updateAsset` method when adding or updating a blog entry. Note that `serviceContext.getAssetPriority()` retrieves the priority:

```
updateAsset(
        userId, entry, serviceContext.getAssetCategoryIds(),
        serviceContext.getAssetTagNames(),
        serviceContext.getAssetLinkEntryIds(),
        serviceContext.getAssetPriority());
```

Sweet! Now you know how to enable priorities for your app's assets.

Related Topics

Adding, Updating, and Deleting Assets For Custom Entities
Implementing Asset Categorization and Tagging
Relating Assets
Rendering an Asset
Publishing Assets

86.6 Rendering an Asset

There are several options you have for rendering an asset in Liferay DXP. Before setting up the rendering process for your asset, make sure it's added to the asset framework by following the Adding, Updating, and Deleting Assets tutorial. Once you have your asset added to the framework, Liferay DXP can render the asset by default using the Asset Publisher application. The default rendering process Liferay DXP provides, however, only displays the asset's title and description text. Any further rendering of your asset requires additional coding. For instance, you might want these additional things:

- An edit feature for modifying an asset.
- Viewing an asset in its original context (e.g., a blog in the Blogs application; a post in the Message Boards application).
- Embedding images, videos, and audio.
- Restricting access to users who do not have permissions to interact with the asset.
- Allowing users to comment on the asset.

Liferay lets you dictate your asset's rendering capabilities by providing the *Asset Renderer* framework. Implementing an asset renderer for an existing asset is easy because Liferay offers interfaces and factories to help get your custom asset rendering implemented fast. There are two things you must do to get your asset renderer functioning properly for your asset:

- Create an asset renderer for your custom asset.
- Create an asset renderer factory to create an instance of the asset renderer for each asset entity.

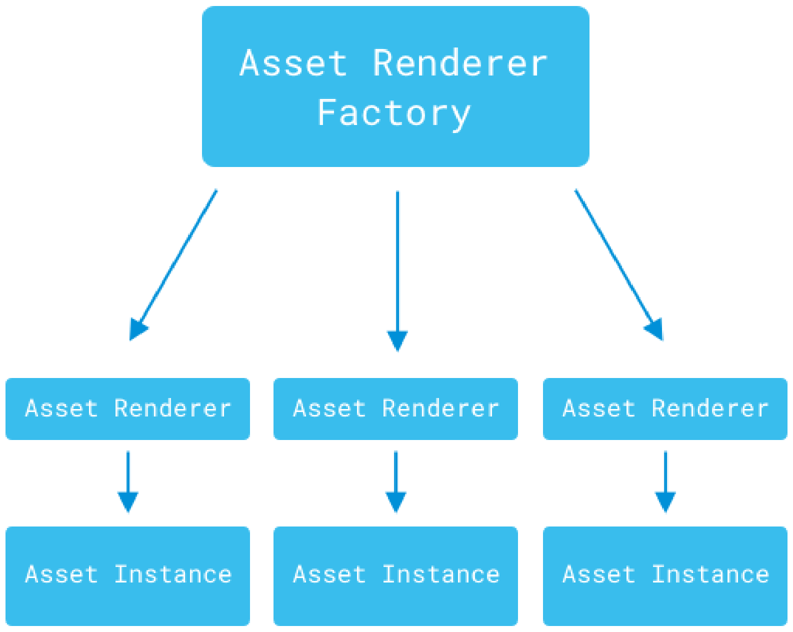


Figure 86.6: The asset renderer factory creates an asset renderer for each asset instance.

You'll learn how to create an asset renderer and an asset renderer factory by studying a Liferay asset that already uses both by default: Blogs. The Blogs application offers many different ways to access and render a blogs asset. You'll learn how a blogs asset provides an edit feature, comment section, original context viewing (i.e., viewing an asset from the Blogs application), workflow, etc. You'll also learn how it uses JSP templates to display various blog views. The Blogs application is an extensive example of how an asset renderer can be customized to fit your needs.

If you want to create an asset and make it do more than display its title and description, read on to learn more!

Prerequisites for Asset Enabling and Application

To asset-enable your application, you need two things:

1. The application must store asset data. Applications that store a data model meet this requirement.

2. The application must contain at least one non-instanceable portlet. `Edit` links for the asset cannot be generated without a non-instanceable portlet.

Some applications may consist of only one non-instanceable portlet, while others may consist of a both instanceable and non-instanceable portlets. If your application does not currently include a non-instanceable portlet, adding a configuration interface through a panel app will both enhance the usability of the application, and meet the requirement for adding a non-instanceable component to the application. See our tutorial on Adding Custom Panel Apps to learn how to add one.

Now that you have all that taken care of, you can move on to creating an Asset Renderer.

Creating an Asset Renderer

An asset renderer lets you provide your own HTML for your asset. The `AssetRenderer` interface requires that you choose a templating technology (JSP, FreeMarker, Soy, etc.) to display an asset's HTML. For this tutorial, you'll use JSP templates to render an asset's HTML. You'll learn how to associate your JSP templates with an asset renderer, along with configuring several other options.

To learn how an asset renderer is created, you'll create the pre-existing BlogsEntryAssetRenderer class, which configures the asset renderer framework for the Blogs application.

1. Create a new package in your existing project for your asset-related classes. For instance, the `BlogsEntryAssetRenderer` class resides in the `com.liferay.blogs.web` module's `com.liferay.blogs.web.asset` package.

2. Create your `-AssetEntry` class for your application in the new `-.asset` package and have it implement the AssetEntry interface. Consider the `BlogsEntryAssetRenderer` class as an example:

```
public class BlogsEntryAssetRenderer
    extends BaseJSPAssetRenderer<BlogsEntry> implements TrashRenderer {
```

   The `BlogsEntryAssetRenderer` class extends the BaseJSPAssetRenderer, which is an extension class intended for those who plan on using JSP templates to generate their asset's HTML. The `BaseJSPAssetRenderer` class implements the `AssetRenderer` interface. You'll notice the asset renderer is also implementing the TrashRenderer interface. This is a common practice for many Liferay DXP applications, which enables its assets to use Liferay DXP's Recycle Bin.

3. Define the asset renderer class's constructor, which typically sets the asset object to use in the asset renderer class.

```
public BlogsEntryAssetRenderer(
    BlogsEntry entry, ResourceBundleLoader resourceBundleLoader) {

    _entry = entry;
    _resourceBundleLoader = resourceBundleLoader;
}
```

The `BlogsEntryAssetRenderer` also sets the resource bundle loader, which loads the language keys for a module. You can learn more about the resource bundle loader in the Overriding Language Keys tutorial.

Also, make sure to define the `_entry` and `_resourceBundleLoader` fields in the class:

```
private final BlogsEntry _entry;
private final ResourceBundleLoader _resourceBundleLoader;
```

4. Now that your class declaration and constructor are defined for the blogs asset renderer, you must begin connecting your asset renderer to your asset. The following getter methods accomplish this:

```
@Override
public BlogsEntry getAssetObject() {
    return _entry;
}

@Override
public String getClassName() {
    return BlogsEntry.class.getName();
}

@Override
public long getClassPK() {
    return _entry.getEntryId();
}

@Override
public long getGroupId() {
    return _entry.getGroupId();
}

@Override
public String getType() {
    return BlogsEntryAssetRendererFactory.TYPE;
}

@Override
public String getUuid() {
    return _entry.getUuid();
}
```

These methods are pretty self-explanatory, but there are a couple things to point out. The `getAssetObject()` method sets the `BlogsEntry` that was set in the constructor as your asset to track. Likewise, the `getType()` method references the blogs asset renderer factory for the type of asset your asset renderer renders. Of course, the asset renderer type is `blog`, which you'll set in the factory later.

5. Your asset renderer must link to the portlet that owns the entity. In the case of a blogs asset, its portlet ID should be linked to the Blogs application.

```
@Override
public String getPortletId() {
    AssetRendererFactory<BlogsEntry> assetRendererFactory =
        getAssetRendererFactory();

    return assetRendererFactory.getPortletId();
}
```

The `getPortletId()` method instantiates an asset renderer factory for a `BlogsEntry` and retrieves the portlet ID for the portlet used to display blogs entries.

6. If you're interested in enabling workflow for your asset, add the following method similar to what was done for the Blogs application:

   ```
   @Override
   public int getStatus() {
       return _entry.getStatus();
   }
   ```

   This method retrieves the workflow status for the asset.

7. Another popular feature many developers want for their asset is to comment on it. This is enabled for the Blogs application with the following method:

   ```
   @Override
   public String getDiscussionPath() {
       if (PropsValues.BLOGS_ENTRY_COMMENTS_ENABLED) {
           return "edit_entry_discussion";
       }
       else {
           return null;
       }
   }
   ```

   A comments section is an available option if it returns a non-null value. A JSP template defining a comments section is not required. For the comments section to display for your asset, you must enable it in the Asset Publisher's *Options* (⋮) → *Configuration* → *Setup* → *Display Settings* section.

8. At a minimum, you should create a title and summary for your asset. Here's how the `BlogsEntryAssetRenderer` does it:

   ```
   @Override
   public String getSummary(
       PortletRequest portletRequest, PortletResponse portletResponse) {

       int abstractLength = AssetUtil.ASSET_ENTRY_ABSTRACT_LENGTH;

       if (portletRequest ≠ null) {
           abstractLength = GetterUtil.getInteger(
               portletRequest.getAttribute(
                   WebKeys.ASSET_ENTRY_ABSTRACT_LENGTH),
               AssetUtil.ASSET_ENTRY_ABSTRACT_LENGTH);
       }

       String summary = _entry.getDescription();

       if (Validator.isNull(summary)) {
           summary = HtmlUtil.stripHtml(
               StringUtil.shorten(_entry.getContent(), abstractLength));
       }

       return summary;
   }

   @Override
   public String getTitle(Locale locale) {
       ResourceBundle resourceBundle =
           _resourceBundleLoader.loadResourceBundle(
               LanguageUtil.getLanguageId(locale));

       return BlogsEntryUtil.getDisplayTitle(resourceBundle, _entry);
   }
   ```

These two methods return information about your asset in a generic way, so the asset publisher can display it. Anything appropriate for your asset can be the title or the summary.

The `getSummary(...)` method for Blogs returns the abstract description for a blog asset. If the abstract description does not exist, the content of the blog is used as an abstract. You'll learn more about abstracts and other content specifications later.

The `getTitle(...)` method for Blogs uses the resource bundle loader you configured in the constructor to load your module's resource bundle and return the display title for your asset.

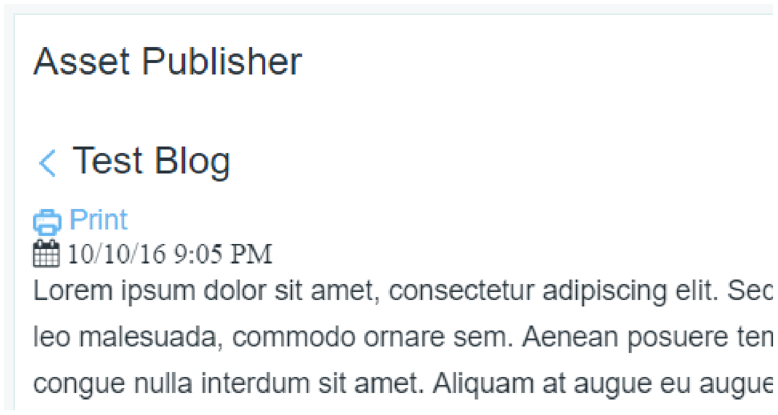
9. If you want to provide a unique URL for your asset, you can specify a URL title. A URL title is the URL used to access your asset directly (e.g., localhost:8080/-/this-is-my-blog-asset). You can do this by providing the following method:

```
@Override
public String getUrlTitle() {
    return _entry.getUrlTitle();
}
```

10. Insert the `isPrintable()` method, which enables the Asset Publisher's printing capability for your asset.

```
@Override
public boolean isPrintable() {
    return true;
}
```

This displays a Print icon when your asset is displayed in the Asset Publisher. For the icon to appear, you must enable it in the Asset Publisher's *Options* (⋮) → *Configuration* → *Setup* → *Display Settings* section.



Asset Publisher

Test Blog

Print
10/10/16 9:05 PM
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sec
leo malesuada, commodo ornare sem. Aenean posuere ten
congue nulla interdum sit amet. Aliquam at augue eu augue

Figure 86.7: Enable printing in the Asset Publisher to display the Print icon for your asset.

11. If your asset is protected by permissions, you should set permissions for the asset. You can do this via the asset renderer as well. See the logic below for an example used in the `BlogsEntryAssetRenderer` class:

```
@Override
public long getUserId() {
    return _entry.getUserId();
}
```

```
    @Override
    public String getUserName() {
        return _entry.getUserName();
    }

    public boolean hasDeletePermission(PermissionChecker permissionChecker) {
        return BlogsEntryPermission.contains(
            permissionChecker, _entry, ActionKeys.DELETE);
    }

    @Override
    public boolean hasEditPermission(PermissionChecker permissionChecker) {
        return BlogsEntryPermission.contains(
            permissionChecker, _entry, ActionKeys.UPDATE);
    }

    @Override
    public boolean hasViewPermission(PermissionChecker permissionChecker) {
        return BlogsEntryPermission.contains(
            permissionChecker, _entry, ActionKeys.VIEW);
    }
```

Before you can check if a user has permission to view your asset, you must distinguish the user. The `getUserId()` and `getUserName()` retrieves the entry's user ID and username, respectively. Then there are three boolean permission methods, which check if the user can view, edit, or delete your blogs entry. These permissions are for specific entity instances. Global permissions for blog entries are implemented in the factory, which you'll do later.

Awesome! You've learned how to set up the blogs asset renderer to

- connect to an asset
- connect to the asset's portlet
- use workflow management
- use a comments section
- retrieve the asset's title and summary
- generate the asset's unique URL
- display a print icon
- set up permissioning for the asset

You may recall that a major cog in asset renderer development generating HTML using a templating technology. The `BlogsEntryAssetRenderer` is configured to use JSP templates to generate HTML for the Asset Publisher. You'll learn more about how to do this next.

Configuring JSP Templates for an Asset Renderer

An asset can be displayed in several different ways in the Asset Publisher, by default. There are three templates to implement provided by the `AssetRenderer` interface:

- `abstract`
- `full_content`
- `preview`

Besides these supported templates, you can also create JSPs for buttons you'd like to provide for direct access and manipulation of the asset. For example,

- Edit
- View
- View in Context

The BlogsEntryAssetRenderer customizes the AssetRenderer's provided JSP templates and adds a few other features using JSPs. You'll inspect how the blogs asset renderer is put together to satisfy JSP template development requirements.

1. Add the getJspPath(...) method to your asset renderer. This method should return the path to your JSP, which is rendered inside the Asset Publisher. This is how the BlogsEntryAssetRenderer uses this method:

   ```
   @Override
   public String getJspPath(HttpServletRequest request, String template) {
       if (template.equals(TEMPLATE_ABSTRACT) ||
           template.equals(TEMPLATE_FULL_CONTENT)) {

           return "/blogs/asset/" + template + ".jsp";
       }
       else {
           return null;
       }
   }
   ```

   Blogs assets provide abstract.jsp and full_content.jsp templates. This means that a blogs asset can render a blog's abstract description or the blog's full content in the Asset Publisher. Those templates are located in the com.liferay.blogs.web module's src/main/resources/META-INF/resources/blogs/asset folder. You could create a similar folder for your JSP templates used for this method. The other template provided by the AssetRenderer interface, preview.jsp, is not customized by the blogs asset renderer, so its default template is implemented.

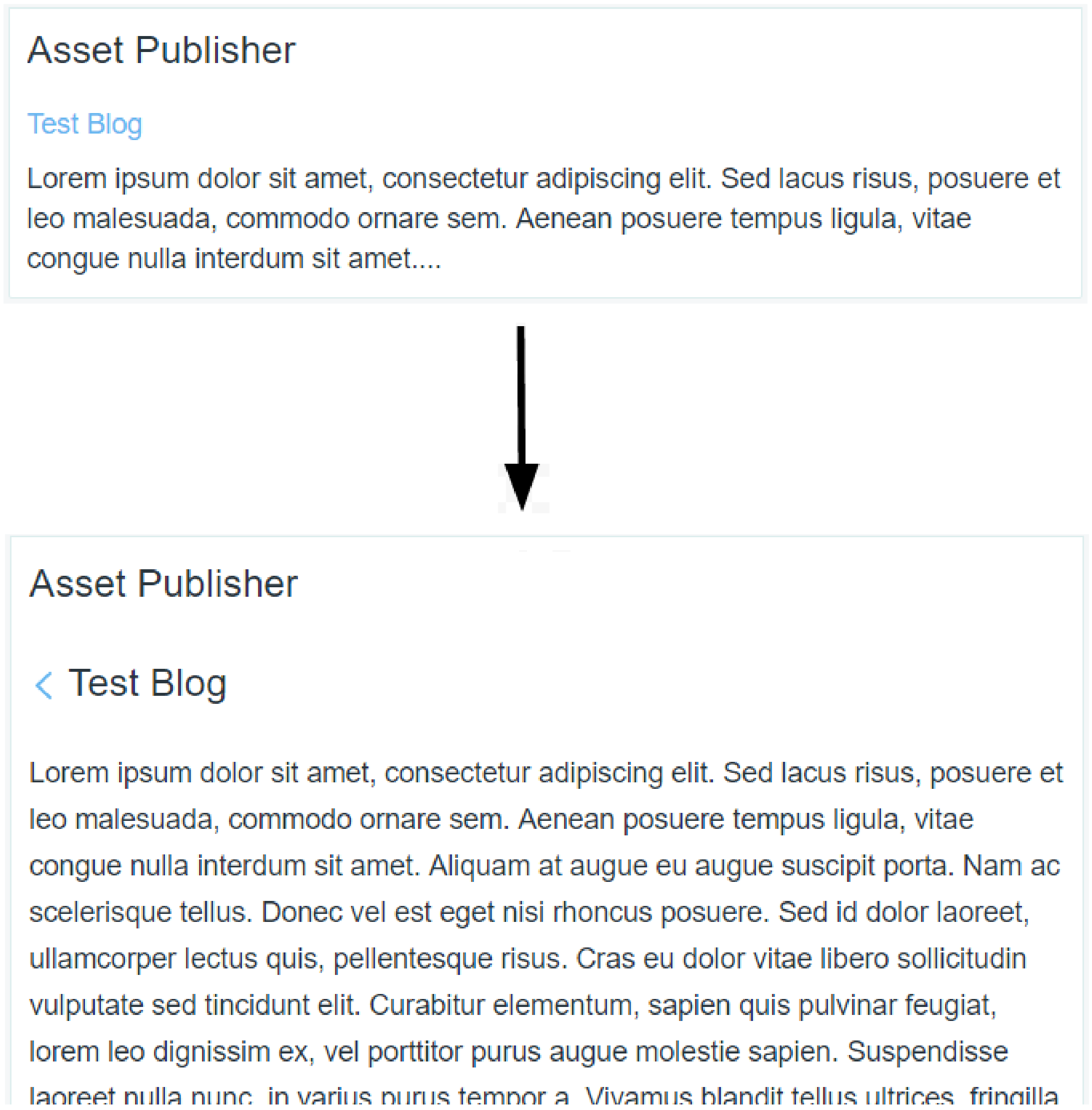
   You must create a link to display the full content of the asset. You'll do this later.

2. Now that you've added the path to your JSP, you must include that JSP. Since the BlogsEntryAssetRenderer class extends the BaseJSPAssetRenderer, it already has an include(...) method to render a specific JSP. You must override this method to set an attribute in the request to use in the blog's views:

   ```
   @Override
   public boolean include(
           HttpServletRequest request, HttpServletResponse response,
           String template)
       throws Exception {

       request.setAttribute(WebKeys.BLOGS_ENTRY, _entry);

       return super.include(request, response, template);
   }
   ```

   The attribute includes the blogs entry object. Adding the blog object this way is not mandatory; you could obtain the blog entry directly from the view. Using the include(...) method, however, follows the best practice for MVC portlets.

Terrific! You've learned how to apply JSPs supported by the Asset Publisher for your asset. That's not all you can do with JSP templates, however! The asset renderer framework provides several other methods that let you render convenient buttons for your asset.


Asset Publisher

Test Blog

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed lacus risus, posuere et leo malesuada, commodo ornare sem. Aenean posuere tempus ligula, vitae congue nulla interdum sit amet....

Asset Publisher

Test Blog

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed lacus risus, posuere et leo malesuada, commodo ornare sem. Aenean posuere tempus ligula, vitae congue nulla interdum sit amet. Aliquam at augue eu augue suscipit porta. Nam ac scelerisque tellus. Donec vel est eget nisi rhoncus posuere. Sed id dolor laoreet, ullamcorper lectus quis, pellentesque risus. Cras eu dolor vitae libero sollicitudin vulputate sed tincidunt elit. Curabitur elementum, sapien quis pulvinar feugiat, lorem leo dignissim ex, vel porttitor purus augue molestie sapien. Suspendisse laoreet nulla nunc, in varius purus tempor a. Vivamus blandit tellus ultrices, fringilla

Figure 86.8: The abstract and full content views are rendererd differently for blogs.

1. Blogs assets provide an Edit button ( ) that lets you edit the asset. This is provided by adding the following method to the `BlogsEntryAssetRenderer` class:

```
@Override
public PortletURL getURLEdit(
        LiferayPortletRequest liferayPortletRequest,
        LiferayPortletResponse liferayPortletResponse)
    throws Exception {

    Group group = GroupLocalServiceUtil.fetchGroup(_entry.getGroupId());

    PortletURL portletURL = PortalUtil.getControlPanelPortletURL(
        liferayPortletRequest, group, BlogsPortletKeys.BLOGS, 0, 0,
        PortletRequest.RENDER_PHASE);

    portletURL.setParameter("mvcRenderCommandName", "/blogs/edit_entry");
    portletURL.setParameter("entryId", String.valueOf(_entry.getEntryId()));
    return portletURL;
}
```

The Asset Publisher loads the blogs asset using the Blogs application. Then the `edit_entry.jsp` template generates the HTML for an editing UI. Once the necessary edits are made to the asset, it can be saved from the Asset Publisher. Pretty cool, right?

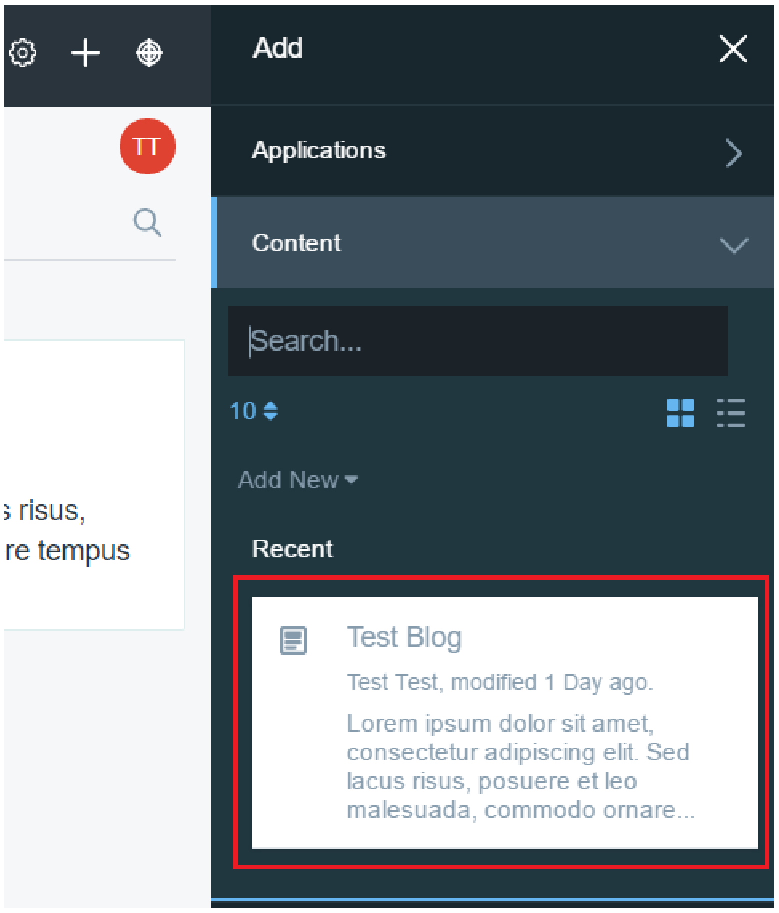
2. You can specify how to view your asset by providing methods similar to the methods outlined below in the `BlogsEntryAssetRenderer` class:

```
@Override
public String getURLView(
        LiferayPortletResponse liferayPortletResponse,
        WindowState windowState)
    throws Exception {

    AssetRendererFactory<BlogsEntry> assetRendererFactory =
        getAssetRendererFactory();

    PortletURL portletURL = assetRendererFactory.getURLView(
        liferayPortletResponse, windowState);

    portletURL.setParameter("mvcRenderCommandName", "/blogs/view_entry");
    portletURL.setParameter("entryId", String.valueOf(_entry.getEntryId()));
    portletURL.setWindowState(windowState);

    return portletURL.toString();
}

@Override
public String getURLViewInContext(
    LiferayPortletRequest liferayPortletRequest,
    LiferayPortletResponse liferayPortletResponse,
    String noSuchEntryRedirect) {

    return getURLViewInContext(
        liferayPortletRequest, noSuchEntryRedirect, "/blogs/find_entry",
        "entryId", _entry.getEntryId());
}
```

The `getURLView(...)` method generates a URL that displays the full content of the asset in the Asset Publisher. This is assigned to the clickable asset name. The `getURLViewInContext(...)` method provides a similar URL assigned to the asset name, but the URL redirects to the original context of the asset (e.g., viewing a blogs asset in the Blogs application). Deciding which view to render in Liferay DXP is configurable by navigating to the Asset Publisher's *Options* (⋮) → *Configuration* → *Setup* → *Display Settings* section and choosing between *Show Full Content* and *View in Context* for the Asset Link Behavior drop-down menu.

The Blogs application provides `abstract` and `full_content` JSP templates that override the ones provided by the `AssetRenderer` interface. The third template, `preview`, could also be customized. You can view the default `preview.jsp` template rendered in the *Add* (➕) → *Content* menu.

You've learned all about implementing the `AssetRenderer`'s provided templates and customizing them to fit your needs. Next, you'll put your asset renderer into action by creating a factory.



Test Blog

Test Test, modified 1 Day ago.

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed lacus risus, posuere et leo malesuada, commodo ornare...

Figure 86.9: The `preview` template displays a preview of the asset in the Content section of the Add menu.

Creating a Factory for the Asset Renderer

You've successfully created an asset renderer, but you must create a factory class to generate asset renderers for each asset instance. For example, the blogs asset renderer factory instantiates `BlogsEntryAssetRenderer` for each blogs asset displayed in an Asset Publisher.

You'll continue the blogs asset renderer example by creating the blogs asset renderer factory.

1. Create an `-AssetRenderFactory` class in the same folder as its asset renderer class. For blogs, the `BlogsEntryAssetRendererFactory` class resides in the `com.liferay.blogs.web` module's `com.liferay.blogs.web.asset` package. The factory class should extend the `BaseAssetRendererFactory` class and the asset type should be specified as its parameter. You can see how this was done in the `BlogsEntryAssetRendererFactory` class below

```
public class BlogsEntryAssetRendererFactory
    extends BaseAssetRendererFactory<BlogsEntry> {
```

2. Create an `@Component` annotation section above the class declaration. This annotation is responsible for registering the factory instance for the asset.

```
@Component(
    immediate = true,
    property = {"javax.portlet.name=" + BlogsPortletKeys.BLOGS},
    service = AssetRendererFactory.class
)
public class BlogsEntryAssetRendererFactory
    extends BaseAssetRendererFactory<BlogsEntry> {
```

There are a few annotation elements you should set:

- The `immediate` element directs the factory to start in Liferay DXP when its module starts.
- The `property` element sets the portlet that is associated with the asset. The Blogs portlet is specified, since this is the Blogs asset renderer factory.
- The `service` element should point to the `AssetRendererFactory.class` interface.

```
**Note:** In previous versions of Liferay DXP, you had to register the asset
renderer factory in a portlet's `liferay-portlet.xml` file. The registration
process is now completed automatically by OSGi using the `@Component`
annotation.
```

---

3. Create a constructor for the factory class that presets private attributes of the factory.

```
public BlogsEntryAssetRendererFactory() {
    setClassName(BlogsEntry.class.getName());
    setCategorizable(true);
    setLinkable(true);
    setPortletId(BlogsPortletKeys.BLOGS);
    setSearchable(true);
    setSelectable(true);
}
```

   - *linkable*: other assets can select blogs assets as their related assets.
   - *categorizable*: blogs can be used to delimit the scope of a vocabulary from the Categories Administration.
   - *searchable*: blogs can be found when searching for assets.
   - *selectable*: blogs can be selected when choosing assets to display in the Asset Publisher.

   Setting the class name and portlet ID links the asset renderer factory to the entity.

4. Create the asset renderer for your asset. This is done by calling its constructor.

```
@Override
public AssetRenderer<BlogsEntry> getAssetRenderer(long classPK, int type)
    throws PortalException {

    BlogsEntry entry = _blogsEntryLocalService.getEntry(classPK);

    BlogsEntryAssetRenderer blogsEntryAssetRenderer =
        new BlogsEntryAssetRenderer(entry, _resourceBundleLoader);

    blogsEntryAssetRenderer.setAssetRendererType(type);
    blogsEntryAssetRenderer.setServletContext(_servletContext);

    return blogsEntryAssetRenderer;
}
```

For blogs, the asset is retrieved by calling the Blogs application's local service. Then the asset renderer is instantiated using the blogs asset and resource bundle loader. Next, the type and servlet context is set for the asset renderer. Finally, the configured asset renderer is returned.

There are a few variables in the `getAssetRenderer(...)` method you must create. You'll set those variables and learn what they're doing next.

a. You must get the entry by calling the Blogs application's local service. You can instantiate this service by creating a private field and setting it using a setter method:

   @Reference(unbind = "-") protected void setBlogsEntryLocalService(BlogsEntryLocalService blogsEntryLocalService) {

   ```
    _blogsEntryLocalService = blogsEntryLocalService;
   ```

   }

   private BlogsEntryLocalService _blogsEntryLocalService;

The setter method is annotated with the `@Reference` tag. Visit the Invoking Liferay Services Locally section of the *Finding and Invoking Liferay Services* tutorial for more information.

b. You must specify the resource bundle loader since it was specified in the `BlogsEntryAssetRenderer`'s constructor:

   @Reference(target = "(bundle.symbolic.name=com.liferay.blogs.web)", unbind = "-") public void setResourceBundleLoader(ResourceBundleLoader resourceBundleLoader) {

   ```
    _resourceBundleLoader = resourceBundleLoader;
   ```

   }

   private ResourceBundleLoader _resourceBundleLoader;

Make sure the `osgi.web.symbolicname` in the `target` property of the `@Reference` annotation is set to the same value as the `Bundle-SymbolicName` defined in the `bnd.bnd` file of the module the factory resides in.

c. The asset renderer `type` integer is set for the asset renderer, but why an integer? Liferay DXP needs to differentiate when it should display the latest *approved* version of the asset, or the latest version, even if it's unapproved (e.g., unapproved versions would be displayed for reviewers of the asset in a workflow). For these situations, the asset renderer factory should receive either

   - `0` for the latest version of the asset
   - `1` for the latest approved version of the asset

d. Since the Blogs application provides its own JSPs, it must pass a reference of the servlet context to the asset renderer. This is always required when using custom JSPs in an asset renderer:

   @Reference(target = "(osgi.web.symbolicname=com.liferay.blogs.web)", unbind = "-") public void setServletContext(ServletContext servletContext) { _servletContext = servletContext; }

   private ServletContext _servletContext;

5. Set the type of asset that the asset factory associates with and provide a getter method to retrieve that type. Also, provide another getter to retrieve the blogs entry class name, which is required:

   ```
   public static final String TYPE = "blog";

   @Override
   public String getType() {
       return TYPE;
   }

   @Override
   public String getClassName() {
       return BlogsEntry.class.getName();
   }
   ```

6. Set the Lexicon icon for the asset:

   ```
   @Override
   public String getIconCssClass() {
       return "blogs";
   }
   ```

   You can find a list of all available Lexicon icons at https://liferay.github.io/clay/content/icons-lexicon/.

7. Add methods that generate URLs to add and view the asset.

   ```
   @Override
   public PortletURL getURLAdd(
       LiferayPortletRequest liferayPortletRequest,
       LiferayPortletResponse liferayPortletResponse, long classTypeId) {

       PortletURL portletURL = PortalUtil.getControlPanelPortletURL(
           liferayPortletRequest, getGroup(liferayPortletRequest),
           BlogsPortletKeys.BLOGS, 0, 0, PortletRequest.RENDER_PHASE);

       portletURL.setParameter("mvcRenderCommandName", "/blogs/edit_entry");

       return portletURL;
   }

   @Override
   public PortletURL getURLView(
       LiferayPortletResponse liferayPortletResponse,
       WindowState windowState) {

       LiferayPortletURL liferayPortletURL =
           liferayPortletResponse.createLiferayPortletURL(
               BlogsPortletKeys.BLOGS, PortletRequest.RENDER_PHASE);

       try {
           liferayPortletURL.setWindowState(windowState);
       }
       catch (WindowStateException wse) {
       }

       return liferayPortletURL;
   }
   ```

   If you're paying close attention, you may have noticed the `getURLView(...)` method was also implemented in the `BlogsEntryAssetRenderer` class. The asset renderer's `getURLView(...)` method creates a URL for the specific asset instance, whereas the factory uses the method to create a generic URL that only points to the application managing the assets (e.g., Blogs application).

8. Set the global permissions for all blogs assets:

```
@Override
public boolean hasAddPermission(
        PermissionChecker permissionChecker, long groupId, long classTypeId)
    throws Exception {

    return BlogsPermission.contains(
        permissionChecker, groupId, ActionKeys.ADD_ENTRY);
}

@Override
public boolean hasPermission(
        PermissionChecker permissionChecker, long classPK, String actionId)
    throws Exception {

    return BlogsEntryPermission.contains(
        permissionChecker, classPK, actionId);
}
```

Great! You've finished creating the Blogs application's asset renderer factory! Now you have the knowledge to implement an asset renderer and produce an asset renderer for each asset instance using a factory!

Chapter 87

Liferay's Workflow Framework

Enabling your application's entities to support workflow is so easy, you could do it in your sleep (but don't try). Workflow enabled entities require a few things:

- A workflow handler class to interact with Liferay's workflow back end and the entity's service layer.
- Some extra fields in their database table that help keep track of their status.

  In most Liferay applications, Service Builder will be used to create those fields.
- Updates to the service layer.

  The service layer needs code to populate the new fields when entities are added to the database.

  The service layer needs to send the entity through Liferay's workflow, and it needs to handle the workflow status of the entity when it's returned by the workflow.

  The service layer needs getters that return entities of the desired workflow status (usually *approved*).
- The View layer should account for the workflow status of displayed entities.

The code for most Liferay applications spans multiple modules, so where should you implement the workflow handler? It should go in the module with your service implementations. It's nice to keep your back end code separate from your view layer and controller (in the MVC pattern).

87.1 Creating a Workflow Handler

First create a Component class. It should extend `BaseWorkflowHandler<T>`, an abstract class that provides a default implementation of the `WorkflowHandler<T>` service. Pass the interface for your model as the type parameter for the class.

```
FooEntity WorkflowHandler extends BaseWorkflowHandler<FooEntity>
```

Since you're publishing a service to be consumed in the OSGi runtime, your workflow handler class needs to be registered. If you're using Declarative Services, make it a Component class, using the `@Component` annotation.

```
@Component(
    property = {"model.class.name=com.my.app.package.model.FooEntity"},
    service = WorkflowHandler.class
)
```

It needs one property, to set `model.class.name` to the fully qualified class name of the class you passed as the type parameter. It also needs to declare the type of service being implemented (`WorkflowHandler.class`).

What methods do you need to override in your workflow handler? Just three:

```
@Override
public String getClassName() {

@Override
public String getType(Locale locale) {

@Override
public FooEntity updateStatus(int status, Map<String, Serializable> workflowContext) {
```

The first two are pretty boilerplate. Most of the heavy lifting is being done in the `updateStatus` method. It returns a call to a local service method of the same name, so the status returned from the workflow back end can be persisted to the entity table in the database.

The `updateStatus` method should take a user ID, the primary key for the class (for example, `fooEntityId`), the workflow status, the service context, and the workflow context. The status and the workflow context can be obtained from the workflow back end. You'll need to define the rest of the parameters, which can be obtained from the workflow context.

```
@Override
public FooEntity updateStatus(
        int status, Map<String, Serializable> workflowContext)
    throws PortalException {

    long userId = GetterUtil.getLong(
        (String)workflowContext.get(WorkflowConstants.CONTEXT_USER_ID));
    long classPK = GetterUtil.getLong(
        (String)workflowContext.get(
            WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));

    ServiceContext serviceContext = (ServiceContext)workflowContext.get(
        "serviceContext");

    return _fooEntityLocalService.updateStatus(
        userId, classPK, status, serviceContext, workflowContext);
}
```

Now your entity can be handled by Liferay's workflow framework. Next, update the service methods to account for workflow status, and add a new method to update the status of an entity in the database.

87.2 Updating the Service Layer

Make sure your entity database table has `status`, `statusByUserId`, `statusByUserName`, and `statusDate` fields. If you're using service builder, add this to your `service.xml` if you haven't already:

```
<column name="status" type="int" />
<column name="statusByUserId" type="long" />
<column name="statusByUserName" type="String" />
<column name="statusDate" type="Date" />
```

Wherever you're setting the other database fields in your persistence code, set the workflow status as a draft and set the other fields.

```
fooEntity.setStatus(WorkflowConstants.STATUS_DRAFT);
fooEntity.setStatusByUserId(userId);
fooEntity.setStatusByUserName(user.getFullName());
fooEntity.setStatusDate(serviceContext.getModifiedDate(null));
```

With Service Builder driven Liferay applications, this will be in the local service implementation class (`-LocalServiceImpl`).

Whenever an entity is added to the database you need to detect whether workflow is installed and active. If not, you need to automatically mark the entity as approved so it appears in the UI. If it is, you want to leave it in draft status and send it to the workflow back end where it can be properly handled. Thankfully, this whole process is easily done with a single call to `WorkflowHandlerRegistryUtil.startWorkflowInstance`. There are several methods of this name which take a different parameter set, so inspect the `WorkflowHandlerRegistryUtil` class and decide which is right for your case.

```
WorkflowHandlerRegistryUtil.startWorkflowInstance(fooEntity.getCompanyId(),
        fooEntity.getGroupId(), fooEntity.getUserId(), FooEntity.class.getName(),
        fooEntity.getPrimaryKey(), fooEntity, serviceContext);
```

Once you've set the database fields for workflow status and started the workflow instance, implement the `updateStatus` method that you need to call in the workflow handler. The workflow handler gets the entity's status from the workflow back end and passes it to your service layer, which persists the updated entity to the database.

```
fooEntity.setStatus(status);
fooEntity.setStatusByUserId(user.getUserId());
fooEntity.setStatusByUserName(user.getFullName());
fooEntity.setStatusDate(serviceContext.getModifiedDate(now));

fooEntityPersistence.update(fooEntity);
```

After setting the workflow fields for the entity, think about the specifics of your situation and whether any additional logic should be added to this method. For instance, if your entities are Liferay Assets already, you'll want to change the visibility of the asset depending on its workflow status. You don't want the Asset Publisher displaying entities that haven't yet been approved in the workflow process.

```
if (status == WorkflowConstants.STATUS_APPROVED) {

    assetEntryLocalService.updateEntry(
        FooEntity.class.getName(), fooEntityId, fooEntity.getDisplayDate(),
        null, true, true);
}

else {

    assetEntryLocalService.updateVisible(
        fooEntity.class.getName(), entryId, false);
}
```

If approved, the asset is updated, with the publication date, a `listable` boolean, and a `visible` boolean being updated to reflect the current state of the asset. If the workflow status is anything other than approved, its visibility is set to `false`.

For an example of a fully implemented `updateStatus` method, see the `com.liferay.portlet.blogs.service.impl.BlogsEntry` class in `portal-impl`.

Before leaving the service layer, add a call to `deleteWorkflowInstanceLinks` in the `deleteEntity` method. Here's what it looks like:

```
workflowInstanceLinkLocalService.deleteWorkflowInstanceLinks(
    fooEntity.getCompanyId(), fooEntity.getGroupId(),
    FooEntity.class.getName(), fooEntity.getFooEntityId());
```

When you send an entity to the workflow framework via the `startWorkflowInstance` call, it creates an entry in the `workflowinstancelink` database table. This `delete` call ensures there are no orphaned entries in the `workflowinstancelinks` table.

Note, to get the `WorkflowInstanceLocalService` injected into your `*LocalServiceBaseImpl` so you can call its methods in the `LocalServiceImpl`, add this to your entity declaration in `service.xml`:

```
    <reference entity="WorkflowInstanceLink" package-path="com.liferay.portal" />
```

Save your work and run Service Builder. Once you've accounted for workflow status in your service layer, there's only one thing left to do: update the user interface.

87.3 Workflow Status and the View Layer

If you have an application with database entities, you're likely displaying them. If you're sending entities through a workflow process, you only want to display approved entities to your end users.

This often involves the following steps:

- Create a *finder* for your entities that accounts for the `status` field in your database table.
- Expose the finder in a *getter* method of your service layer.
- Update the view layer to use the new getter for displaying entities (e.g., in a Search Container).

If you're using Service Builder, define your finder in your application's `service.xml` and let Service Builder generate it for you.

```
<finder name="G_S" return-type="Collection">
    <finder-column name="groupId"></finder-column>
    <finder-column name="status"></finder-column>
</finder>
```

Then make sure you have a getter in your service layer that uses the new finder.

```
public List<FooEntity> getFooEntities(long groupId, int status)
       throws SystemException {
    return fooEntityPersistence.findByG_S(groupId,
         WorkflowConstants.STATUS_APPROVED);
}
```

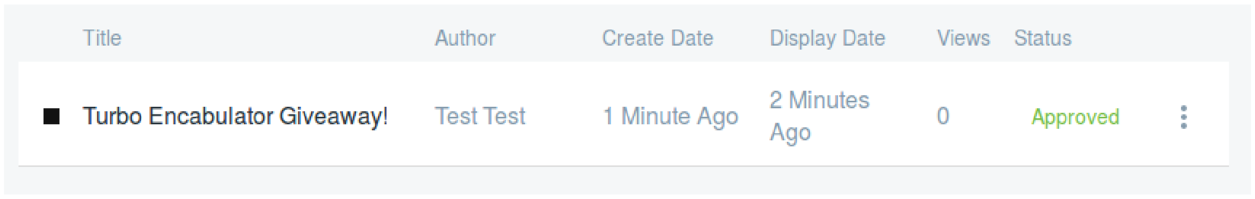
Now all you need to do is update your JSP to use the appropriate getter.

```
<liferay-ui:search-container-results
    results="<%=FooEntityLocalServiceUtil.getFooEntities(scopeGroupId,
             fooEntityId(), Workflowconstants.STATUS_APPROVED, searchContainer.getStart(),
             searchContainer.getEnd())%>"
    ...
```

In an *administrative* type application (in other words, one that's displayed in the Site Menu's *Content* section) you might want to display all the entities, with their current workflow status (for example, include workflow status as a column in the search container). To do so, use the `<aui:worklfow-status>` tag.

```
    <aui:workflow-status markupView="lexicon" showIcon="<%= false %>" showLabel="<%= false %>" status="<%= fooEntity.getStatus() %>" />
```

You only needed one new class, one new method in the service layer, and some updates to your view, and workflow is fully implemented and ready to use in your Liferay application.



	Title	Author	Create Date	Display Date	Views	Status	
■	Turbo Encabulator Giveaway!	Test Test	1 Minute Ago	2 Minutes Ago	0	Approved	⋮

Figure 87.1: You can display the workflow status of your entities. This is useful in administrative applications.

Chapter 88

Export/Import and Staging

Liferay DXP's Export/Import and Staging features give users the power to plan page publication and manage content. The Export/Import feature lets users export content from the Portal and import external content into the Portal. Providing the export feature in your application allows users the flexibility of exporting content they've created in your application to other places, such as another portal instance, or to save the content for a later use. Import does the opposite: it brings the data from a LAR file into your portal.

For instance, suppose you're managing an online education course using Liferay DXP. Because of the nature of an online course, the site's data (grades, assignments, etc.) is purged every semester to make way for new incoming students. In a scenario like this, there is a need frequently to store a complete record of all data given during a course. The institution offering the course must usually keep records of the course's data for a minimum number of years. To abide by these requirements, having a gradebook application with an export/import feature would allow you to clear the application's data for a new semester, but save the previous class's work. You could export the students' grades as a LAR file and save it outside the course's site. If the grades ever needed to be accessed again, you could import the LAR and view the student records.

The Export/Import feature adds another dimension to your application by letting you produce reusable content and import content from other places. To learn more about using the Export/Import feature, visit the Exporting/Importing App Data section of the User Guide.

Staging lets you change your site behind the scenes without affecting the live site, and then you can publish all the changes in one fell swoop. Keep in mind that Staging leverages the Export/Import framework, which is an essential part of the publishing process. If you include staging support in your application, your users can stage its content until it's ready.

For example, if you have an application that provides information intended only during a specific holiday, supporting the Staging environment lets users save your application's assets specific for that holiday. They'll reside in the Staging environment until they're ready for publishing. To learn more about Staging, visit the Staging Content for Publication section.

Besides configuring these features for your application, Liferay also provides an API that allows developers to write custom code, extending Liferay's default functionality.

In this section of tutorials, you'll learn how to implement Staging and the Export/Import framework. The main areas of Staging code to focus on are outlined below:

1. `StagedModel` Interface: The `StagedModel` is the cornerstone of Staging. All content that must be handled in Staging should implement this interface; it provides the behavior contract for the entities Staging uses during the Staging process.

2. `StagedModelDataHandler`: These data handlers are responsible for handling one specific entity class. For example, the `BookmarksEntryStagedModelDataHandler` handles the `BookmarksEntry` during Staging: exporting data, serializing content, finding existing entries, etc.

3. `PortletDataHandler`: These data handlers are responsible for handling aspects of the portlet's configuration and publication during Staging.

4. `ExportActionableDynamicQuery`: This framework is useful when developing Staging support. Its purpose is to query data from the database and process it during publication. It's automatically generated if your entity contains the right fields so there's no need to worry about configuring it.

5. `ExportImportContentProcessor` and `ExportImportPortletPreferencesProcessor`: Advanced frameworks only needed in special cases. The `ExportImportContentProcessor` lets you process your content during a publication process. The `ExportImportPortletPreferencesProcessor` lets you process your portlet preferences (application's configuration) during a publication process.

88.1 Decision to Implement Staging

Liferay DXP's Staging feature is an advanced publication tool that lets you create or modify your site before releasing it to the public. Most of Liferay DXP's included applications (e.g., Web Content and Bookmarks) support Staging. Implementing Staging in your own application can be beneficial, but how do you know if it's the right move?

Not every application needs to support Staging and Export/Import. The most important question to consider during the decision process is

What part of your application are you primarily focused on using Staging for?

When Staging is enabled in Liferay DXP, all pages and applications are staged automatically. Liferay DXP's architecture separates the application and its configuration from the actual content, meaning that content can exist without any application to display it and vice versa. Although Staging and the Export/Import framework supports all applications and their configurations by default, not all applications' content is supported by Staging.

Implementing Staging for your application means you're defining the logic for how the Staging framework should process, serialize, and de-serialize your app's content, and how to insert it into a database.

Therefore, if you want to track your application's content, you should implement Staging in your application. Here are a few other scenarios where you should implement Staging in your application:

- You're using remote staging. When publishing to a remote live site, your content must be transferred to a different Liferay DXP installation. Therefore, Staging must be able to recognize the content to facilitate the transfer.
- You want a space where you can freely edit and test your content before publishing it to a live audience.
- Your content is being referenced from another content type that supports Staging.
- You want to process your portlet's preferences during publication (i.e., you might want to publish some content with it or complete extra steps).
- You want to process the content during publication (e.g., writing validation for your content during the import process).

If none of these options are beneficial for you, implementing Staging in your application is unnecessary.

When content supports Staging and Staging is enabled, it is created in a Staging group and is only published to a live site when that site is published. When content is **not** supported by Staging, it is never added to a Staging group and is not reviewable during the Staging publication process; it's added and removed from the live site only.

From a technical standpoint, publishing an entity or content follows the process below:

1. The entity's possible references are discovered and processed.
2. The entity's fields are processed.
3. The entity is serialized into a LAR file.
4. The LAR is transferred to the live site (local or remote live).
5. After de-serialization, the entity's fields are processed.
6. The entity is added to the database.

Awesome! You should now have a good idea about whether you should implement Staging for your application.

88.2 Understanding Staged Models

To track an entity of an application with the Staging framework, you must implement the StagedModel interface in the app's model classes. It provides the behavior contract for entities during the Staging process. For example, the Bookmarks application manages BookmarksEntrys and BookmarksFolders, and both implement the `StagedModel` interface. Once you've configured your staged models, you can create staged model data handlers, which supply information about a staged model (entity) and its referenced content to the Export/Import and Staging frameworks. See the Understanding Data Handlers tutorial for more information.

There are two ways to create staged models for your application's entities:

- Using Service Builder to generate the required Staging implementations (tutorial).
- Implementing the required Staging interfaces manually (tutorial).

You can follow step-by-step procedures for creating staged models for your entities by visiting their respective tutorials.

Using Service Builder to generate your staged models is the easiest way to create staged models for your app. You define the necessary columns in your `service.xml` file and set the `uuid` attribute to `true`. Then you run Service Builder, which generates the required code for your new staged models.

Implementing the necessary staged model logic *manually* should be done if you **don't** want to extend your model with special attributes only required to generate Staging logic (i.e., not needed by your business logic). In this case, you should adapt your business logic to meet the Staging framework's needs. You'll learn more about this later.

You'll explore the provided staged model interfaces next.

Staged Model Interfaces

The `StagedModel` interface must be implemented by your app's model classes, but this is typically done through inheritance by implementing one of the interfaces that extend the base interface:

- StagedAuditedModel
- StagedGroupedModel

You must implement these when you want to use certain features of the Staging framework like automatic group mapping or entity level *Last Publish Date* handling. So how do you choose which is right for you?

The `StagedAuditedModel` interface provides all the audit fields to the model that implements it. You can check the AuditedModel interface for the specific audit fields provided. The `StagedAuditedModel` interface is intended for models that function independent from the group concept (sometimes referred to as company models). If your model is a group model, you should not implement the `StagedAuditedModel` interface.

The `StagedGroupedModel` interface must be implemented for group models. For example, if your application requires the `groupId` column, your model is a group model. If your model satisfies both the `StagedGroupModel` and `StagedAuditedModel` requirements, it should implement the `StagedGroupedModel`. Your model should only implement the `StagedAuditedModel` if it doesn't fulfill the grouped model needs, but does fulfill the audited model needs. If your model does not fulfill either the `StagedAuditedModel` or `StagedGroupedModel` requirements, you should implement the base `StagedModel` interface.

As an example for extending your model class, you can visit the BookmarksEntryModel class, which extends the `StagedGroupedModel` interface; this is done because bookmark entries are group models.

```
public interface BookmarksEntryModel extends BaseModel<BookmarksEntry>,
    ShardedModel, StagedGroupedModel, TrashedModel, WorkflowedModel {
```

Now that you have a better understanding about staged model interfaces, you'll dive into the attributes used in Staging and why they're important.

Important Attributes in Staging

If you'd like to generate your staged models using Service Builder, you must define the proper attributes in your project's `service.xml` file. If you'd like more detail on how this is done, see the Generating Staged Models using Service Builder tutorial. You'll learn some general information about this process next.

One of the most important attributes used by the Staging framework is the UUID (Universally Unique Identifier). This attribute must be set to `true` in your `service.xml` file for Service Builder to recognize your model as an eligible staged model. The UUID is used to differentiate entities between environments. Because the UUID always remains the same, it's unique across multiple systems. Why is this so important?

Suppose you're using remote staging and you create a new entity on your local staging site and publish it to your remote live site. What happens when you go back to modify the entity on your local site and want to publish those changes? Without a UUID, the Staging framework has no way to know the local and remote entities are the same. To publish entities properly, the Staging framework needs entities uniquely identified across systems to recognize the original entity on the remote site and update it. The UUID provides that.

In addition to the UUID, there are several columns that must be defined in your `service.xml` file for Service Builder to define your model as a staged model:

- `companyId`
- `createDate`
- `modifiedDate`

If you want a staged grouped model, also include the `groupId` and `lastPublishDate` columns. If you want a staged audited model, include the `userId` and `userName` columns.

Next, you'll learn how to build staged models from scratch.

Adapting Your Business Logic to Build Staged Models

What if you don't want to extend your model with special attributes that may not be needed in your business logic? In this case, you should adapt your business logic to meet the Staging framework's needs. Liferay provides the ModelAdapterBuilder framework, which lets you adapt your model classes to staged models.

As an example, assume you have an app that is fully developed and you want to configure it to work with Staging. Your app, however, does not require a UUID for any of its entities, and therefore, does not provide them. Instead of configuring your app to handle UUIDs just for the sake of generating staged models, you can leverage the Model Adapter Builder to build your staged models.

Another example for building staged models from scratch is for applications that use REST services to access their attributes instead of the database. Since this kind of app is developed to pull its attributes from a remote system, it would be more convenient to build your staged models yourself instead of relying on Service Builder, which is database driven.

To adapt your model classes to staged models, follow the steps outlined below:

1. Create a `Staged[Entity]` interface, which extends the model specific interface (e.g., `[Entity]`) and the appropriate staged model interface (e.g., `StagedModel`). This class serves as the Staged Model Adapter.
2. Create a `Staged[Entity]Impl` class that implements the `Staged[Entity]` interface and provides necessary logic for your entity model to be recognized as a staged model.
3. Create a `Staged[Entity]ModelAdapterBuilder` class that implements `ModelAdapterBuilder<[Entity], Staged[Entity]>`. This class adapts the original model to the newly created Staged Model Adapter.
4. Adapt your existing model and call one of the provided APIs to export or import the entity automatically.

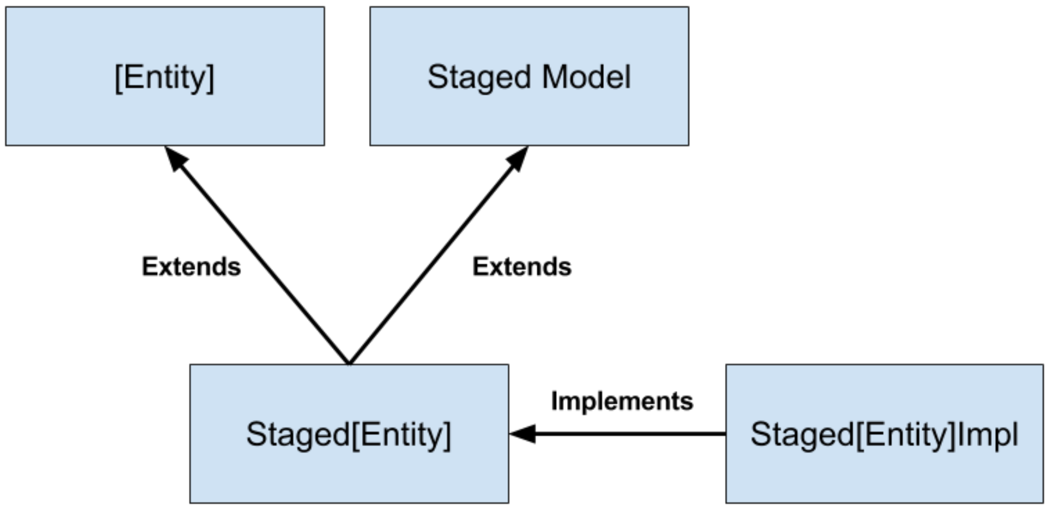


Figure 88.1: The Staged Model Adapter class extends your entity and staged model interfaces.

To step through the process for leveraging the Model Adapter Builder for an existing app, visit the Creating Staged Models Manually tutorial.

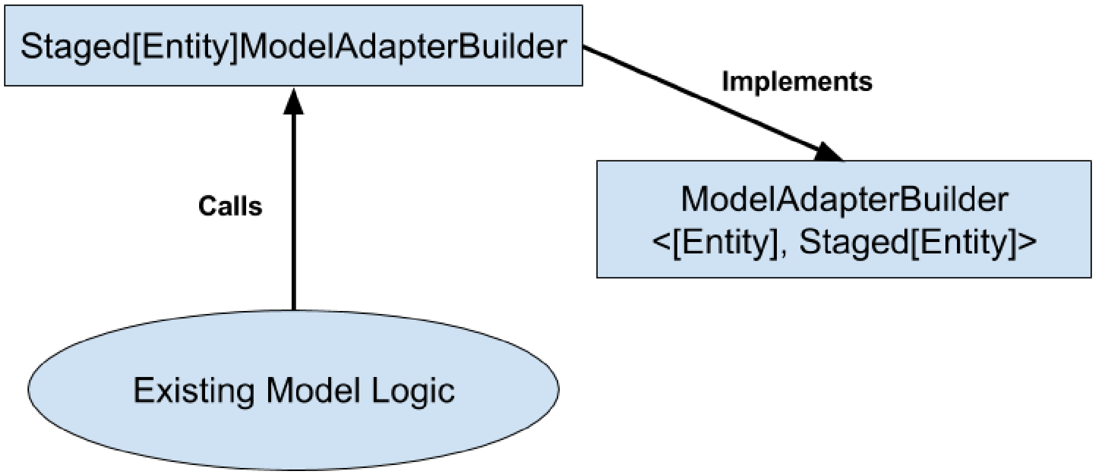


Figure 88.2: The Model Adapter Builder gets an instance of the model and outputs a staged model.

88.3 Generating Staged Models Using Service Builder

This document has been updated and ported to Liferay Learn and is no longer maintained here.

A Staged model is an essential building block to implementing the Staging and Export/Import frameworks in your application. Instead of having to create staged models for your app manually, you can leverage Service Builder to generate the necessary staged model logic for you. Before diving into this tutorial, make sure you've read the Understanding Staged Models tutorial for information on how staged models work. Also, if your app doesn't use Liferay's Service Builder, you must configure it in your project. If you need help doing this, follow the Defining an Object-Relational Map with Service Builder tutorial.

This tutorial assumes you have a Service Builder project with `*api` and `*service` modules. If you want to follow along with this tutorial, download the staged-model-example Service Builder project. This is a bare-bones project that you can test to observe the Staging-related changes generated by running Service Builder. This tutorial assumes your project is built with Gradle. The example project's `service.xml` file contains the following configuration:

```
<service-builder package-path="com.liferay.docs">
    <namespace>FOO</namespace>
    <entity local-service="true" name="Foo" remote-service="true" uuid="true">

        <!-- PK fields -->

        <column name="fooId" primary="true" type="long" />

        <!-- Group instance -->

        <column name="groupId" type="long" />

        <!-- Audit fields -->
        <column name="companyId" type="long" />
        <column name="createDate" type="Date" />
        <column name="modifiedDate" type="Date" />

        ...
        ...

    </entity>
</service-builder>
```

For simplicity, you'll track the Service Builder-generated changes applied to an entity model file to observe how staged models are assigned to your entity. Keep in mind the specific staged attributes necessary for each staged model. Depending on the attributes defined in your `service.xml` file, Service Builder assigns your entity model to a specific staged model type.

1. Navigate to your project's *`service` module at the command line. Run Service Builder (e.g., `gradlew buildService`) to generate your project's models based on the current `service.xml` configuration.

2. Open your project's `[Entity]Model.java` interface and observe the inherited interfaces.

   ```
   public interface FooModel extends BaseModel<Foo>, ShardedModel, StagedModel {
   ```

   Your model was generated as a staged model! This is because the UUID is set to true and the `companyId`, `createDate`, and `modifiedDate` columns are defined. There is much more logic generated for your app behind the scenes, but this shows that Service Builder deemed your entity eligible for the Staging and Export/Import frameworks.

3. Add the `userId` and `userName` columns to your `service.xml` file:

   ```
       <column name="userId" type="long" />
       <column name="userName" type="String" />
   ```

4. Rerun Service Builder and observe your `[Entity]Model.java` interface again:

   ```
   public interface FooModel extends BaseModel<Foo>, GroupedModel, ShardedModel,
       StagedAuditedModel {
   ```

   Your model is now a staged audited model!

5. Add the `lastPublishDate` column to your `service.xml` file:

   ```
   <column name="lastPublishDate" type="Date" />
   ```

6. Rerun Service Builder and observe your `[Entity]Model.java` interface again:

   ```
   public interface FooModel extends BaseModel<Foo>, ShardedModel,
       StagedGroupedModel {
   ```

   Your model is now a staged grouped model! The `groupId` column is also required to extend the `StagedGroupedModel` interface, but it was already defined in the original `service.xml` file.

Fantastic! You've witnessed firsthand how easy it is to generate staged models using Service Builder.

88.4 Creating Staged Models Manually

There are times when using Service Builder to generate your staged models is not practical. In these cases, you should create your staged models manually. Make sure to read the Adapting Your Business Logic to Build Staged Models section to determine if creating staged models manually is beneficial for your use case.

In this tutorial, you'll explore how the Asset Link framework (a Liferay DXP framework used for relating assets) manually creates staged models. This framework is separate from Staging and is referenced solely as an example for how to leverage the ModelAdapterBuilder framework, which lets you adapt your model classes to staged models.

Asset links do not provide UUIDs by default; however, they still need to be tracked in the Staging and Export/Import frameworks. Therefore, they require staged models. Since they don't provide a UUID, Service Builder cannot generate staged models for asset links. The Asset Link framework has to create staged models differently using the Model Adapter Builder. The naming convention for this interface typically follows the `Staged[Entity]` syntax. The Asset Link framework uses a generic entity called `AssetLink`.

Follow the steps below to leverage the Model Adapter Builder in your app.

1. Create a new interface that extends one of the staged model interfaces and your model specific interface. For example,

   ```
   public interface StagedAssetLink extends AssetLink, StagedModel {

   }
   ```

   This interface should define methods required for your model to qualify as a staged model. For asset links, methods for retrieving entry UUIDs (among others) are defined:

   ```
   public String getEntry1Uuid();

   public String getEntry2Uuid();
   ```

   These will be implemented by a new implementation class later.

2. Create an implementation class that implements your new `Staged[Entity]`. For example, the Asset Link framework does this:

   ```
   public class StagedAssetLinkImpl implements StagedAssetLink {

   }
   ```

   This class provides necessary logic for your entity model to be recognized as a staged model. Below is a subset of logic in the example `StagedAssetLinkImpl` class used to populate UUIDs for asset link entries:

   ```
   public StagedAssetLinkImpl(AssetLink assetLink) {
        _assetLink = assetLink;

       ...

        populateUuid();
   }

   @Override
   public String getEntry1Uuid() {
       if (Validator.isNotNull(_entry1Uuid)) {
           return _entry1Uuid;
   ```

```
    }

    populateEntry1Attributes();

    return _entry1Uuid;
}

@Override
public String getEntry2Uuid() {
    if (Validator.isNotNull(_entry2Uuid)) {
        return _entry2Uuid;
    }

    populateEntry2Attributes();

    return _entry2Uuid;
}

protected void populateEntry1Attributes() {

    ...

    AssetEntry entry1 = AssetEntryLocalServiceUtil.fetchAssetEntry(
        _assetLink.getEntryId1());

    ...

    _entry1Uuid = entry1.getClassUuid();
}

protected void populateEntry2Attributes() {

    ...

    AssetEntry entry2 = AssetEntryLocalServiceUtil.fetchAssetEntry(
        _assetLink.getEntryId2());

    ...

    _entry2Uuid = entry2.getClassUuid();
}

protected void populateUuid() {

    ...

    String entry1Uuid = getEntry1Uuid();
    String entry2Uuid = getEntry2Uuid();

    ...

    _uuid = entry1Uuid + StringPool.POUND + entry2Uuid;
        }
}

private AssetLink _assetLink;
private String _entry1Uuid;
private String _entry2Uuid;
private String _uuid;
```

This logic retrieves asset link entries and populates UUIDs for them usable by the Staging and Export/Import frameworks. With the newly generated UUIDs, asset link model classes can be converted to staged models.

3. Create a Model Adapter Builder class and implement the ModelAdapterBuilder interface. You should define the entity type and your Staged Model Adapter class when implementing the interface:

```
public class StagedAssetLinkModelAdapterBuilder
    implements ModelAdapterBuilder<AssetLink, StagedAssetLink> {

    @Override
    public StagedAssetLink build(AssetLink assetLink) {
        return new StagedAssetLinkImpl(assetLink);
    }

}
```

For the `StagedAssetLinkModelAdapterBuilder`, the entity type is `AssetLink` and the Staged Model Adapter is `StagedAssetLink`. Your app should follow a similar design. The Model Adapter Builder outputs a new instance of the `Staged[Entity]Impl` object.

4. Now you need to adapt your existing business logic to call the provided APIs. You can call the ModelAdapterUtil class to create an instance of your Staged Model Adapter. See how the Asset Link framework does this below:

```
StagedAssetLink stagedAssetLink = ModelAdapterUtil.adapt(
    assetLink, AssetLink.class, StagedAssetLink.class);
```

Once you've created Staged Model Data Handlers, you can begin exporting/importing your now Staging-compatible entities:

```
StagedModelDataHandlerUtil.exportStagedModel(
    portletDataContext, stagedAssetLink);
```

Visit the Understanding Data Handlers tutorial if you're unfamiliar with how data handlers work.

Awesome! You've successfully adapted your business logic to build staged models!

88.5 Understanding Data Handlers

A common requirement for many data driven applications is to import and export data. This *could* be accomplished by accessing your database directly and running SQL queries to export/import data; however, this has several drawbacks:

- Working with different database vendors might require customized SQL scripts.
- Access to the database may be tightly controlled, restricting the ability to export/import on demand.
- You'd have to come up with your own means of storing and parsing the data.

Liferay provides a more convenient and reliable way to export/import your data without accessing the database.

Liferay Archive (LAR) File

An easier way to export/import your application's data is to use a Liferay ARchive (LAR) file. Liferay provides the LAR feature to address the need to export/import data in a database agnostic manner. So what exactly is a LAR file?

A LAR file is a compressed file (ZIP archive) Liferay DXP uses to export/import data. LAR files can be created for single portlets, pages, or sets of pages. Portlets that are LAR-capable provide an interface to let you control how their data are imported/exported. There are several Liferay DXP use cases that require the use of LAR files:

- Backing up and restoring portlet-specific data without requiring a full database backup.
- Cloning sites.
- Specifying a template to be used for users' public or private pages.
- Using Local Live or Remote Live staging.

Liferay provides the data handler framework so developers don't have to create/modify a LAR file manually. **It is strongly recommended never to modify a LAR file.** You should always use Liferay's provided data handler APIs to construct it.

Knowing how a LAR file is constructed, however, is beneficial to understand the overall purpose of your application's data handlers. Next, you'll explore a LAR file's anatomy.

LAR File Anatomy

What is a LAR file? You know the general concept for *why* it's used, but you may want to know what lives inside to make your export/import processes work. With a fundamental understanding for how a LAR file is constructed, you can better understand what your data handlers generate behind the scenes.

Below is the structure of a simple LAR file. It illustrates the exportation of a single Bookmarks entry and the portlet's configuration:

- `Bookmarks_Admin-201701091904.portlet.lar`
  - `group`
    - `20143`
      - `com.liferay.bookmarks.model.BookmarksEntry`
      - `35005.xml`

      - `portlet`
      - `com_liferay_bookmarks_web_portlet_BookmarksAdminPortlet`
      - `20137`
      - `portlet.xml`

      - `20143`
      - `portlet-data.xml`
  - `manifest.xml`

You can tell from the LAR's generated name what information is contained in the LAR: the Bookmarks Admin app's data. The `manifest.xml` file sits at the root of the LAR file. It provides essential information about the export process. The `manifest.xml` for the sample Bookmarks LAR is pretty bare since it's not exporting much content, but this file can become large when exporting pages of content. There are four main parts (tags) to a `manifest.xml` file.

- `header`: contains information about the LAR file, current process, and site you're exporting (if necessary). For example, it can include locales, build information, export date, company ID, group ID, layouts, themes, etc.

- `missing-references`: lists entities that must be validated during import. For example, suppose you're exporting a web content article that references an image (e.g., an embedded image residing in the document library). If the image was not selected for export, the image must already exist in the site where the article is imported. Therefore, the image would be flagged as a missing reference in the LAR file. If the missing reference does not exist in the site when the LAR is imported, the import process fails. If your import fails, the Import UI shows you the missing references that weren't validated.
- `portlets`: defines the portlets (i.e., portlet data) exported in the LAR. Each portlet definition has basic information on the exported portlet and points to the generated `portlet.xml` for more specialized portlet information.
- `manifest-summary`: contains information on what has been exported. The Staging and Export frameworks export or publish some entities even though they weren't marked for it, because the process respects data integrity. This section holds information for all the entities that have been processed. The entities defining a non-zero `addition-count` attribute are displayed in the Export/Import UI.

The `manifest.xml` file also defines layout information if you've exported pages in your LAR. For example, your manifest could have `LayoutSet`, `Layout`, and `LayoutFriendlyURL` tags specifying staged models and their various references in an exported page.

Now that you've learned about the LAR's `manifest.xml` and how it's used to store high-level data about your export process, you can dive deeper into the LAR file's `group` folder. The `group` folder has two main parts:

- Entities
- Portlets

If you look at the anatomy of the sample Bookmarks LAR, you'll notice that `group/[groupId]` folder holds a folder named after the entity you're exporting (e.g., `com.liferay.bookmarks.model.BookmarksEntry`) and a `portlet` folder holding a folder named after the portlet from which you're exporting (e.g., `com_liferay_bookmarks_web_portlet_BookmarksAdminPortlet`). For each entity/portlet you export, there are subsequent folders holding data about them. Entities and portlets can also be stored in a `company` folder. Although the majority of entities belong to a group, some exist outside of a group scope (e.g., users).

If you open the `/group/20143/com.liferay.bookmarks.model.BookmarksEntry/35005.xml` file, you'll find serialized data about the entity, which is similar to what is stored in the database.

The `portlet` folder holds all the portlets you exported. Each portlet has its own folder that holds various XML files with data describing the exported content. There are three main XML files that can be generated for a single portlet:

- `portlet.xml`: provides essential information about the portlet, similar to a manifest file. For example, this can include the portlet ID, high-level entity information stored in the portlet (e.g., web content articles in a web content portlet), permissioning, etc.
- `portlet-data.xml`: describes specific entity data stored in the portlet. For example, for the web content portlet, articles stored in the portlet are defined in `staged-model` tags and are linked to their serialized entity XML files.
- `portlet-preferences.xml`: defines the settings of the portlet. For example, this can include portlet preferences like the portlet owner, default user, article IDs, etc.

Note that when you import a LAR, it only includes the portlet data. You have to deploy the portlet to be able to use it.

You now know how exported entities, portlets, and pages are defined in a LAR file. For a summarized outline of what you've learned about LAR file construction, see the diagram below.

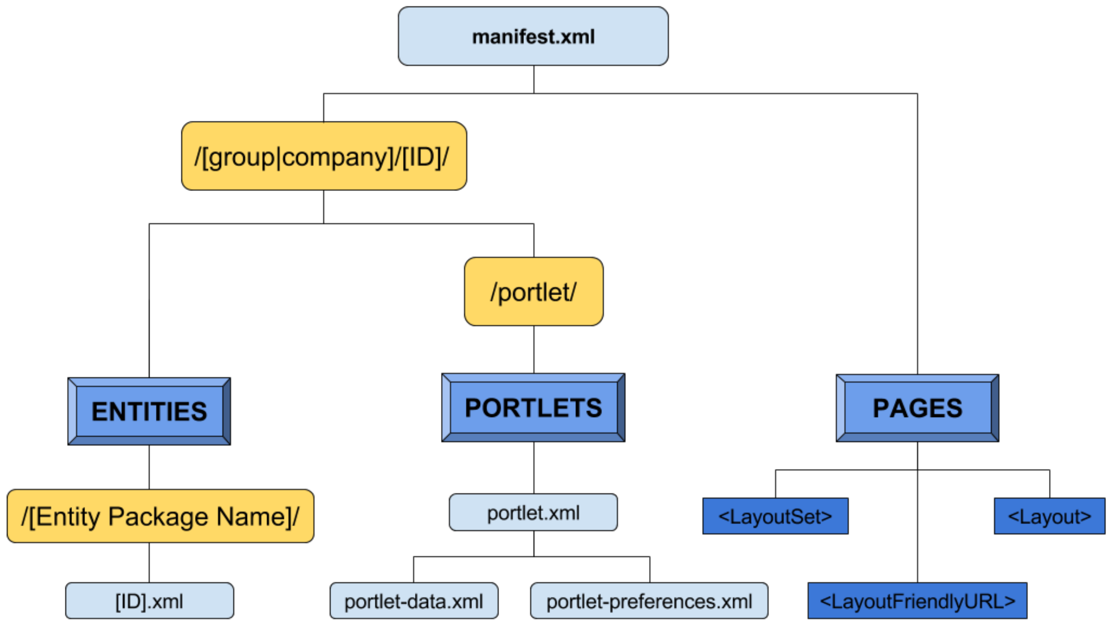


Figure 88.3: Entities, Portlets, and Pages are defined in a LAR in different places.

Excellent! You now have a fundamental understanding for how a LAR file is generated and how it's structured.

Next, you'll learn about data handler fundamentals and the prerequisites required to implement them.

Data Handler Fundamentals

To leverage the Export/Import framework's ability to export/import a LAR file, you can implement Data Handlers in your application. There are two types of data handlers: *Portlet Data Handlers* and *Staged Model Data Handlers*.

A Portlet Data Handler imports/exports portlet specific data to a LAR file. These classes only have the role of querying and coordinating between staged model data handlers. For example, the Bookmarks application's portlet data handler tracks system events dealing with Bookmarks entities. It also configures the Export/Import UI options for the Bookmarks application.

To track each entity of an application for staging, you should create staged models by implementing the StagedModel interface. Staged models are the parent interface of an entity in the Staging framework. For example, the Bookmarks application manages BookmarksEntrys and BookmarksFolders, and both implement the `StagedModel` interface.

A Staged Model Data Handler supplies information about a staged model (entity) to the Export/Import framework, defining a display name for the UI, deleting an entity, etc. It's also responsible for exporting referenced content. For example, if a Bookmarks entry resides in a Bookmarks folder, the `BookmarksEntry` staged model data handler invokes the export of the `BookmarksFolder`.

You're not required to implement a staged model data handler for every entity in your application, but they're necessary for any entity you want to export/import or have the staging framework track.

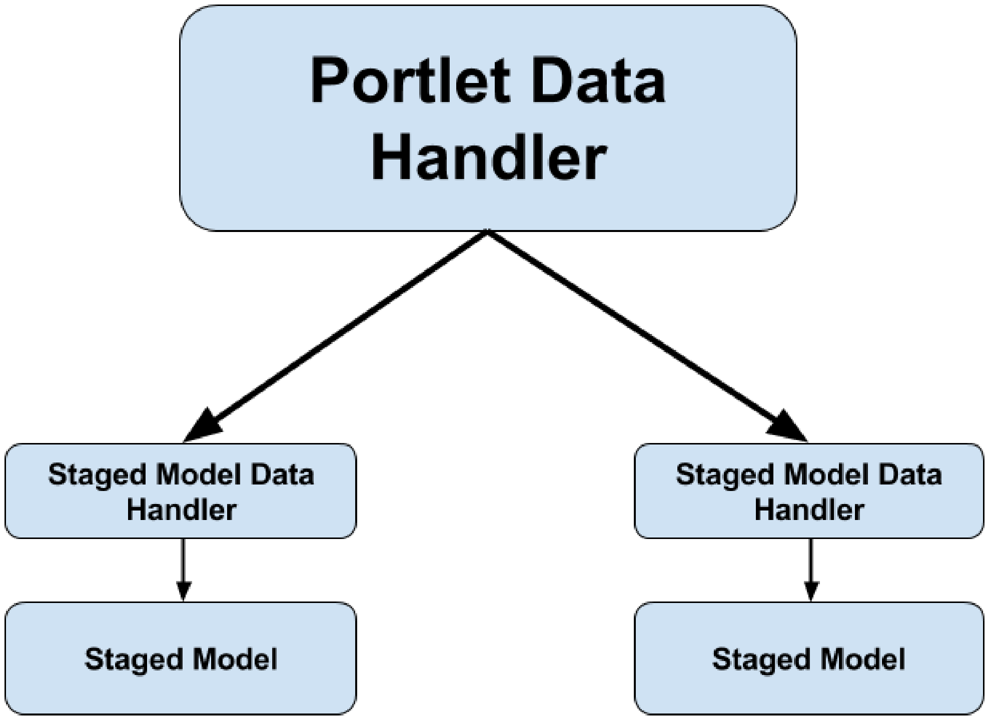


Figure 88.4: The Data Handler framework uses portlet data handlers and staged model data handlers to track and export/import portlet and staged model information, respectively.

Before implementing data handlers, make sure your application is ready for the Export/Import and Staging frameworks by running Service Builder in your application. Using Service Builder to create staged models is not required, but is recommended since it generates many requirements for you. To ensure Service Builder recognizes your entity as a staged model, you must set the `uuid` attribute to `true` in your `service.xml` file and have the following columns declared:

- `companyId`
- `groupId`
- `userId`
- `userName`
- `createDate`
- `modifiedDate`

You can learn how to create a `service.xml` file for your application by visiting the Defining an Object-Relational Map with Service Builder tutorial.

To learn how to develop data handlers for your app, visit the Developing Data Handlers tutorial.

88.6 Developing Data Handlers

To leverage the Export/Import framework's ability to export/import a LAR file, you can implement Data Handlers in your application. There are two types of data handlers you can implement: *Portlet Data Handlers* and *Staged Model Data Handlers*. For more information on the fundamentals behind Liferay's data handlers and how a LAR file is constructed, see the Understanding Data Handlers tutorial. You also must ensure your application is properly configured to use data handlers; this is also covered in the linked tutorial.

To learn how to implement data handlers for your custom application, you'll examine how the Bookmarks application does it. First you'll start with its portlet data handler implementation.

Portlet Data Handlers

The following steps create the `BookmarksPortletDataHandler` class used for the Bookmarks application.

1. Create a new package in your existing Service Builder project for your data handler classes. For instance, the Bookmarks application's data handler classes reside in the `com.liferay.bookmarks.service` module's `com.liferay.bookmarks.internal.exportimport.data.handler` package.

2. Create your `-PortletDataHandler` class for your application in the new `-exportimport.data.handler` package and have it implement the PortletDataHandler interface by extending the BasePortletDataHandler class. See the `BookmarksPortletDataHandler` class as an example:

   ```
   public class BookmarksPortletDataHandler extends BasePortletDataHandler {
   ```

3. Create an `@Component` annotation section above the class declaration. This annotation registers this class as a portlet data handler in the OSGi service registry.

   ```
   @Component(
       immediate = true,
       property = {
           "javax.portlet.name=" + BookmarksPortletKeys.BOOKMARKS,
           "javax.portlet.name=" + BookmarksPortletKeys.BOOKMARKS_ADMIN
       },
       service = PortletDataHandler.class
   )
   ```

   There are a few annotation attributes you should set:

   - The `immediate` element directs the container to activate the component immediately once its provided module has started.
   - The `property` element sets various properties for the component service. You must associate the portlets you wish to handle with this service so they function properly in the export/import environment. For example, since the Bookmarks data handler is used for two portlets, they're both configured using the `javax.portlet.name` property.
   - The `service` element should point to the `PortletDataHandler.class` interface.

```
**Note:** In previous versions of Liferay DXP, you had to register the portlet
data handler in a portlet's `liferay-portlet.xml` file. The registration
process is now completed automatically by OSGi using the `@Component`
annotation.
```

---

4. Set what the portlet data handler controls and the portlet's Export/Import UI by adding an `activate` method:

```
@Activate
protected void activate() {
    setDataPortletPreferences("rootFolderId");
    setDeletionSystemEventStagedModelTypes(
        new StagedModelType(BookmarksEntry.class),
        new StagedModelType(BookmarksFolder.class));
    setExportControls(
        new PortletDataHandlerBoolean(
            NAMESPACE, "entries", true, false, null,
            BookmarksEntry.class.getName()));
    setImportControls(getExportControls());
}
```

This method is called during initialization of the component by using the @Activate annotation. This method is invoked after dependencies are set and before services are registered.

The four set methods called in the BookmarksPortletDataHandler's activate method are described below:

- setDataPortletPreferences: sets portlet preferences the Bookmarks application should handle.
- setDeletionSystemEventStagedModelTypes: sets the staged model deletions that the portlet data handler should track. For the Bookmarks application, Bookmark entries and folders are tracked.
- setExportControls: adds fine grained controls over export behavior that are rendered in the Export UI. For the Bookmarks application, a checkbox is added to select Bookmarks content (entries) to export.
- setImportControls: adds fine grained controls over import behavior that are rendered in the Import UI. For the Bookmarks application, a checkbox is added to select Bookmarks content (entries) to import.

For more information on these methods, visit the PortletDataHandler API.

5. For the Bookmarks portlet data handler to reference its entry and folder staged models successfully, you must set them in your class:

```
@Reference(unbind = "-")
protected void setBookmarksEntryLocalService(
    BookmarksEntryLocalService bookmarksEntryLocalService) {

    _bookmarksEntryLocalService = bookmarksEntryLocalService;
}

@Reference(unbind = "-")
protected void setBookmarksFolderLocalService(
    BookmarksFolderLocalService bookmarksFolderLocalService) {

    _bookmarksFolderLocalService = bookmarksFolderLocalService;
}

private BookmarksEntryLocalService _bookmarksEntryLocalService;
private BookmarksFolderLocalService _bookmarksFolderLocalService;
```

The set methods must be annotated with the @Reference annotation. Visit the Invoking Liferay Services Locally section of the *Finding and Invoking Liferay Services* tutorial for more information on using the 'Liferay DXPproduct@.

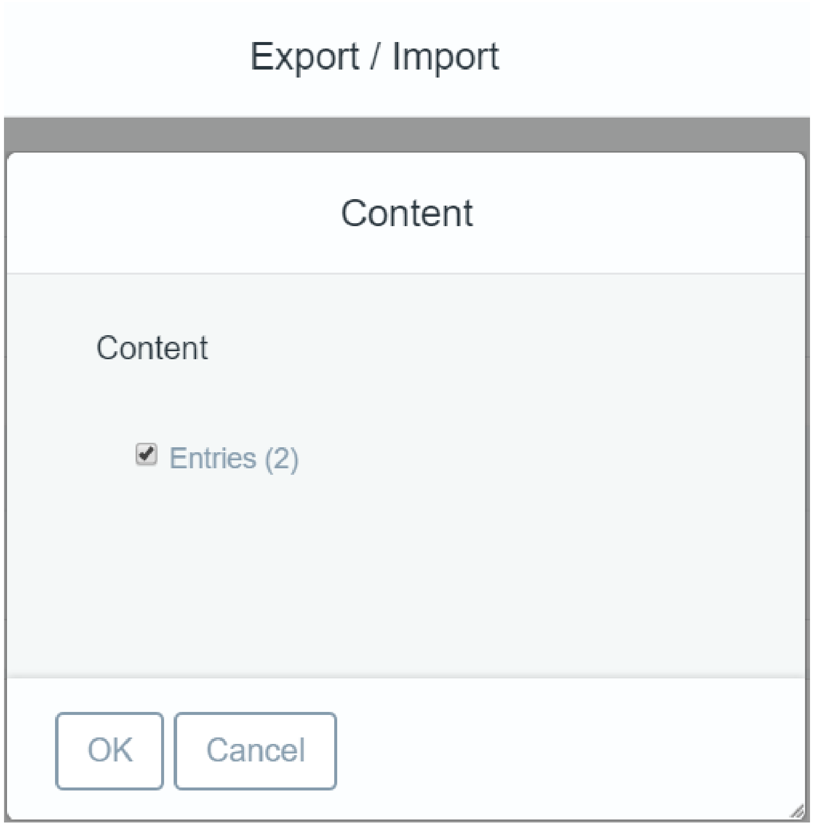


Figure 88.5: You can select the content types you'd like to export/import in the UI.

Important: Liferay DXP's official Bookmarks app does not use local services in its portlet data handler; instead, it uses the `StagedModelRepository` framework. This is not recommended for custom applications, however; it's only intended for internal Liferay applications at this time. Because of this, the code in this tutorial has been modified to highlight the recommended way for custom apps.

6. You must create a namespace for your entities so the Export/Import framework can identify your application's entities from other entities in Liferay DXP. The Bookmarks application's namespace declaration looks like this:

```
public static final String NAMESPACE = "bookmarks";
```

   You'll see how this namespace is used later.

7. Your portlet data handler should retrieve the data related to its staged model entities so it can properly export/import it. Add this functionality by inserting the following methods:

```
@Override
protected String doExportData(
        final PortletDataContext portletDataContext, String portletId,
        PortletPreferences portletPreferences)
    throws Exception {

    Element rootElement = addExportDataRootElement(portletDataContext);

    if (!portletDataContext.getBooleanParameter(NAMESPACE, "entries")) {
        return getExportDataRootElementString(rootElement);
    }

    portletDataContext.addPortletPermissions(
        BookmarksResourcePermissionChecker.RESOURCE_NAME);

    rootElement.addAttribute(
        "group-id", String.valueOf(portletDataContext.getScopeGroupId()));

    ExportActionableDynamicQuery folderActionableDynamicQuery =
        _bookmarksFolderLocalService.
            getExportActionableDynamicQuery(portletDataContext);

    folderActionableDynamicQuery.performActions();

    ActionableDynamicQuery entryActionableDynamicQuery =
        _bookmarksEntryLocalService.
            getExportActionableDynamicQuery(portletDataContext);

    entryActionableDynamicQuery.performActions();

    return getExportDataRootElementString(rootElement);
}

@Override
protected PortletPreferences doImportData(
        PortletDataContext portletDataContext, String portletId,
        PortletPreferences portletPreferences, String data)
    throws Exception {

    if (!portletDataContext.getBooleanParameter(NAMESPACE, "entries")) {
        return null;
    }

    portletDataContext.importPortletPermissions(
        BookmarksResourcePermissionChecker.RESOURCE_NAME);

    Element foldersElement = portletDataContext.getImportDataGroupElement(
        BookmarksFolder.class);

    List<Element> folderElements = foldersElement.elements();

    for (Element folderElement : folderElements) {
        StagedModelDataHandlerUtil.importStagedModel(
            portletDataContext, folderElement);
    }

    Element entriesElement = portletDataContext.getImportDataGroupElement(
        BookmarksEntry.class);

    List<Element> entryElements = entriesElement.elements();

    for (Element entryElement : entryElements) {
        StagedModelDataHandlerUtil.importStagedModel(
            portletDataContext, entryElement);
    }

    return null;
}
```

The doExportData method first checks if anything should be exported. The `portletDataContext.getBooleanParameter(...` method checks if the user selected Bookmarks entries for export. Later, the `ExportImportActionableDynamicQuery` framework runs a query against bookmarks folders and entries to find ones which should be exported to the LAR file.

The `-ActionableDynamicQuery` classes are automatically generated by Service Builder and are available in your application's local service. It queries the database searching for certain Staging-specific parameters (e.g., `createDate` and `modifiedDate`), and based on those parameters, finds a list of exportable records from the staged model data handler.

The `doImportData` queries for Bookmark entry and folder data in the imported LAR file that should be added to the database. This is done by extracting XML elements from the LAR file by using utility methods from the StagedModelDataHandlerUtil class. The extracted elements tell Liferay DXP what data to import from the LAR file.

8. Add a method that deletes the portlet's data. The Staging framework has an option called *Delete Portlet Data Before Importing* that lets the user delete portlet data before importing any new data. The `doDeleteData(...)` method is called to execute this deletion operation.

```
@Override
protected PortletPreferences doDeleteData(
        PortletDataContext portletDataContext, String portletId,
        PortletPreferences portletPreferences)
    throws Exception {

    if (portletDataContext.addPrimaryKey(
            BookmarksPortletDataHandler.class, "deleteData")) {

        return portletPreferences;
    }

    _bookmarksEntryLocalService.deleteEntries(
        portletDataContext.getScopeGroupId(),
        BookmarksFolderConstants.DEFAULT_PARENT_FOLDER_ID);

    _bookmarksFolderLocalService.deleteFolders(
        portletDataContext.getScopeGroupId());

    return portletPreferences;
}
```

   This method can also return a modified version of the portlet preferences if it contains references to data that no longer exists.

9. Add a method that counts the number of affected entities based on the current export or staging process:

```
@Override
protected void doPrepareManifestSummary(
        PortletDataContext portletDataContext,
        PortletPreferences portletPreferences)
    throws Exception {

    ActionableDynamicQuery entryExportActionableDynamicQuery =
        _bookmarksEntryLocalService.
            getExportActionableDynamicQuery(portletDataContext);

    entryExportActionableDynamicQuery.performCount();

    ActionableDynamicQuery folderExportActionableDynamicQuery =
        _bookmarksFolderLocalService.
            getExportActionableDynamicQuery(portletDataContext);

    folderExportActionableDynamicQuery.performCount();
}
```

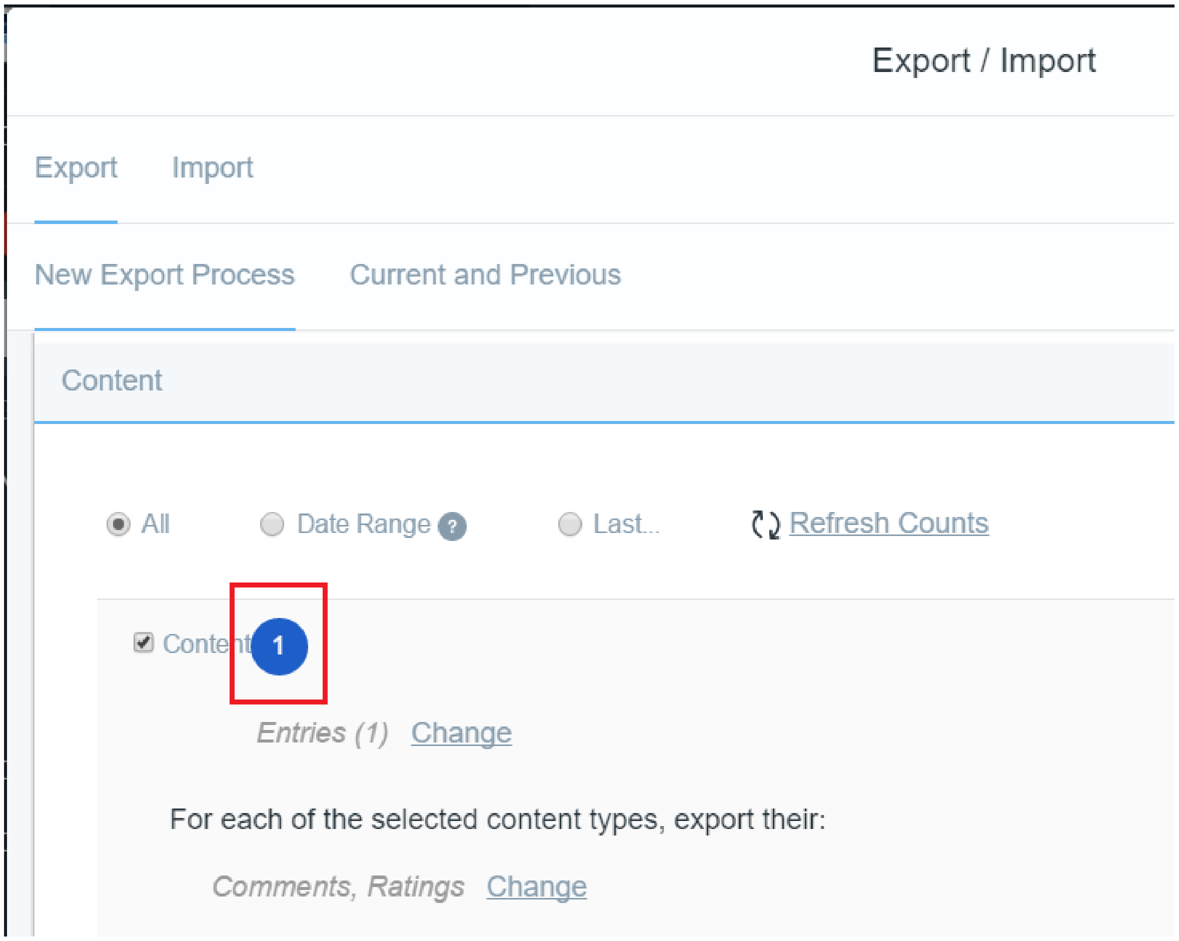


Figure 88.6: The number of modified Bookmarks entities are displayed in the Export UI.

This number is displayed in the Export and Staging UI. Note that since the Staging framework traverses the entity graph during export, the built-in components provide an approximate value in some cases.

10. Set the XML schema version for the XML files included in your exported LAR file:

```
public static final String SCHEMA_VERSION = "1.0.0";

@Override
public String getSchemaVersion() {
    return SCHEMA_VERSION;
}
```

Awesome! You've set up your portlet data handler and your application can now support the Export/Import framework and display a UI for it. The next step for supporting data handlers in your app is to implement staged model data handlers for your staged models. You'll do this next.

Staged Model Data Handlers

Now that your application has a portlet data handler and staged models, you'll create the staged model data handlers. The Bookmarks application has two staged models: entries and folders. Creating data handlers for these two entities is similar, so you'll examine how this is done for Bookmark entries.

1. Create a `-StagedModelDataHandler` class in the same folder as its portlet data handler class. For Bookmarks, the `BookmarksEntryStagedDataHandler` class resides in the `com.liferay.bookmarks.service` module's `com.liferay.bookmarks.internal.exportimport.data.handler` package. The staged model data handler class should extend the BaseStagedModelDataHandler class and the entity type should be specified as its parameter. You can see how this was done for the `BookmarksEntryStagedModelDataHandler` class below:

   ```
   public class BookmarksEntryStagedModelDataHandler
       extends BaseStagedModelDataHandler<BookmarksEntry> {
   ```

2. Create an `@Component` annotation section above the class declaration. This annotation is responsible for registering the class as a staged model data handler similar to the portlet data handler.

   ```
   @Component(immediate = true, service = StagedModelDataHandler.class)
   ```

   The `immediate` element directs the container to activate the component immediately once its provided module has started. The `service` element should point to the `StagedModelDataHandler.class` interface.

```
**Note:** In previous versions of Liferay DXP, you had to register the staged
model data handler in a portlet's `liferay-portlet.xml` file. The
registration process is now completed automatically by OSGi using the
`@Component` annotation.
```

---

3. Create a getter and setter method for the local service of the staged model for which you want to provide a data handler:

   ```
   @Override
   protected BookmarksEntryLocalService getBookmarksEntryLocalService() {
       return _bookmarksEntryLocalService;
   }

   @Reference(unbind = "-")
   protected void setBookmarksEntryLocalService(
       BookmarksEntryLocalService bookmarksEntryLocalService) {

       _bookmarksEntryLocalService = bookmarksEntryLocalService;
   }

   private BookmarksEntryLocalService _bookmarksEntryLocalService;
   ```

   These methods are used to link this data handler with the staged model for bookmark entries.

   Important: Liferay DXP's official Bookmarks app does not use local services in its staged model data handlers; instead, it uses the `StagedModelRepository` framework. This is not recommended for custom applications, however; it's only intended for internal Liferay applications at this time. Because of this, the code in this tutorial has been modified to highlight the recommended way for custom apps.

4. You must provide the class names of the models the data handler tracks. You can do this by overriding the StagedModelDataHandler's `getClassnames()` method:

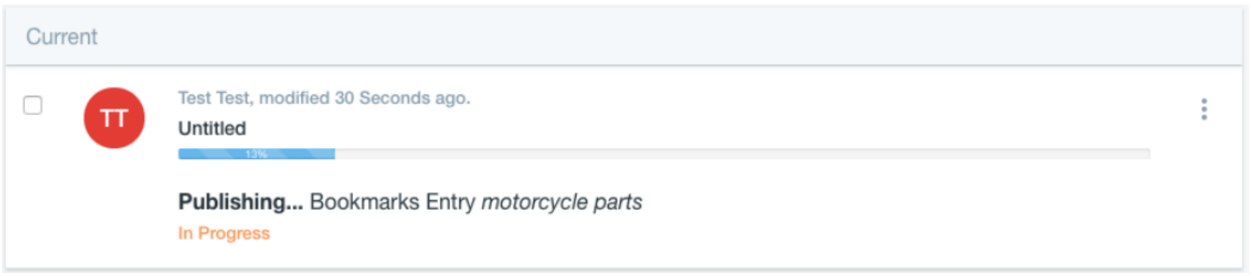
```
public static final String[] CLASS_NAMES = {BookmarksEntry.class.getName()};

@Override
public String[] getClassNames() {
    return CLASS_NAMES;
}
```

As a best practice, you should have one staged model data handler per staged model. It's possible to use multiple class types, but this is not recommended.

5. Add a method that retrieves the staged model's display name:

```
@Override
public String getDisplayName(BookmarksEntry entry) {
    return entry.getName();
}
```

The display name is presented with the progress bar during the export/import process.



Test Test, modified 30 Seconds ago.

Untitled

Publishing... Bookmarks Entry *motorcycle parts*

In Progress

Figure 88.7: Your staged model data handler provides the display name in the Export/Import UI.

6. A staged model data handler should ensure everything required for its operation is also exported. For example, in the Bookmarks application, an entry requires its folder to keep the folder structure intact. Therefore, the folder should be exported first followed by the entry.

   Add methods that import and export your staged model and its references.

```
@Override
protected void doExportStagedModel(
        PortletDataContext portletDataContext, BookmarksEntry entry)
    throws Exception {

    if (entry.getFolderId() ≠
            BookmarksFolderConstants.DEFAULT_PARENT_FOLDER_ID) {

        StagedModelDataHandlerUtil.exportReferenceStagedModel(
            portletDataContext, entry, entry.getFolder(),
            PortletDataContext.REFERENCE_TYPE_PARENT);
    }

    Element entryElement = portletDataContext.getExportDataElement(entry);

    portletDataContext.addClassedModel(
        entryElement, ExportImportPathUtil.getModelPath(entry), entry);
}

@Override
protected void doImportStagedModel(
        PortletDataContext portletDataContext, BookmarksEntry entry)
    throws Exception {

    Map<Long, Long> folderIds =
        (Map<Long, Long>)portletDataContext.getNewPrimaryKeysMap(
            BookmarksFolder.class);

    long folderId = MapUtil.getLong(
        folderIds, entry.getFolderId(), entry.getFolderId());

    ServiceContext serviceContext =
        portletDataContext.createServiceContext(entry);

    BookmarksEntry importedEntry = null;

    if (portletDataContext.isDataStrategyMirror()) {

        BookmarksEntry existingEntry =
            _bookmarksEntryLocalService. fetchBookmarksEntryByUuidAndGroupId(
                entry.getUuid(), portletDataContext.getScopeGroupId());

        if (existingEntry == null) {

            serviceContext.setUuid(entry.getUuid());
            importedEntry = _bookmarksEntryLocalService.addEntry(
              userId, portletDataContext.getScopeGroupId(), folderId, entry.getName(), entry.getUrl(), entry.getDescription(), serviceContext);
        }
        else {
            importedEntry = _bookmarksEntryLocalService.updateEntry(
                userId, existingEntry.getEntryId(), portletDataContext.getScopeGroupId(), folderId, entry.getName(), entry.getUrl(), entry.getDescri
        }
    }
    else {
        importedEntry = _bookmarksEntryLocalService.addEntry(userId, portletDataContext.getScopeGroupId(), folderId,entry.getName(), entry.getUrl(),
    }

    portletDataContext.importClassedModel(entry, importedEntry);
}
```

The doExportStagedModel method retrieves the Bookmark entry's data element from the PortletDataContext and then adds the class model characterized by that data element to the PortletDataContext. The PortletDataContext is used to populate the LAR file with your application's data during the export process. Note that once an entity has been exported, subsequent calls to the export method won't actually repeat the export process multiple times, ensuring optimal performance.

An important feature of the import process is that all exported reference elements in the Bookmarks example are automatically imported when needed. The doImportStagedModel method does not need to import the reference elements manually; it must only find the new assigned ID for the folder before importing the entry.

The PortletDataContext keeps this information and a slew of other data up-to-date during the import process. The old ID and new ID mapping can be reached by using the portletDataContext.getNewPrimaryKeysMap() method as shown in the example. The method proceeds with checking the import mode (e.g., *Copy As New* or *Mirror*) and depending on the process configuration and existing environment, the entry is either added or updated.

7. When importing a LAR that specifies a missing reference, the import process expects the reference to be available and must validate that it's there. You must add a method that maps the missing reference ID from the export to the existing ID during import.

   For example, suppose you export a FileEntry as a missing reference with an ID of 1. When importing that information, the LAR only provides the ID but not the entry itself. Therefore, during the import process, the Data Handler framework searches for the entry to replace, but the entry to replace has a different ID of 2. You must provide a method that maps these two IDs so the import process can recognize the missing reference.

```
@Override
protected void doImportMissingReference(
        PortletDataContext portletDataContext, String uuid, long groupId,
        long entryId)
    throws Exception {

    BookmarksEntry existingEntry = fetchMissingReference(uuid, groupId);

    if (existingEntry == null) {
        return;
    }

    Map<Long, Long> entryIds =
        (Map<Long, Long>)portletDataContext.getNewPrimaryKeysMap(
            BookmarksEntry.class);

    entryIds.put(entryId, existingEntry.getEntryId());
}
```

This method maps the existing staged model to the old ID in the reference element. When a reference is exported as missing, the Data Handler framework calls this method during the import process and updates the new primary key map in the portlet data context.

Fantastic! You've created a data handler for your staged model. The Export/Import framework can now track your entity's behavior and data.

With the ability to track entity data, your application is available during the Export/Import and Staging processes.

88.7 Initiating New Processes with ExportImportConfiguration Objects

Liferay DXP's Staging and Export/Import features are the building blocks for creating, managing, and publishing a site. These features can be accessed in your Portal's *Publishing Tools* menu. You can also, however, start these processes programatically. This lets you provide new interfaces or mimic the functionality of these features in your own application.

Providing the ability to stage your application's assets makes using your application much more site administrator-friendly. Your new assets no longer have to be saved somewhere off-site until they're ready to be published. You can publish them to a staging environment, test their usability, and save them to a page. Once the time is right for publishing, you can publish your application's assets to the live site with one mouse click. The export/import feature offers similar conveniences: if you want to export your application's assets to use in another place or you need to clear its data but save a copy you can implement the export feature. Implementing the import feature lets you bring your assets/data back into your application.

To initiate a export/import or staging process, you must pass in an `ExportImportConfiguration` object. This object encapsulates many parameters and settings that are required while the export/import is running. Having one single object with all your necessary data makes executing these frameworks quick and easy.

When you want to implement, for example, export, you must call services offered by the `ExportImportService` interface. All the methods in this interface require an `ExportImportConfiguration` object. Liferay DXP provides a way to generate these configuration objects, so you can easily pass them in your service methods.

It's also important to know that `ExportImportConfiguration` is a Portal entity, similar to `User` or `Group`. This means that the `ExportImportConfiguration` framework offers local and remote services, models, persistence classes, and more.

In this tutorial, you'll learn about the `ExportImportConfiguration` framework and how you can take advantage of provided services and factories to create these controller obects. Once they're created, you can easily impment whatever import/export functionality you need.

Your first step is to create an `ExportImportConfiguration` object and use it to initiate your custom export/import or staging process.

1. Use the Export Import Configuration factory classes to build your `ExportImportConfiguration` object. Below is a common way to do it:

   ```
   Map<String, Serializable> exportLayoutSettingsMap =
       ExportImportConfigurationSettingsMapFactory.
           buildExportLayoutSettingsMap(...);

   ExportImportConfiguration exportImportConfiguration =
       exportImportConfigurationLocalService.
           addDraftExportImportConfiguration(
               user.getUserId(),
               ExportImportConfigurationConstants.TYPE_EXPORT_LAYOUT,
               exportLayoutSettingsMap);
   ```

   This example uses the ExportImportConfigurationSettingsMapFactory to create a layout export settings map. Then this map is used as a parameter to create an `ExportImportConfiguration` by calling an *add* method in the entity's local service interface. The ExportImportConfigurationLocalService provides several useful methods to create and modify your custom `ExportImportConfiguration`.

   The `ExportImportConfigurationSettingsMapFactory` provides many `build` methods to create settings maps for various scenarios, like importing, exporting, and publishing layouts and portlets. For examples of this particular scenario, you can reference UserGroupLocalServiceImpl.exportLayouts(...) and GroupLocalServiceImpl.addDefaultGuestPublicLayoutsByLAR(...).

   There are two other important factories provided by this framework that are useful during the creation of `ExportImportConfiguration` objects:

   - ExportImportConfigurationFactory: This factory is used to build `ExportImportConfiguration` objects used for default local/remote publishing.
   - ExportImportConfigurationParameterMapFactory: This factory is used to build parameter maps, which are required during export/import and publishing.

2. Call the appropriate service to initiate the export/import or staging process. There are two important service interfaces that you can use in the cases of exporting, importing, and staging: ExportImportLocalService and StagingLocalService. In the previous step's example code snippet, you created an `ExportImportConfiguration` object intended for exporting layouts. Here's how to initiate that process:

   ```
   files[0] = exportImportLocalService.exportLayoutsAsFile(
       exportImportConfiguration);
   ```

   By calling this interface's method, you're exporting layouts from Portal into a `java.io.File` array. Notice that your `ExportImportConfiguration` object is the only needed parameter in the method. Your configuration object holds all the required parameters and settings necessary to export your layouts from Portal. Although this example code resides in Liferay DXP, you could easily use this framework from your own plugin or module.

```
**Note:** If you're not calling the export/import or staging service methods
from an OSGi module, you should not use the interface. The Liferay
OSGi container automatically handles interface referencing, which is why
using the interface is permitted for modules. If you're calling
export/import or staging service methods outside of a module, you should use
their service Util classes (e.g., `ExportImportLocalServiceUtil`).
```

It's that easy! To start your own export/import or staging process, you must create an `ExportImportConfiguration` object using a combination of the three provided `ExportImportConfiguration` factories. Once you have your configuration object, provide it as a parameter in one of the many service methods available to you by the Export/Import or Staging interfaces to begin your desired process.

88.8 Using the Export/Import Lifecycle Listener Framework

The `ExportImportLifecycleListener` framework allows developers to write code that listens for certain staging or export/import events during the publication process. The staging and export/import processes have many behind-the-scenes events that you cannot listen to by default. Some of these, like export successes and import failures, may be events on which you'd want to take some action. You also have the ability to listen for processes comprised of many events and implement custom code when these processes are initiated. Here is a short list of events you can listen for:

- Staging has started
- A portlet export has failed
- An entity export has succeeded

The concept of listening for export/import and staging events sounds cool, but you may be curious as to why listening for certain events is useful. Listening for events can help you know more about your application's state. Suppose you'd like a detailed log of when certain events occur during an import process. You could configure a listener to listen for certain import events you're interested in and print information about those events to your console when they occur.

Liferay DXP uses this framework by default in several cases. For instance, Liferay clears the cache when a web content import process finishes. To accomplish this, the lifecycle listener framework listens for an event that specifies that a web content import process has completed. Once that event occurs, there is an event listener that automatically clears the cache. You could implement this sort of functionality yourself for any Portal event. You can listen for a specific event and then complete an action based on when that event occurs. For a list of events you can listen for during Export/Import and Staging processes, see ExportImportLifecycleConstants.

Some definitions are in order:

Events are particular actions that occur during processing.

Processes are longer running groups of events.

In this tutorial, you'll learn how to use the `ExportImportLifecycleListener` framework to listen for processes/events during the staging and export/import lifecycles.

Listening to Lifecycle Events

To begin creating your lifecycle listener, you must create a module. Follow the steps below:

1. Create an OSGi module.

2. Create a unique package name in the module's `src` directory and create a new Java class in that package. To follow naming conventions, begin the class name with the entity or action name you're processing, followed by *ExportImportLifecycleListener* (e.g., `LoggerExportImportLifecycleListener`).

3. You must extend one of the two Base classes provided with the Export/Import Lifecycle Listener framework: BaseExportImportLifecycleListener or BaseProcessExportImportLifecycleListener. To choose, you'll need to consider what parts of a lifecycle you want to listen for.

   Extend the `BaseExportImportLifecycleListener` class if you want to listen for specific *events* during a lifecycle. For example, you may want to write custom code if a layout export fails.

   Extend the `BaseProcessExportImportLifecycleListener` class if you want to listen for *processes* during a lifecycle. For example, you may want to write custom code if a site publication fails. Keep in mind that a process usually consists of many individual events. Methods provided by this base class are only run once when the desired process action occurs.

4. Directly above the class's declaration, insert the following annotation:

   ```
   @Component(immediate = true,
              service = ExportImportLifecycleListener.class)
   ```

   This annotation declares the implementation class of the component and specifies that the portal should start the module immediately.

5. Specify the methods you want to implement in your class.

Once you've successfully created your export/import lifecycle listener module, generate the module's JAR file and copy it to your Portal's `osgi/modules` directory. Once your module is installed and activated in your Portal's service registry, your lifecycle listener is ready for use in your Portal instance.

If you're still thirsting for more information on this framework, you're in luck! Here's an example, using the LoggerExportImportLifecycleListener. This listener extends the `BaseExportImportLifecycleListener`, so you should immediately know that it deals with lifecycle events.

The first method `isParallel()` determines whether your listener should run in parallel with the import/export process, or if the calling method should stop, execute the listener, and return to where the event was fired after the listener has finished. The next method is the `onExportImportLifecycleEvent(...)` method, which consumes the lifecycle event and passes it through the base class's method (as long as Debug mode is not enabled).

Each remaining method is called to print logging information for the user. For example, when a layout export starts, succeeds, or fails, logging information directly related to that event is printed. In summary, the `LoggerExportImportLifecycleListener` uses the lifecycle listener framework to print messages to the log when an export/import event occurs. Other good examples of event lifecycle listeners are CacheExportImportLifecycleListener and JournalCacheExportImportLifecycleListener.

For an example of a lifecycle listener extending the `BaseProcessExportImportLifecycleListener` class, inspect the ExportImportProcessCallbackLifecycleListener class. Instead of listening for lifecycle events, this class only listens for process actions.

Terrific! You learned about the Export/Import Lifecycle Listener framework, and you've learned how to create your own listener for events/processes that occur during export/import of your portal's content.

Chapter 89

Configuration

Many applications provide users a way of setting preferences for their use. In Liferay DXP, this could be as simple as setting a location for a weather display or as complex as settings for a mail or a time sheet application.

The Portlet standard defines an API for portlet preferences that can be used for this sort of thing, but it's limited and for that reason isn't used much. Instead, many developers have tended to create their own methods for configuring their applications.

But that isn't necessary anymore: now there's a configuration API that's easy to use and full-featured. It's used throughout Liferay DXP's applications, and because we like it, we think you'll like it too.

The following tutorials show you how to use it.

89.1 Making Your Applications Configurable

This tutorial explains how to make your applications configurable. It starts with basic configuration and then covers some advanced use cases.

Note that the methods described here are not mandatory. You can make your applications configurable using any other mechanism that you're already familiar with. We have found, however, that the method described below provides the greatest benefit with the least amount of effort.

Note: To quickly see a working application configuration, deploy the `configuration-action` Blade sample and navigate to System Settings (*Control Panel → Configuration → System Settings*). In the Other category, click the *Message display configuration* entry.

Add the *Blade Message Portlet* to a page to test your configuration choices.

Fundamentals

While you don't need to know much to make your applications configurable, understanding a few key concepts helps you achieve a higher degree of configurability with little effort.

- Typed Configuration: The method described here uses *typed* configuration. This means that the application configuration isn't just a list of key-value pairs. The values can have types, like `Integer`, a list of `Strings`, a URL, etc. It's even possible to use your own custom types, although that's beyond the scope of this tutorial. Typed configurations are easier to use than untyped configurations, and they prevent many programmatic errors. Related to this, the configuration options should be programmatically explicit, so that developers can use autocomplete in modern IDEs to find out all of the configuration options of a given application or one of its components.

- Modularity: In Liferay DXP, applications are *modular* and built as a collection of lightweight components. A *component* is just a class that has the `@Component` annotation, often along with a set of properties to provide metadata. The configuration mechanisms described here leverage the concept of components.

- Configuration Scope: If your application must support different configurations at different scopes, the APIs described below handle most of the burden for you. It's still important, however, for you to understand the term *configuration scope*. Here are the most common configuration scopes that Liferay applications can have:

    1. **System:** configuration that is unique for the complete installation of the application.

    2. **Virtual Instance:** configuration that can vary per virtual instance.

    3. **Site:** configuration that can vary per Liferay site.

    4. **Portlet Instance:** applicable for applications that are placed on a page (i.e., portlets). Each placement (instance) of the application on the page can have a different configuration.

Enough with the conceptual stuff. You're ready to get started with some code. If you already had a portlet or service that was configurable using the traditional mechanisms of Liferay Portal 6.2 and before, refer to the Transitioning from Portlet Preferences to the Configuration API tutorial.

Making Your Application Configurable

There's a minimal amount of code you need to write to make your application configurable the Liferay DXP way. First, you'll learn how to create a configuration at the system scope.

First create a Java interface to represent the configuration and its default values. Using a Java interface allows for an advanced type system for each configuration option. Here is an example of such an interface:

```
@Meta.OCD(id = "com.liferay.docs.exampleconfig.ExampleConfiguration")
public interface ExampleConfiguration {

    @Meta.AD(
        deflt = "blue",
        required = false
    )
    public String favoriteColor();

    @Meta.AD(
       deflt = "red|green|blue",
       required = false
    )
    public String[] validColors();

    @Meta.AD(required = false)
    public int itemsPerPage();

}
```

As you can see, you are using two Java annotations to provide some metadata about the configuration. Here is what they do:

1. **Meta.OCD** Registers this class as a configuration with a specific id. **The ID must be the fully qualified configuration class name.**
2. **Meta.AD** Specifies the default value of a configuration field as well as whether it's required or not. Note that if you set a field as required and don't specify a default value, the system administrator must specify a value in order for your application to work properly. Use the `deflt` property to specify a default value.

The fully-qualified class name of the `Meta` class referred to above is `aQute.bnd.annotation.metatype.Meta`. For more information about this class and the `Meta.OCD` and `Meta.AD` annotations, please refer to this bnd documentation: http://bnd.bndtools.org/chapters/210-metatype.html. In order to use the `Meta.OCD` and `Meta.AD` annotations in your modules, you must specify a dependency on the bnd library. We recommend using bnd version 3. Here's an example of how to include this dependency in a Gradle project:

```
dependencies {
    compile group: "biz.aQute.bnd", name: "biz.aQute.bndlib", version: "3.1.0"
}
```

Note: The annotations `@Meta.OCD` and `@Meta.AD` are part of the bnd library but have also been included as part of the OSGi standard version R6 with the names `@ObjectClassDefinition` and `@AttributeDefinition`. However, Liferay still uses the bnd annotations since the standard annotations are not available at runtime, which is necessary for some of the Liferay specific features described below. For the basic usage (the one described in this section) the standard annotations can be used safely.

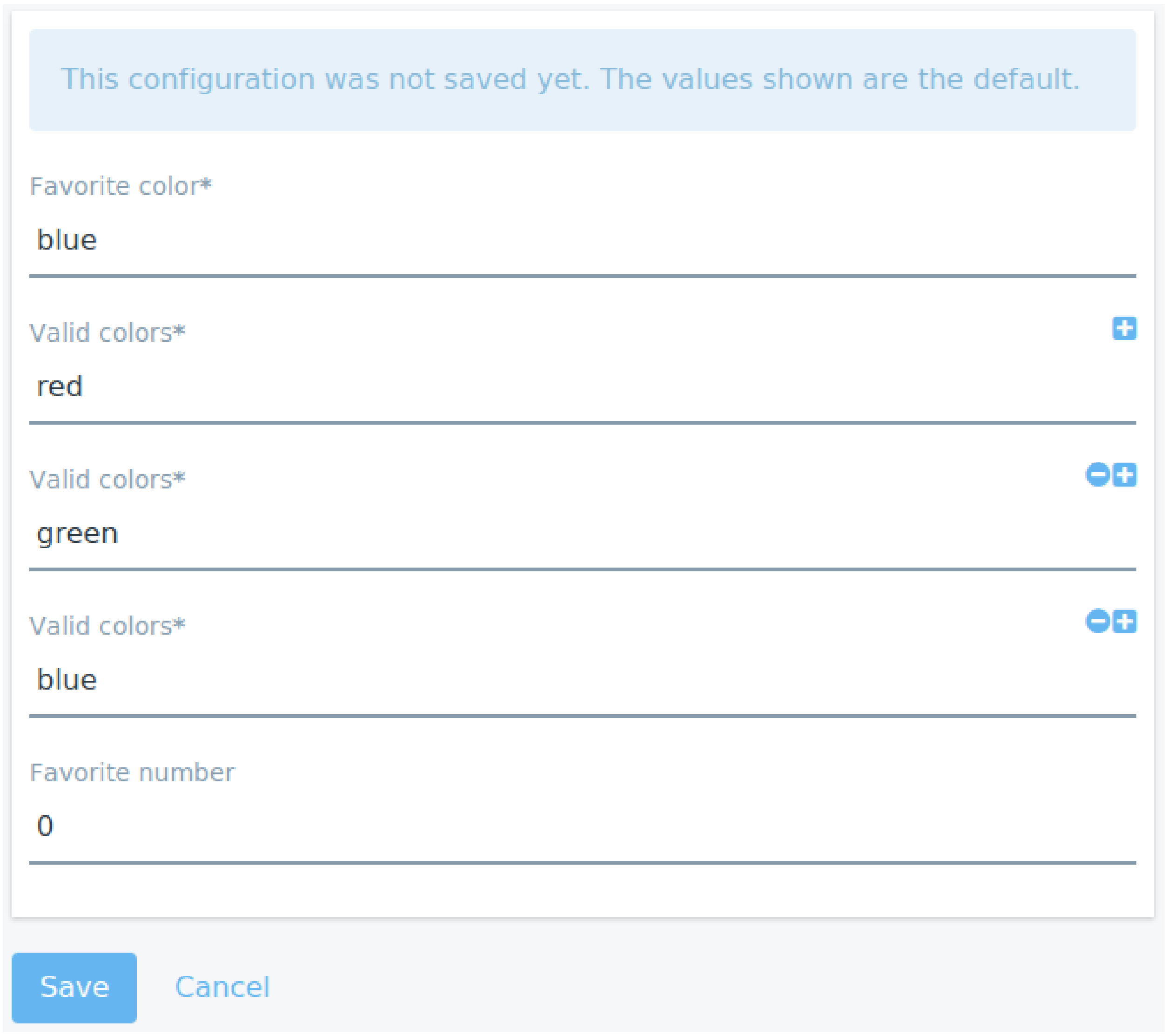
Add the following line to your project's `bnd.bnd` file:

```
-metatype: *
```

This line lets bnd use your configuration interface to generate an XML configuration file. With this information, Liferay already knows a lot about your application's configuration options. In fact, it knows enough to generate a user interface automatically. Cool, isn't it?

Even if you agree that this is pretty cool, you might be wondering how to read the configuration from your application's code. It's actually quite easy. Here's a simple example:

```
@Component(configurationPid = "com.liferay.docs.exampleconfig.ExampleConfiguration")
public class MyAppManager {

    public String getFavoriteColor(Map colors) {
       return colors.get(_configuration.favoriteColor());
    }

    @Activate
    @Modified
    protected void activate(Map<String, Object> properties) {
        _configuration = ConfigurableUtil.createConfigurable(
        ExampleConfiguration.class, properties);
    }

    private volatile ExampleConfiguration _configuration;

}
```


This configuration was not saved yet. The values shown are the default.

Favorite color*

blue

Valid colors*

red

Valid colors*

green

Valid colors*

blue

Favorite number

0

Figure 89.1: Navigate to the Control Panel and then click on *Configuration* → *System Settings*. Then click on *Other*, find the *Example configuration* link, and click on it.

Here are the most relevant aspects of this example:

1. This class is a component, specified with the `@Component` annotation.
2. This component uses the configuration with the ID `com.liferay.docs.exampleconfig.ExampleConfiguration`. As a result, this method is invoked when the application starts (due to the `@Activate` annotation) and whenever the configuration is modified (due to the `@Modified` annotation).
3. The `activate()` method uses the method `ConfigurableUtil.createConfigurable()` to convert a map of properties with the configuration to our typed class, which is easier to handle.
4. The configuration is stored in a `volatile` field. Don't forget to make it `volatile` or you'll run into weird problems.

Note: The bnd library also provides a class called `aQute.bnd.annotation.metatype.Configurable` with a `createConfigurable()` method. You can use that instead of Liferay's `com.liferay.portal.configuration.metatype.bnd.util.Co` without any problems. Liferay's developers created the `ConfigurableUtil` class to improve the performance of bnd's implementation, and it's used in internal code. Feel free to use whichever method you prefer.

That's it. With very few lines of code, you have a configurable application that dynamically changes its configuration, has an auto-generated UI, and uses a simple API to access the configuration.

Accessing Your Configuration in a JSP Portlet Application

In Liferay DXP it's very common to read a configuration from a portlet class. If the portlet is a JSP portlet, the configuration object can be added to the request object so that configurations can be read from the JSPs that comprise the application's view layer. In this section, you'll see an example of reading a configuration from a portlet class, adding it to the request, and reading from the view layer. The import statements are included in the code snippets so that you can see the fully qualified class names of all the classes that are used.

Note: There's shortcut method for obtaining a portlet instance configuration. The method described in this section takes a straightforward approach that does not use this shortcut. See the Accessing the Portlet Instance Configuration Through the PortletDisplay section below to learn about the shorter method.

```
package com.liferay.docs.exampleconfig;

import java.io.IOException;
import java.util.Map;

import javax.portlet.Portlet;
import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Modified;

import com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet;

import com.liferay.portal.configuration.metatype.bnd.util.ConfigurableUtil;

@Component(
    configurationPid = "com.liferay.docs.exampleconfig.ExampleConfiguration",
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.security-role-ref=power-user,user",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.resource-bundle=content.Language"
    },
    service = Portlet.class
)
public class ExampleConfigPortlet extends MVCPortlet {

    @Override
    public void doView(RenderRequest renderRequest,
        RenderResponse renderResponse) throws IOException, PortletException {

        renderRequest.setAttribute(
            ExampleConfiguration.class.getName(), _configuration);

        super.doView(renderRequest, renderResponse);
    }


    public String getFavoriteColor(Map colors) {
        return (String) colors.get(_configuration.favoriteColor());
    }

    @Activate
    @Modified
    protected void activate(Map<String, Object> properties) {
        _configuration = ConfigurableUtil.createConfigurable(
        ExampleConfiguration.class, properties);
    }

    private volatile ExampleConfiguration _configuration;

}
```

The main difference between this example and the first one is that this class is a portlet class and it sets the configuration object as a request attribute in its `doView()` method. To read configuration values from a JSP, first add these imports to your `init.jsp` file:

```
<%@ page import="com.liferay.docs.exampleconfig.ExampleConfiguration" %>
<%@ page import="com.liferay.portal.kernel.util.GetterUtil" %>
```

In a JSP portlet application, your full `init.jsp` file should at least have contents like this:

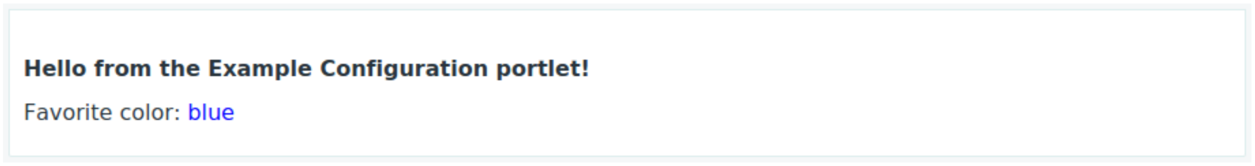
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>

<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
<%@ taglib uri="http://liferay.com/tld/portlet" prefix="liferay-portlet" %>
<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>

<%@ page import="com.liferay.docs.exampleconfig.ExampleConfiguration" %>
<%@ page import="com.liferay.portal.kernel.util.GetterUtil" %>

<portlet:defineObjects />
<liferay-theme:defineObjects />
```

It's a Liferay convention that all JSP imports in your application should go in an `init.jsp` file. All of your application's other JSPs import `init.jsp`. This convention ensures that you only have to manage JSP dependencies in a single file.

Next, obtain the configuration object from request object and read the desired configuration value from it. Here's an example `view.jsp` file that does this:

```
<%@ include file="/init.jsp" %>

<p>
    <b>Hello from the Example Configuration portlet!</b>
</p>

<%
ExampleConfiguration configuration = (ExampleConfiguration) GetterUtil.getObject(
    renderRequest.getAttribute(ExampleConfiguration.class.getName()));

String favoriteColor = configuration.favoriteColor();
%>

<p>Favorite color: <span style="color: <%= favoriteColor %>;"><%= favoriteColor %></span></p
```

The example code here would make the application display a message like this:


Hello from the Example Configuration portlet!

Favorite color: blue

Figure 89.2: Here, the Example Configuration portlet's `view.jsp` is rendered. This JSP reads the value of the `favoriteColor` configuration and displays it.

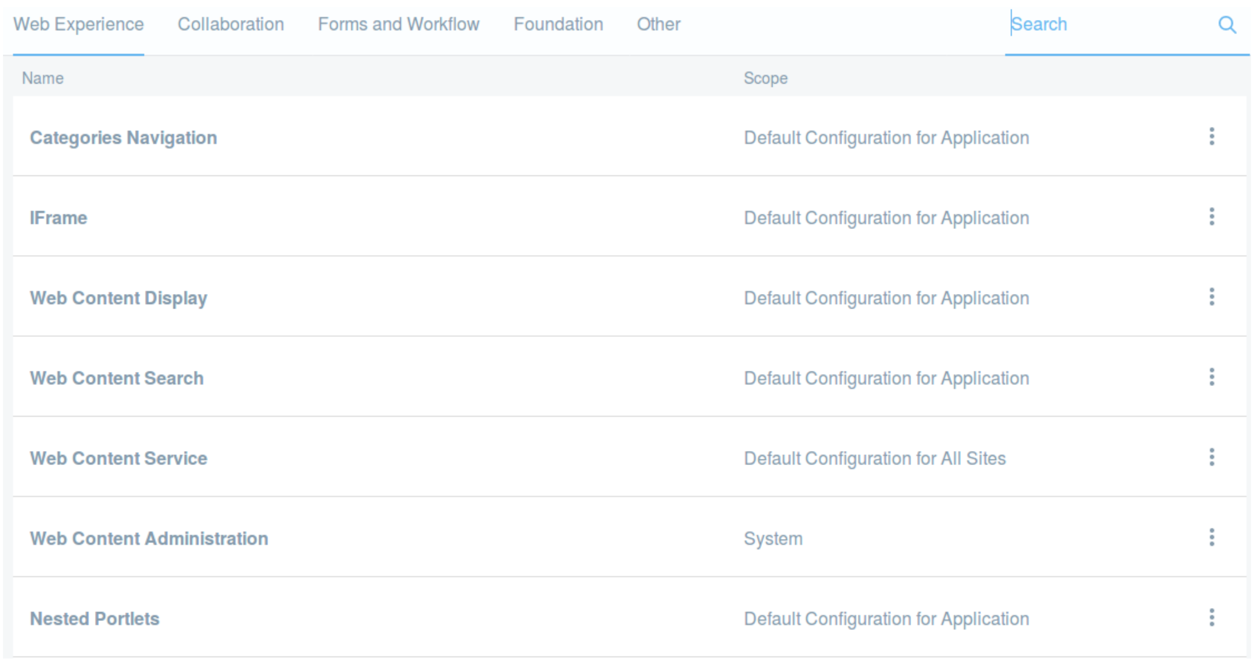
```
Favorite color: blue
```

The word 'blue' should be written in blue text. Note that 'blue' is displayed by default since you specified it as the default in your `ExampleConfiguration` interface. If you go to *System* → *System Settings* → *Other* and click on the *Example configuration* link, you can find the `Favorite color` setting and change its value. Your application's JSP will reflect this update when you refresh the page.

Categorizing the Configuration

Because it's easy to make any application or service configurable, there are already lots of configuration options in Liferay DXP by default. If you've deployed custom applications and services to your portal, there will be even more. To make it easier for portal administrators to find the right configuration options, Liferay provides a mechanism for developers to specify a category in which the configuration will be shown in the auto-generated System Settings UI in the Control Panel.

Here's how the System Settings UI looks:



Name	Scope
Categories Navigation	Default Configuration for Application
IFrame	Default Configuration for Application
Web Content Display	Default Configuration for Application
Web Content Search	Default Configuration for Application
Web Content Service	Default Configuration for All Sites
Web Content Administration	System
Nested Portlets	Default Configuration for Application

Figure 89.3: Navigate to the Control Panel, click on *Configuration* and then *System Settings*. You'll find five categories of configurations, including Other. Click on any configuration to access a form through which the configuration values can be updated.

By default, the following configuration categories are defined:

1. Web Experience
2. Collaboration
3. Forms and Workflow
4. Foundation
5. Other

You can use any other category and it will be injected in alphabetical order after Platform. Other will always be shown last. In order to specify a category, you must use the `@ExtendedObjectClassDefinition` annotation as in the following example:

```
@ExtendedObjectClassDefinition(category = "platform")
@Meta.OCD(
    factory = true,
    id = "com.liferay.portal.ldap.configuration.SystemLDAPConfiguration",
    localization = "content/Language"
)
```

The fully qualified class name of the `@ExtendedObjectClassDefinition` class is `com.liferay.portal.configuration.metatype.`

Note: Currently, the infrastructure used by System Settings relies on the `configurationPid` being the same as the class name of the interface. If they don't match, it will not be able to provide any information provided through `ExtendedObjectClassConfiguration`.

The `@ExtendedObjectClassDefinition` annotation is distributed through the `com.liferay.portal.configuration.metatype` module, which you can configure as a dependency.

Supporting Different Configurations per Virtual Instance, Site, or Portlet Instance

When an application is deployed to Liferay, it's common to need different configurations depending on the scope. That means having different configurations for a given application per virtual instance (a.k.a. Company), site (a.k.a. Group), or portlet instance. Liferay DXP provides an easy way to achieve this with little effort through a new framework called the Configuration API that is based on the standard OSGi Configuration Admin API shown in the previous section.

Using the Configuration Provider

When using the Configuration Provider, instead of receiving the configuration directly, the class that wants to access it will need to receive a `ConfigurationProvider` from which to obtain the configuration. Additionally, you need to "register" your class.

Note: `ConfigurationProvider` is part of Liferay's kernel API so you don't need to add a new dependency to use it. However, its implementation is distributed as a module called `portal-configuration-module-configuration`, so you will need to make sure it is installed in order to use it.

Before using the `ConfigurationProvider`, register the configuration class by writing a class that implements `ConfigurationBeanDeclaration`. This class only has one method that returns the class of the interface you created in the previous section. By doing this, the system is able to keep track of any configuration changes as they happen. This makes requests for the configuration very fast.

Declare the configuration interface by creating a `ConfigurationBeanDeclaration` class:

```
@Component
public class RSSPortletInstanceConfigurationBeanDeclaration
    implements ConfigurationBeanDeclaration {

    @Override
    public Class getConfigurationBeanClass() {
        return RSSPortletInstanceConfiguration.class;
    }

}
```

Once you have created your `ConfigurationBeanDeclaration`, you can use a `ConfigurationProvider`. Here's how you can obtain a reference to it:

- For components:

  ```
  @Reference
  protected void setConfigurationProvider(ConfigurationProvider configurationProvider) {
      _configurationProvider = configurationProvider;
  }
  ```

- For Service Builder services:

  ```
  @ServiceReference(type = ConfigurationProvider.class)
  protected ConfigurationProvider configurationProvider;
  ```

- For Spring beans: It is possible to use the same mechanism as for Service Builder services (`@ServiceReference`). Check the documentation on how to integrate Spring beans with OSGi services for more details.

Later, the configuration can be obtained using one of the following methods of the provider:

- `getCompanyConfiguration()`: Used when you want to support different configurations per virtual instance. In this case, the configuration is usually entered by an admin through Control Panel → Configuration → Instance Settings. Since this UI is not automatically generated (yet) you will need to extend the UI with your own form.

- `getGroupConfiguration()`: Used when you want to support different configurations per site (or, if desired, per page scope). Usually this configuration is specified by an admin through the Configuration menu option in an app accessing through the site administration menu. That UI is developed as a portlet configuration view.

- `getPortletInstanceConfiguration()`: Used to obtain the configuration for an specific portlet instance. Most often you should not be using this directly and use the convenience method in `PortletDisplay` instead as shown below.

- `getSystemConfiguration`: Used to obtain the configuration for the system scope. These settings are specified by an admin via the System Settings application or with an OSGi configuration file.

Here are a couple real world examples from Liferay's source code:

```
JournalGroupServiceConfiguration configuration =
    configurationProvider.getGroupConfiguration(
        JournalGroupServiceConfiguration.class, groupId);

MentionsGroupServiceConfiguration configuration =
  _configurationProvider.getCompanyConfiguration(
     MentionsGroupServiceConfiguration.class, entry.getCompanyId());
```

Next, you'll learn how to access a portlet's configuration from outside of an OSGi component.

Accessing the Portlet Instance Configuration Through the PortletDisplay

Often it's necessary to access a portlet's settings from its JSPs or from Java classes that are not OSGi components. To make it easier to read the settings in these cases, a new method has been added to `PortletDisplay` (available as a request object). Here is an example of how to use it:

```
RSSPortletInstanceConfiguration rssPortletInstanceConfiguration =
    portletDisplay.getPortletInstanceConfiguration(
        RSSPortletInstanceConfiguration.class);
```

As you can see, it knows how to find the values and returns a typed bean containing them just by passing the configuration class.

Specifying the Scope of the Configuration

The `ExtendedObjectClassDefinition` annotation allows you to specify the scope of the configuration. This should match how the configuration object is retrieved through the provider (your choice). The valid options are:

- `Scope.GROUP`: for site scope
- `Scope.COMPANY`: For virtual instance scope
- `Scope.SYSTEM`: for system scope

Here is an example:

```
@ExtendedObjectClassDefinition(
    category = "productivity", scope = ExtendedObjectClassDefinition.Scope.GROUP
)
@Meta.OCD(
    id = "com.liferay.dynamic.data.lists.form.web.configuration.DDLFormWebConfiguration",
    localization = "content/Language", name = "%ddl.form.web.configuration.name"
)
public interface DDLFormWebConfiguration {
...
}
```

In Liferay DXP version 7.0, the scope property isn't used for anything other than making it appear in System Settings so that an administrator can change its value. In future releases, may have more purposes.

Summary

In this tutorial, you've learned how to make your applications configurable. Creating a simple configuration interface allows Liferay to auto-generate a configuration UI that's accessible via System Settings in the Control Panel. You've also learned how to categorize your configurations within System Settings, how read configuration settings in your application at runtime, how to support different configurations at different scopes, and how to reuse the same configuration class for different scenarios.

89.2 Implementing Configuration Actions

When developing an application, it's important to think about the different configuration options that your application should support. It's also important to think about how users should be able to access your application's configuration interface. Liferay DXP supports a flexible mechanism for configuring applications. You can read about it in the Making Your Applications Configurable tutorial. In this tutorial,

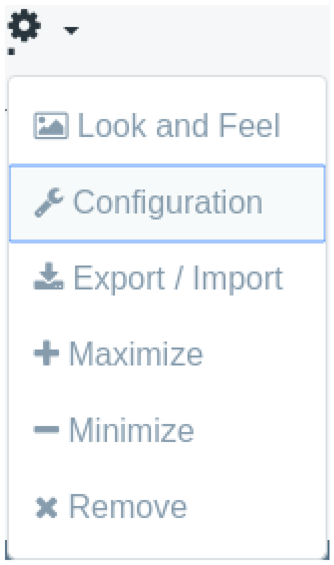


Figure 89.4: When a user clicks the gear icon and selects *Configuration*, the application's configuration action is invoked.

you'll learn to implement a configuration action. The configuration action is invoked when a user clicks on the gear icon and selects *Configuration*.

Liferay applications support a default configuration action. If you click on the gear icon of an application that has not been customized and then select *Configuration*, you'll find two standard tabs: Permissions and Sharing. These tabs provide standard options for configuring who can access your application and how you can make your application more widely available. If you follow the instructions in this tutorial, you'll learn how to create a Setup tab that allows custom configuration fields to be manipulated. To implement a configuration action, follow these steps:

1. Create an interface to represent your configuration
2. Implement your application class and add a reference to your configuration in your application class
3. Implement your configuration action class and add a reference to your configuration in your configuration action class
4. Implement the user interface for configuring your application

---

Note: To quickly see a working configuration action, deploy the `configuration-action` Blade sample and add the *Blade Message Portlet* to a page. Click the *Options* button (⋮) and select *Configuration*. Change the configuration options and save them to see them in action.

Let's get started.

Creating a Configuration Interface

As explained in the Making Your Applications Configurable tutorial, if you want to make your application configurable, you should create a Java interface to represent the configuration. Decorate your interface with the `@Meta.OCD` annotation and specify a unique ID using the annotation's `id` attribute. A common pattern is to use the fully qualified class name of the interface for the ID since fully qualified class names are unique. Create public methods to represent configuration fields and decorate the methods with the `@Meta.AD` annotation. The return type of the method specifies the type of the field. To specify a field's default value, use the annotation's `deflt` attribute. To specify that a field is optional, set `required=false`. For more information about the `Meta.OCD` and `Meta.AD` annotations, please see the bnd documentation. Here's a simple example:

```
package com.liferay.docs.exampleconfig.configuration;

import aQute.bnd.annotation.metatype.Meta;

@Meta.OCD(id = "com.liferay.docs.exampleconfig.configuration.ExampleConfiguration")
public interface ExampleConfiguration {

    @Meta.AD(required = false)
    public String favoriteColor();

}
```

Add the following line to your project's `bnd.bnd` file:

```
-metatype: *
```

This line lets bnd use your configuration interface to generate an XML configuration file. This lets Liferay DXP auto-generate a UI for your configuration in the System Settings area of the Control Panel. However, it's sometimes preferable for users to be able to access your configuration directly from the portlet without having to go to the Control Panel. In this tutorial, you'll learn how to facilitate this.

This sample configuration contains a single string field called `favoriteColor`.

Referencing Your Configuration From Your Application Class

As was also explained in the Making Your Applications Configurable tutorial, if you want a reference to the configuration in your application class, you need to declare the configuration as a `volatile` member variable, decorate your application class with the `@Component` annotation, specify the appropriate `configurationPid` in the `@Component` annotation, add an appropriately annotated `activate` method that instantiates the configuration variable, and add a public getter method for each configuration field. Here's a simple example that makes the sample configuration discussed earlier available to a portlet class:

```
package com.liferay.docs.exampleconfig.portlet;

import java.io.IOException;
import java.util.Map;

import javax.portlet.Portlet;
import javax.portlet.PortletException;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Modified;

import com.liferay.docs.exampleconfig.configuration.ExampleConfiguration;
import com.liferay.portal.configuration.metatype.bnd.util.ConfigurableUtil;
import com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet;

@Component(
    configurationPid =
    "com.liferay.docs.exampleconfig.configuration.ExampleConfiguration",
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.security-role-ref=power-user,user",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.resource-bundle=content.Language"
    },
    service = Portlet.class
)
public class ExampleConfigPortlet extends MVCPortlet {

        @Override
        public void doView(RenderRequest renderRequest,
                RenderResponse renderResponse) throws IOException, PortletException {

                renderRequest.setAttribute(
                        ExampleConfiguration.class.getName(),
                        _exampleConfiguration);

                super.doView(renderRequest, renderResponse);
        }

        public String getFavoriteColor(Map labels) {
                return (String) labels.get(_exampleConfiguration.favoriteColor());
        }

        @Activate
        @Modified
        protected void activate(Map<Object, Object> properties) {
                _exampleConfiguration = ConfigurableUtil.createConfigurable(
                        ExampleConfiguration.class, properties);
        }

        private volatile ExampleConfiguration _exampleConfiguration;

}
```

In this example, overriding the `doView` method is not strictly necessary. However, it's useful since adding the configuration to the request object before calling `super.doView` makes the configuration able to be read from the request by the application's JSPs.

Implementing a Configuration Action

To implement a configuration action, you should create a class that extends Liferay DXP's `DefaultConfigurationAction` class. Then you need to add a reference to your configuration the same way that you added such a reference to your application class. Declare the configuration as a `volatile` member variable, decorate your configuration action class with the `@Component` annotation, specify the appropriate `configurationPid` in the `@Component` annotation, add an appropriately annotated `activate` method that instantiates the configuration variable, and add a public getter method for each configuration field.

Next, you should specify `configurationPolicy = ConfigurationPolicy.OPTIONAL` in your class's `@Component` annotation. An optional configuration policy means that the component is created regardless of whether or not the configuration was set. You also need to specify the portlet to which your configuration action class applies. To do so, make the following specification in your class's `@Component` annotation:

```
property = {
    "javax.portlet.name=com_liferay_docs_exampleconfig_portlet_ExampleConfigPortlet"
},
```

Your component should be registered as a configuration action class so you should specify `service = ConfigurationAction.class` in your class's `@Component` annotation.

Next, you should override the `processAction` method so that it reads a URL parameter from the action request, sets the value as a portlet preference, and invokes the `processAction` method of the `SettingsConfigurationAction` ancestor class. Finally, you should override the `include` method so that it sets the configuration as an attribute of the HTTP servlet request and then invokes the include method of the BaseJSPSettingsConfigurationAction class. Here's an example:

```
package com.liferay.docs.exampleconfig.action;

import java.util.Map;

import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.PortletConfig;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.ConfigurationPolicy;
import org.osgi.service.component.annotations.Modified;

import com.liferay.docs.exampleconfig.configuration.ExampleConfiguration;
import com.liferay.portal.configuration.metatype.bnd.util.ConfigurableUtil;
import com.liferay.portal.kernel.portlet.ConfigurationAction;
import com.liferay.portal.kernel.portlet.DefaultConfigurationAction;
import com.liferay.portal.kernel.util.ParamUtil;

@Component(
    configurationPid = "com.liferay.docs.exampleconfig.configuration.ExampleConfiguration",
    configurationPolicy = ConfigurationPolicy.OPTIONAL,
    immediate = true,
    property = {
        "javax.portlet.name=com_liferay_docs_exampleconfig_portlet_ExampleConfigPortlet"
    },
    service = ConfigurationAction.class
)
public class ExampleConfigurationAction extends DefaultConfigurationAction {

    @Override
    public void processAction(
            PortletConfig portletConfig, ActionRequest actionRequest,
            ActionResponse actionResponse)
        throws Exception {

        String favoriteColor = ParamUtil.getString(actionRequest, "favoriteColor");
        setPreference(actionRequest, "favoriteColor", favoriteColor);

        super.processAction(portletConfig, actionRequest, actionResponse);
    }

    @Override
    public void include(
        PortletConfig portletConfig, HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse) throws Exception {

        httpServletRequest.setAttribute(
            ExampleConfiguration.class.getName(),
            _exampleConfiguration);

        super.include(portletConfig, httpServletRequest, httpServletResponse);
    }

    @Activate
    @Modified
    protected void activate(Map<Object, Object> properties) {
        _exampleConfiguration = ConfigurableUtil.createConfigurable(
            ExampleConfiguration.class, properties);
    }

    private volatile ExampleConfiguration _exampleConfiguration;
}
```

Now that your configuration action class has been created, you're ready to create a user interface for selecting configuration options and submitting the selections.

Implementing the User Interface For Configuring Your Application

When creating a JSP-based user interface, it's convenient to create an `init.jsp` page for your application. The `init.jsp` page should contain all of the imports, tag library declarations, and other page components are required by your other JSPs. Each of your other pages should import `init.jsp` so that you don't need to duplicate code. Liferay DXP follows this convention.

Here's an example `init.jsp` file:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>

<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
<%@ taglib uri="http://liferay.com/tld/portlet" prefix="liferay-portlet" %>
<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>

<%@ page import="com.liferay.docs.exampleconfig.configuration.ExampleConfiguration" %>
<%@ page import="com.liferay.portal.kernel.util.StringPool" %>
<%@ page import="com.liferay.portal.kernel.util.Validator" %>

<portlet:defineObjects />

<liferay-theme:defineObjects />

<%
    ExampleConfiguration exampleConfiguration =
        (ExampleConfiguration)
        renderRequest.getAttribute(ExampleConfiguration.class.getName());

    String favoriteColor = StringPool.BLANK;

    if (Validator.isNotNull(exampleConfiguration)) {
        favoriteColor =
            portletPreferences.getValue(
                "favoriteColor", exampleConfiguration.favoriteColor());
    }
%>
```

This JSP not only declares some tag libraries and imports some classes. It uses the `<portlet:defineObjects />` and `<liferay-theme:defineObjects />` tags to make certain variables available on the page. The scriptlet at the end of the file uses one of these variables, `renderRequest`, to get the configuration which was stored in the `renderRequest` by your portlet's `doView` method. Finally, the value of a specific field (`favoriteColor`) is read from the configuration.

The default view of your application is provided by your application's `view.jsp`. Your `view.jsp` should import your `init.jsp` so that the same tag libraries, imports, and variables are available on the page. Here's an example `view.jsp` file:

```
<%@ include file="/init.jsp" %>

<p>
    <liferay-ui:message key="com_liferay_docs_exampleconfig_portlet_ExampleConfigPortlet.caption"/>
</p>

<%
```

```
    boolean noConfig = Validator.isNull(favoriteColor);
%>

<c:choose>
    <c:when test="<%= noConfig %>">
        <p>
            Please select use the portlet configuration to select a favorite color.
        </p>
    </c:when>

    <c:otherwise>
        <p style="color: <%= favoriteColor %>">
            Favorite color: <%= favoriteColor %>!
        </p>
    </c:otherwise>
</c:choose>
```

This JSP simply checks whether or not the `favoriteColor` variable is empty. If it's empty, a message is displayed that tells the user that they need to select a favorite color in the portlet's configuration. If the `favoriteColor` variable is not empty, the name of the selected color is displayed in the selected color. Note: The value of the `com_liferay_docs_exampleconfig_portlet_ExampleConfigPortlet.caption` language key must be specified in your application's `Language.properties` file. The default location for this file is in the `content` package.

The configuration user interface of your application is provided by your application's `configuration.jsp` file. This interface is displayed on the Setup tab when a user clicks on your application's gear icon and then selects *Configuration*. As previously discussed, your `configuration.jsp` should import your `init.jsp` file. Here's an example `configuration.jsp` file:

```
<%@ include file="/init.jsp" %>

<%@ page import="com.liferay.portal.kernel.util.Constants" %>

<liferay-portlet:actionURL portletConfiguration="<%= true %>"
    var="configurationActionURL" />

<liferay-portlet:renderURL portletConfiguration="<%= true %>"
    var="configurationRenderURL" />

<aui:form action="<%= configurationActionURL %>" method="post" name="fm">

    <aui:input name="<%= Constants.CMD %>" type="hidden"
        value="<%= Constants.UPDATE %>" />

    <aui:input name="redirect" type="hidden"
        value="<%= configurationRenderURL %>" />

    <aui:fieldset>

        <aui:select name="favoriteColor" label="Favorite Color"
            value="<%= favoriteColor %>">
            <aui:option value="indigo">Indigo</aui:option>
            <aui:option value="blue">Blue</aui:option>
            <aui:option value="green">Green</aui:option>
            <aui:option value="yellow">Yellow</aui:option>
            <aui:option value="orange">Orange</aui:option>
            <aui:option value="red">Red</aui:option>
        </aui:select>

    </aui:fieldset>
    <aui:button-row>
        <aui:button type="submit"></aui:button>
    </aui:button-row>
</aui:form>
```

This JSP uses the `<liferay-portlet:actionURL />` and `<liferay-portlet:renderURL />` tags to construct two URLs in the variables `configurationActionURL` and `configurationRenderURL`. The JSP presents a simple form that allows the user to select a favorite color. When the user submits the form, the `configurationActionURL` is invoked and the application's `processAction` method is invoked with the `favoriteColor` included as a request parameter:

```
<aui:form action="<%= configurationActionURL %>" method="post" name="fm">
```

If the request fails, the user is redirected to the configuration page:

```
<aui:input name="redirect" type="hidden"
    value="<%= configurationRenderURL %>" />
```

It's a best practice to supply a URL parameter named `cmd` (`Constants.CMD` equals `cmd`) whose value indicates the purpose of the request. In this example, the value of the `cmd` parameter is `update` (`Constants.CMD` equals `update`):

```
<aui:input name="<%= Constants.CMD %>" type="hidden"
    value="<%= Constants.UPDATE %>" />
```

Many core applications read the value of the `cmd` parameter and perform some processing depending on its value.

If you're developing an application using the example code from this tutorial, deploy the application to Liferay DXP, add it to a page, and click on the *Options* button (⋮), then select *Configuration*. Select a favorite color and click *Save*. To confirm that your selection was saved as a portlet configuration setting, look for the application to display a message like this:

```
Favorite color: blue!
```

Excellent! Now you know how to create application configurations and how to create a mechanism to allow users to edit the configuration.

89.3 Transitioning from Portlet Preferences to the Configuration API

This tutorial describes how to take an existing portlet developed for Liferay Portal 6.2 or prior, which uses portlet preferences to allow administrators to configure the portlet, and convert it to use the new Configuration API.

For more information on the Configuration API and the recommended ways to develop configurable applications for 7.0, please see the Making your applications configurable tutorial.

Before you start, it's important to understand the benefits of making this change. That will allow you to decide whether to go ahead with the change or not, since previous configuration mechanisms of Liferay Portal 6.2 and prior still work in 7.0.

The main benefits are:

- Ability to modify the default portlet preferences through the new System Settings UI. Previously the default configuration of a portlet could only be set by modifying the portlet.xml file, which was not easy to extend.

- In the future Liferay will support modifying the default values for any portlet using this API for each company or site. If you use the new API you won't need to make any change to benefit from this.

- You can now use the PortletPreferences API to allow users to have their own personal preferences for the portlet, which was the original intention of this API.
- Have full programmatic control of the scope of the configuration (Portlet Instance or Group), instead of leaving it to the liferay-portlet.xml file.
- It becomes easier to read the configuration of the portlet from any class or JSP just by knowing its name. The configuration is also presented as a typed object which reduces the possibility of error reading a property name or type.

One final thing to note is that using the new Configuration API preserves all existing configurations of the portlet, since it uses the same persistence storage (the PortletPreferences table). That means that you can deploy a new version of the portlet to a running system and the existing portlets that may have been added to its pages and configured will not need to be reconfigured.

If your existing (Liferay Portal 6.2 or prior) application uses portlet preferences and you have decided to convert it to use the new Configuration API, follow these steps to update your application:

1. Identify the existing preferences of the portlet configuration.

   For example, you could check the `configuration.jsp` (where you will find DOM elements with the name `preferences--XXX--`, where `XXX` is the name of the preference).

2. Determine the scope of the portlet configuration. The traditional way of specifying it is through `liferay-portlet.xml`. Look at the elements `preferences-company-wide`, `preferences-unique-per-layout` and `preferences-owned-by-group` to determine the right scope. The following table maps out the scopes:

liferay-portlet.xml	Scope
preferences-company-wide=true	Company
preferences-owned-by-group=true AND preferences-unique-per-layout=false	Group
preferences-owned-by-group=true AND preferences-unique-per-layout=true	Portlet Instance

```
Related to this, we make the following recommendations for the scope of the
configuration of a portlet:

* For any portlet that can be added to a page it should be Portlet Instance.

* For any portlet that is accessible through the product menu and is used to
administer the site it should be Group.
```

3. Create a Java interface that will represent the configuration, with one method per existing preference.

   By Liferay's convention, the suggested names for these interfaces are `[Portlet Name]PortletInstanceConfiguration` for the portlet scoped ones or `[Portlet Name]GroupServiceConfiguration` for the group scoped ones. However you can choose different conventions.

   This interface will use two annotations (`@Meta.OCD` and `@ExtendedObjectClassDefinition`) to declare that it represents a configuration and to specify the desired scope. The `id` specified in the `@Meta.OCD` annotation must be the fully qualified class name of the interface.

   For example,

```
@ExtendedObjectClassDefinition(
    category = "[category]",
    scope = ExtendedObjectClassDefinition.Scope.GROUP
)
@Meta.OCD(
    id = "[package].[PortletName]GroupServiceConfiguration",
)
public interface [PortletName]GroupServiceConfiguration {
    @Meta.AD(deflt = "", required = false)
    public String displayStyle();

    @Meta.AD(deflt = "0", required = false)
    public long displayStyleGroupId(long defaultDisplayStyleGroupId);
}
```

4. In the code above replace [category] with a category name of your choice. Out of the box Liferay uses: wem, forms, collaboration and foundation. If your portlet fits in any of those go ahead and use them. If you pick any other name it will be added, but you will need to provide a translation as well.

5. If you want to specify the name of the configuration as it will appear in System Settings (and optionally translate it to several languages) add the attributes name and localization (to specify the location of a resource bundle file) to the @Meta.OCD annotation:

```
@Meta.OCD(
    localization = "content/Language",
    name = "[PortletName].group.service.configuration.name"
    id = "[package].[PortletName]GroupServiceConfiguration",
)
```

6. Create one class that implements the Configuration Bean Declaration interface to let the Configuration framework know about the Configuration class.

```
@Component
public class [PortletName]GroupServiceConfigurationBeanDeclaration
        implements ConfigurationBeanDeclaration {

    @Override
    public class getConfigurationBeanClass() {
        return [[PortletName]GroupServiceConfiguration.class;
    }

}
```

7. Change the configuration JSP to retrieve the configuration using the Configuration API. If the scope is "Portlet Instance" the configuration can be retrieved from `portletDisplay`:

```
[PortletName]PortletInstanceConfiguration portletInstanceConfiguration =
    portletDisplay.getPortletInstanceConfiguration(
        [PortletName]PortletInstanceConfiguration.class);
```

Once the configuration object is obtained, the individual preferences can now be changed from this:

```
String displayStyle = portletPreferences.getValue("displayStyle", defaultValue);
```

... to this ...

```
String displayStyle = v.displayStyle();
```

8. Finally it is usually necessary to read the configuration from Java classes or other JSPs. In cases where the `portletDisplay` is not available, or when the scope is "Group" or "Company", the `PortletProvider` class offers methods to obtain the configuration. The best way to access the `PortletProvider` depends on who is making the invocation:

   Within an OSGi Component a reference to the `ConfigurationProvider` can be obtained and used as follows:

```
@Component(service = Foo)
    public class Foo {

        protected void methodWhichNeedsConfiguration() {
            _configurationProvider.getGroupConfiguration(
                [PortletName]GroupServiceConfiguration.class, groupId);
        }

        @Reference
        private ConfigurationProvider _configurationProvider;
    }
```

   Within a service created with Service Builder the code is very similar:

```
    public class FooServiceImpl {

        protected void methodWhichNeedsConfiguration() {
            _configurationProvider.getGroupConfiguration(
                [PortletName]GroupServiceConfiguration.class, groupId);
        }

        @Reference
        private ConfigurationProvider _configurationProvider;
    }
```

   For all other cases it is preferred to get the configuration injected or passed as a parameter. As a last resort it is possible to use `ConfigurationProviderUtil` to obtain the configuration, although this method might have issues in highly dynamic environments:

```
[PortletName]GroupServiceConfiguration groupConfiguration
        ConfigurationProviderUtil.getGroupConfiguration(
            [PortletName]GroupServiceConfiguration.class, groupId);
```

Chapter 90

Social

Liferay DXP's social framework lets users provide feedback on content, share that content with others across social networks, and subscribe to notifications on content, so they can stay up to date on the latest and greatest that you have to share.

The tutorials that follow show you how to take advantage of Liferay DXP's social framework to enable these features in your own app.

90.1 Applying Social Bookmarks

When you enable social bookmarks, icons for sharing on Twitter, Facebook, and Google Plus appear below your app's content. Liferay DXP's taglibs provide the markup you need to add this feature to your app.

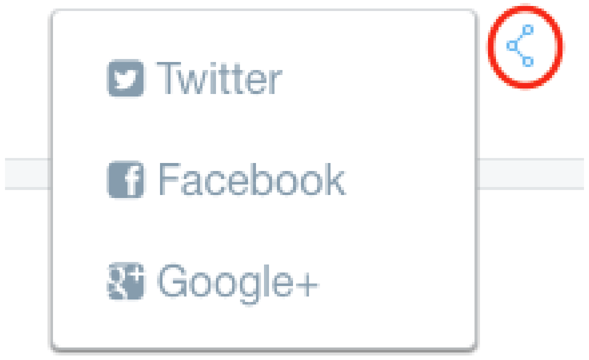


Figure 90.1: Social bookmarks are enabled in the built-in Blogs portlet

Follow these steps to add social bookmarks to your app:

1. Make sure your entity is asset enabled.

2. Choose a view in which to show the social bookmarks. For example, you can display them in one of your portlet's views, or if you've implemented asset rendering you can display them in the full content view in the Asset Publisher portlet.

3. In your JSP, include the `liferay-ui` taglib declaration:

```
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>
```

4. Use `ParamUtil` to get the entity's ID from the render request. Then use your `-LocalServiceUtil` class to create an entity object:

```
<%
long entryId = ParamUtil.getLong(renderRequest, "entryId");
entry = EntryLocalServiceUtil.getEntry(entryId);
%>
```

5. Use the `liferay-ui:social-bookmarks` tag to add the social bookmarks component. The `target` attribute refers to the HTML target. Pass the content's URL in the `url` attribute, using `PortalUtil` to retrieve the URL. The URL and `title` you provide is sent to the social network and becomes part of the link you create there:

```
<liferay-ui:social-bookmarks
    contentId="<%= String.valueOf(assetEntry.getEntryId()) %>"
    displayStyle="menu"
    target="_blank"
    title="<%= String.valueOf(entry.getName()) %>"
    url="<%= PortalUtil.getCanonicalURL((PortalUtil.getCurrentURL(request)),
        themeDisplay, layout) %>"
/>
```

   Pay special attention to the `displayStyle` attribute. This attribute sets the social bookmarks' appearance. Setting `displayStyle` to `menu`, as in this example, hides the share buttons in a clutter-free menu (see above screenshot).

   Setting `displayStyle` to `simple` displays the share buttons in a simple row without share stats.

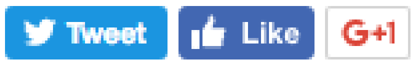


Figure 90.2: Here are the share buttons with `displayStyle` set to `simple`.

   Setting `displayStyle` to `vertical` displays the share buttons in a column, with share stats above each (share stats show the number of times the asset has been shared on the corresponding social network).

   Setting `displayStyle` to `horizontal` displays the share buttons in a row with share stats to the right of each.

The social bookmarks UI component now shows in your entity's view.

Note: You can install the Social Bookmarks app from the Marketplace (available for CE and DXP) to let your users share your app's content across more social networks. For more information, see the article Integrating with Facebook, Twitter, and More.

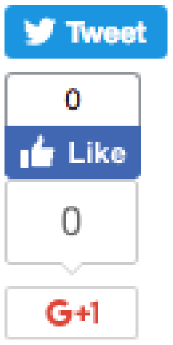


Figure 90.3: Here are the share buttons with displayStyle set to vertical.

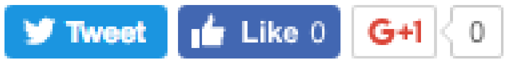


Figure 90.4: Here are the share buttons with displayStyle set to horizontal.

Great! Now you know how to let users share content in your asset enabled apps.

Related Topics

Adding, Updating, and Deleting Assets for Custom Entities
Adding Permissions to Resources
Rendering an Asset
Using the Liferay UI Taglib

90.2 Adding Comments to your App

Users adding comments to your content makes your site come alive. Instead of you statically giving users information, now the flow goes both ways. Liferay DXP's asset framework makes it easy to add a comment system to your app.

This tutorial shows you how to add the comment feature for your application's content.

Follow these steps:

1. Make sure your entity is asset enabled.

2. Choose a read-only view of the entity for the comments. You can display the comments component in your portlet's view, or if you've implemented asset rendering, you can display it in the full content view in the Asset Publisher portlet.

3. Include the Liferay-UI taglib and Portlet taglib declarations in your JSP:

```
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>
<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet_2_0" %>
```

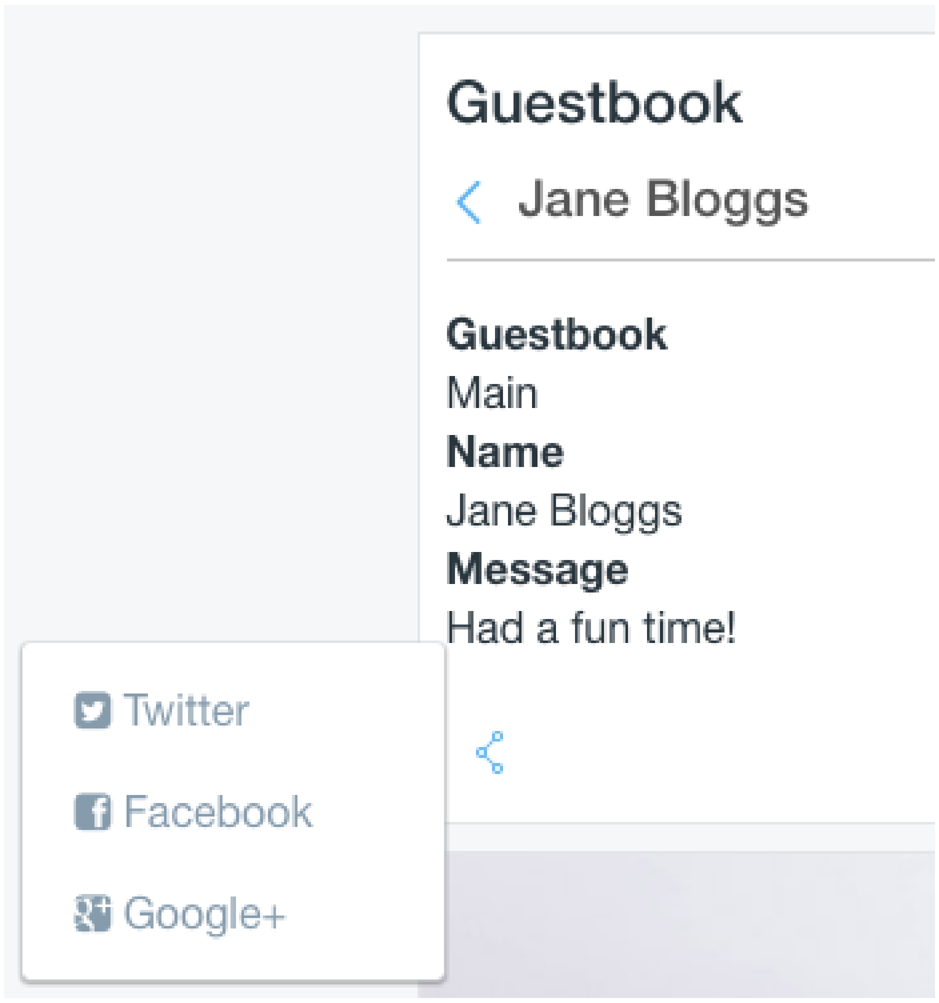


Figure 90.5: The new JSP lets users share app content to social networks.

4. Use `ParamUtil` to get the entity's ID from the render request. Then create an entity object using the `-LocalServiceUtil` class. Below is an example configuration:

```
<%
long entryId = ParamUtil.getLong(renderRequest, "entryId");
entry = EntryLocalServiceUtil.getEntry(entryId);
%>
```

5. Create a collapsible panel for the comments using the `liferay-ui:panel-container` and `liferay-ui:panel` tags. This makes the discussion area hide-able. Below is an example configuration:

```
<liferay-ui:panel-container extended="<%=false%>"
  id="guestbookCollaborationPanelContainer" persistState="<%=true%>">
  <liferay-ui:panel collapsible="<%=true%>" extended="<%=true%>"
    id="guestbookCollaborationPanel" persistState="<%=true%>"
    title="Collaboration">
```

6. Next, create a URL for the discussion using the `portlet:actionURL` tag:

```
<portlet:actionURL name="invokeTaglibDiscussion" var="discussionURL" />
```

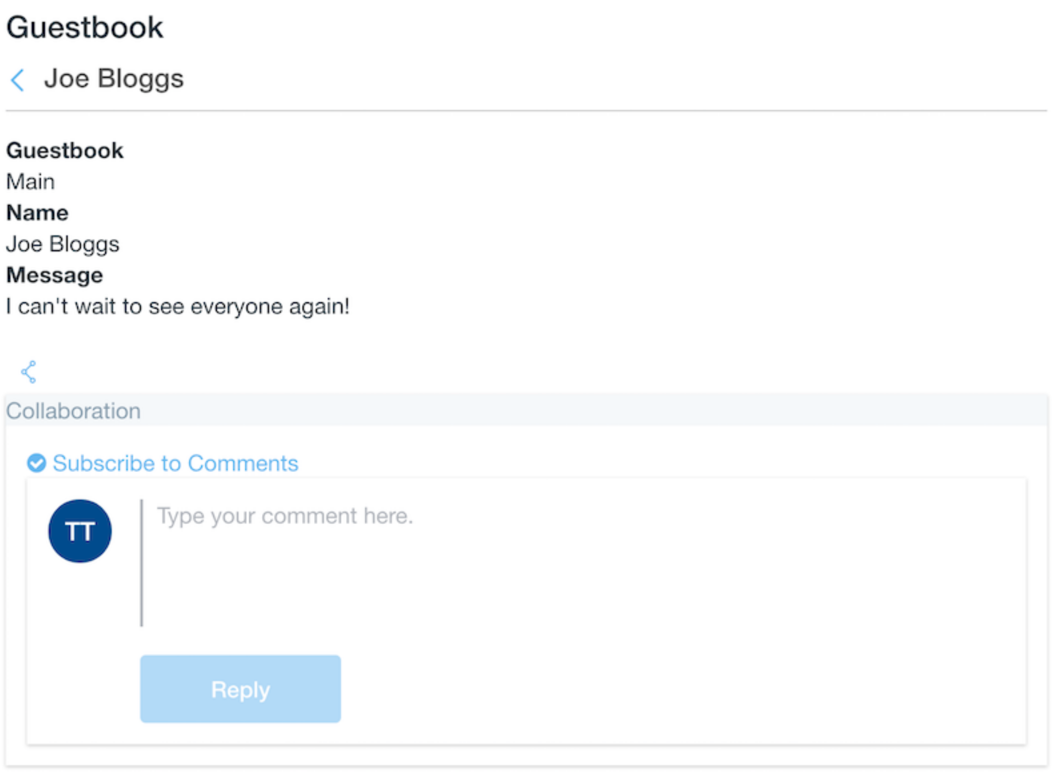


Figure 90.6: Your JSP lets users comment on content in your portlet.

7. Finally, add the discussion with the `liferay-ui:discussion` tag. Pass the current URL using the `redirect` attribute, so the user can return to the JSP after making a comment. You can use `PortalUtil.getCurrentURL((renderRequest))` to get the current URL from the request object. Below is an example configuration:

```
    <liferay-ui:discussion className="<%=Entry.class.getName()%>"
      classPK="<%=entry.getEntryId()%>"
      formAction="<%=discussionURL%>" formName="fm2"
      ratingsEnabled="<%=true%>" redirect="<%=currentURL%>"
      userId="<%=entry.getUserId()%>" />

  </liferay-ui:panel>
</liferay-ui:panel-container>
```

If you haven't already connected your portlet's view to the JSP for your entity, you can refer here to see how to connect a portlet's main view JSP to an entity's view JSP.

Great! Now you know how to let users comment on content in your asset enabled portlets.

Related Topics

Adding, Updating, and Deleting Assets for Custom Entities

Adding Permissions to Resources

Rendering an Asset

Applying Social Bookmarks

90.3 Rating Assets

Liferay DXP's asset framework supports a system for rating content in apps. This feature appears in many core apps, such as the Blogs portlet. Ratings give your users a voice, and you can benefit from their feedback. Thanks to Liferay DXP's taglibs, you can enable ratings for your app in only a few lines of code.

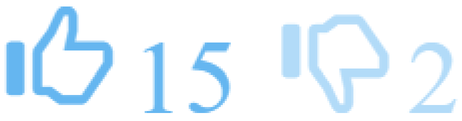


Figure 90.7: Ratings let users quickly provide feedback on content.

Follow these steps:

1. Make sure your entity is asset enabled.
2. Choose a read-only view of the entity for the ratings. You can display them in one of your portlet's views, or if you've implemented asset rendering you can display them in the full content view in the Asset Publisher portlet.
3. In the JSP, include the `liferay-ui`taglib declaration:

```
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>
```

4. Use `ParamUtil` to get the entity's ID from the render request. Then use the `-LocalServiceUtil` class to create an entity object:

```
<%
long entryId = ParamUtil.getLong(renderRequest, "entryId");
entry = EntryLocalServiceUtil.getEntry(entryId);
%>
```

5. Use the `liferay-ui:ratings` tag to add the ratings component:

```
<liferay-ui:ratings className="<%=Entry.class.getName()%>"
    classPK="<%=entry.getEntryId()%>" type="stars" />
```

The type attribute specifies the type of rating system to display:

- *like*: a likes rating system
- *stars*: a five star rating system
- *thumbs*: a thumbs-up or thumbs-down rating system, as shown in the figure above

Although the ratings type is specified here, you can make the type configurable for administrators by Implementing Ratings Type Selection and Value Type Transformation in your app.

Great! Now you know how to let users rate content in your asset-enabled portlets.

Related Topics

Adding Comments to Your App

Applying Social Bookmarks

90.4 Implementing Ratings Type Selection and Value Transformation

Liferay DXP has three different mechanisms for rating content: Stars, Thumbs Up/Down, and Likes. Prior to 7.0, there was no way for administrators to select a ratings type as it was hard-coded. Now, portal and site admins can select the ratings type for portlet entities through the Control Panel and Site Administration. Portal admins can select default values for the ratings type while site admins can override values for their sites. All Liferay portlets take advantage of this feature, and so can custom portlets.

A custom portlet that uses ratings must define its ratings type through an OSGi component that implements the `PortletRatingsDefinition` interface. This class declares the usage of ratings (specifying the portlet and the entity) and the default ratings type (that can be overridden by portal and site admins). Developers can include their portlet entities in this section by defining the ratings type for the entities using OSGi modules.

The ratings values are stored in the database as normalized values to be able to easily switch among different ratings types. When the administrators change the ratings type for an entity, Liferay does a best match of the previous ratings values to values for the new ratings type. For example, when changing from Thumbs Up/Down to Likes, Portal considers all the Thumbs Up values as if they are Likes and omits all Thumbs Down values. Also, when changing from Stars to Thumbs Up/Down Portal considers ratings with 3, 4, or 5 Stars as Thumbs Up and ratings with 1 or 2 Stars as Thumbs Down.

However, there are some cases where this may not be enough or where you want to apply different or complex criteria to determine the new ratings values. Liferay provides a mechanism to allow developers to transform the ratings values as needed when admins change the ratings type. Keep in mind, this is not an interpretation of the data, this is modification of the stored ratings values. Third party developers can define their own transformations by creating an OSGi component that implements the `RatingsDataTransformer` interface. The highest OSGi ranking implementation is used.

Liferay by default doesn't provide any `RatingsDataTransformer` implementation; the ratings values always remain the same while Liferay interprets existing values for the selected ratings type.

Specifying an Entity's Ratings Type

Ratings type definitions needs to implement the `PortletRatingsDefinition` interface. The implementation of method `getDefaultRatingsType` returns the entity's default ratings type. Note, it can be overridden by portal and site admins. The implementation of method `getPortletId` returns the portlet ID of the main portlet that uses the entity.

Implementations of `PortletRatingsDefinition` need to be registered in OSGi to be used by the portal. You do this in the implementation by specifying the OSGi annotation `@Component` with an attribute `model.class.name` set to the name or names of the classes to use the ratings definition.

Here is an example of a `PortletRatingsDefition` implementation for the Blogs portlet that defines the `blogsEntry` ratings data:

```
@Component(
    property = {
        "model.class.name=com.liferay.portlet.blogs.model.BlogsEntry"
    }
)
public class BlogsPortletRatingsDefinition implements PortletRatingsDefinition {
    @Override
    public RatingsType getDefaultRatingsType() {
        return RatingsType.THUMBS;
    }

    @Override
    public String getPortletId() {
        return PortletKeys.BLOGS;
    }

}
```

Next, let's examine how to transform values between ratings types.

Transforming Ratings Values Between Ratings Types

If the site admin or portal admin changes the ratings type and there are existing ratings values in the database, Liferay doesn't modify any of them. It interprets the values using the selected ratings type according to the following mechanism:

1. When changing from Stars to …

    - **Like:** 1 or 2 Stars aren't considered. 3, 4, or 5 Stars is considered a Like.
    - **Thumbs Up/Down:** 1 or 2 Stars are considered a Thumbs Down. 3, 4, or 5 Stars is considered a Thumbs Up.

2. When changing from Thumbs Up/Down to …

    - **Like:** A Like is considered a Thumbs Up.
    - **Stars:** A Thumbs Down is considered 1 Star. A Thumbs Up is considered 5 Stars.

3. When changing from Like to …

    - **Stars:** A Like is considered 5 Stars.
    - **Thumbs Up/Down:** A Like is considered a Thumbs Up.

There may be cases where this interpretation is insufficient for you or where a different criteria or algorithm needs to be applied. For these cases, Liferay provides a mechanism to allow transforming the existing ratings values. Keep in mind that this modifies the values stored in the database. Since these changes may not be reversible, you must be careful. Developers will need to create an OSGi component that implements the `RatingsDataTransformer` interface.

The implementation of method `transformRatingsData` performs the data transformation. In order to obtain a fine-grained behaviour, the framework provides the methods `fromRatingsType` and `toRatingsType`. The developer can implement them to transform data as needed, and perform different transformations for different circumstances.

This is an example of a `RatingsDataTransformer` that resets the score from the ratings type when passing from `LIKE` to `STARS`. In this particular case, changes are not reversible.

```
@Component
public class DummieRatingsDataTransformer implements RatingsDataTransformer {
    @Override
    public ActionableDynamicQuery.PerformActionMethod transformRatingsData(
            final RatingsType fromRatingsType, final RatingsType toRatingsType)
        throws PortalException {
```

```
        return new ActionableDynamicQuery.PerformActionMethod() {

            @Override
            public void performAction(Object object)
                throws PortalException {

                if (fromRatingsType.getValue().equals(RatingsType.LIKE) &&
                    toRatingsType.getValue().equals(RatingsType.STARS)) {

                    RatingsEntry ratingsEntry = (RatingsEntry) object;

                    ratingsEntry.setScore(0);

                    RatingsEntryLocalServiceUtil.updateRatingsEntry(
                        ratingsEntry);
                }
            }
        };
    }

}
```

Once you've implemented ratings type selection and value type transformation for your app's entities, you can configure the default ratings type values through the Control Panel by going to *Configuration* → *Instance Settings* and selecting the *Social* tab. To override the default values for a site, go to Site Administration → *Configuration* → *Site Settings* and select the *Social* tab.

In this tutorial, you have learned how to set a ratings type for an entity and how to implement a ratings data transformer. Liferay salutes you with a Thumbs Up!

Related Topics

Rating Assets

Adding Comments to Your App

90.5 Flagging Inappropriate Asset Content

In a perfect world, people would post nice, kind, and decent content. They would reply to comments with constructive feedback and never lash out at one another. Unfortunately, sometimes people have a bad day and decide to take their frustration out on Joe Bloggs in the form of an inappropriate post. No worries though, Liferay DXP's asset framework supports a system for flagging content in apps. Letting users flag inappropriate content takes much of the work off site administrators.

This tutorial shows you how to enable flagging of content in a portlet.

Follow these steps:

1. Make sure your entity is asset enabled.

2. Choose a read-only view of the entity for the flags. You can display them in one of your portlet's views, or if you've implemented asset rendering, you can display them in the full content view in the Asset Publisher portlet.

3. In the JSP, include the `liferay-flags` taglib declaration:

   ```
   <%@ taglib prefix="liferay-flags" uri="http://liferay.com/tld/flags" %>
   ```

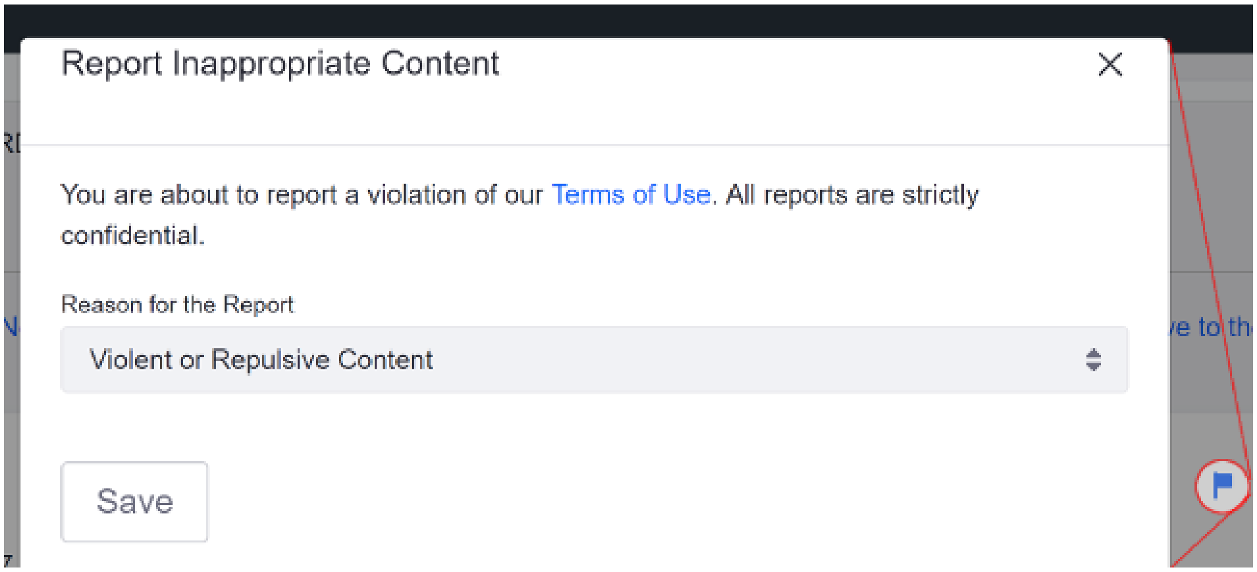

Report Inappropriate Content
You are about to report a violation of our Terms of Use. All reports are strictly confidential.
Reason for the Report
Violent or Repulsive Content
Save

Figure 90.8: Flags for letting users mark objectionable content are enabled in the Message Boards portlet.

4. Use the -LocalServiceUtil class to get the entity:

```
<%
long entryId = ParamUtil.getLong(renderRequest, "entryId");
entry = EntryLocalServiceUtil.getEntry(entryId);
%>
```

5. Use the liferay-flags:flags tag to add the flags component:

```
<liferay-flags:flags
    className="<%= Entry.class.getName() %>"
    classPK="<%= entry.getEntryId() %>"
    contentTitle="<%= title %>"
    message="flag-this-content"
    reportedUserId="<%= reportedUserId %>"
/>
```

   The reportedUserId attribute specifies the user who flagged the asset.

Great! Now you know how to let users flag content in your asset-enabled portlets.

Related Topics

Adding Comments to Your App
Applying Social Bookmarks

Chapter 91

Item Selector

People's tasks often involve making choices: which shirt to wear, what to eat for breakfast, or what profile picture to use. The power of choice is a terrific privilege, but the means by which a person makes his/her selection can make an experience good or bad. Here are some factors that influence the experience:

- Item organization
- Consistency
- UI intuitiveness

Liferay DXP's Item Selector provides all these things. It is a UI component that enables users to select entities in a consistent easy-to-use manner. Many Liferay portlets, such as Documents and Media, Web Content, Blogs and more, use Item Selectors for selecting things such as images, videos, audio files, documents, and pages.

The Item Selector API provides a framework for developers to use and extend Item Selectors. They can add Item Selectors to their apps, customize Liferay DXP's Item Selectors, and create Item Selectors to select any kind of entity.

Selection views are the framework's key components. In an Item Selector, selection views show entities of particular types from different sources.

For example, an Item Selector configured to show images might show selection views from the following sources:

- Documents and Media
- Third-party image provider
- Drag-and-drop interface

Here are some different Item Selector use cases:

1. Enabling your application to select Liferay DXP entities such as sites, pages, or documents from Documents and Media.
2. Customizing a selection experience by adding a new selection view for an entity (e.g., a view of images from an external image repository).
3. Creating new selectable entities for other applications to use in their Item Selectors.

This group of tutorials demonstrates how to satisfy these use cases.

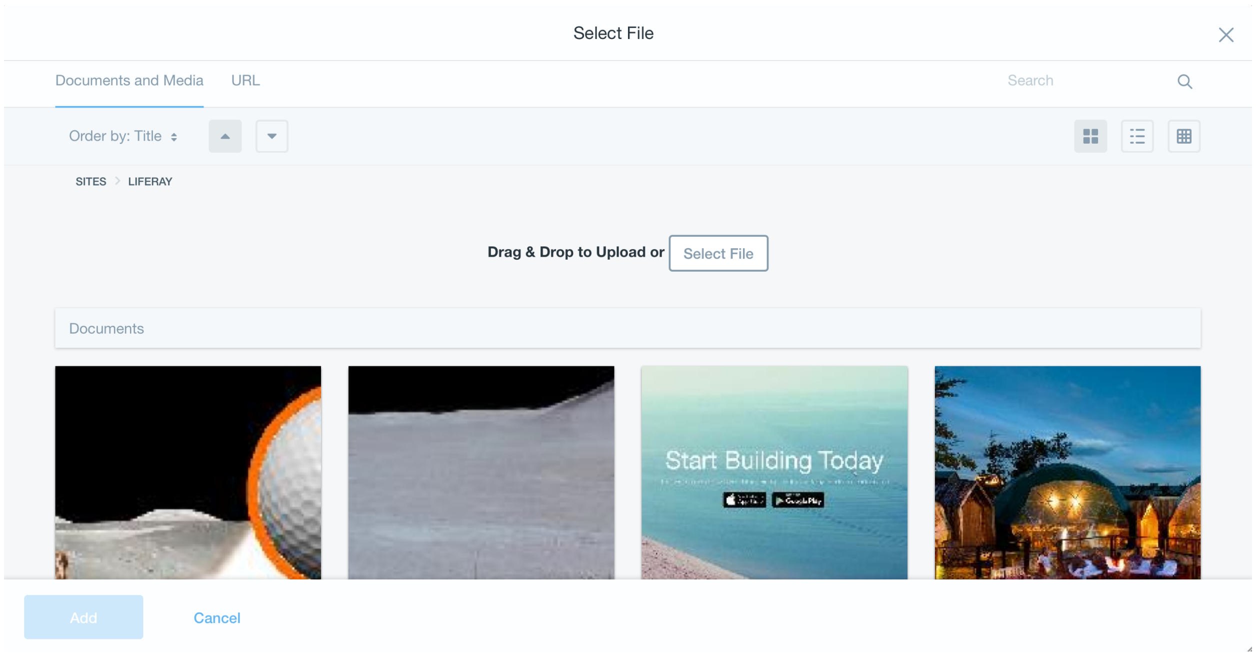
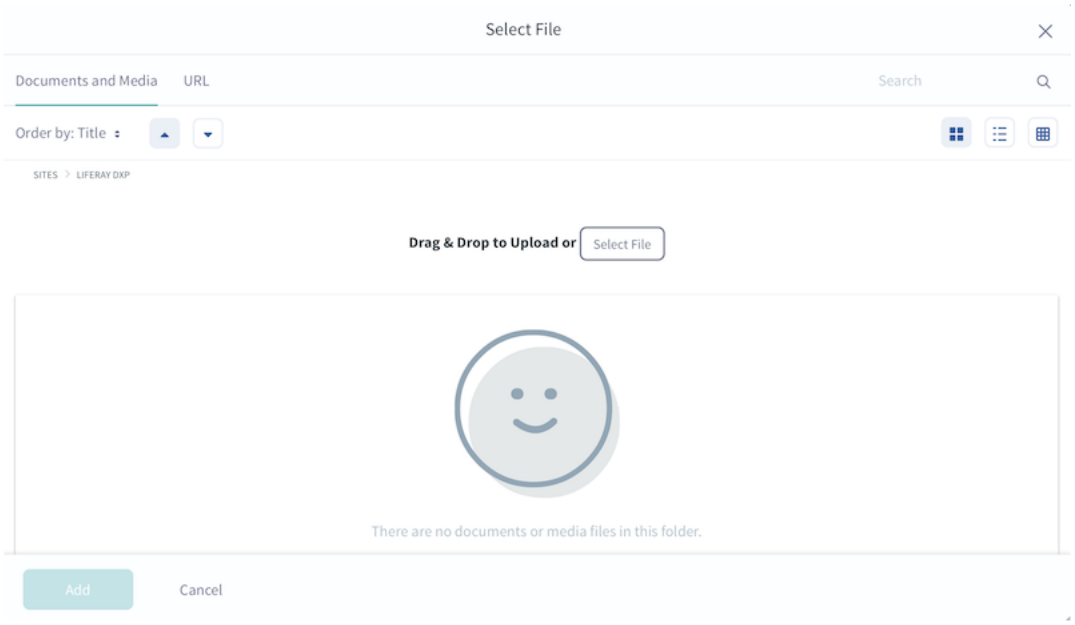


Figure 91.1: Item Selectors let users browse and select different kinds of entities.

91.1 Selecting Entities Using the Item Selector

The Item Selector allows users to select entities, such as images, videos, documents, and sites.



There are no documents or media files in this folder.

Figure 91.2: The Item Selector makes selecting entities a breeze.

Here's what's required to use an Item Selector:

1. **Determine Item Criteria**

2. **Get an Item Selector for the Criteria**

3. **Use an Item Selector Dialog**

Determining Item Selector Criteria

The first step is determining entity types to select from the Item Selector and the data you expect from them. What kind of entity do you want to select? Do you want to select a user, an image, a video, or something else?

Once you know the entities you want, you need *criterion* classes to represent them in the Item Selector. Criterion classes must implement the `ItemSelectorCriterion` interface. The Item Selector Criterion and Return Types reference lists criterion classes Liferay's apps and app suites provide.

If there's no criterion class for your entity, you can create your own `ItemSelectorCriterion` class (tutorial coming soon).

Then determine the type of information (return type) you expect from the entities when users select them. Do you expect a URL? A Universally Unique Identifier (UUID)? A primary key? Each return type must be represented by an implementation of the `ItemSelectorReturnType` class. The Item Selector Criterion and Return Types reference also lists return type classes Liferay's apps and app suites provide.

If there's no return type class that meets your needs, you can implement your own `ItemSelectorReturnType` class (tutorial coming soon).

Note: Each criterion must have at least one `ItemSelectorReturnType` (return type) associated with it.

For example, if you want to allow users to select an image and want the image's URL returned, you could use the `ImageItemSelectorCriterion` criterion class and the `URLItemSelectorReturnType` return type.

The criterion and return types are collectively referred to as the Item Selector's *criteria*. The Item Selector uses it to decide which selection views (tabs of items) to show.

Once you've defined your criteria, you can get an Item Selector to use with it.

Getting an Item Selector for the Criteria

In order to use an Item Selector with your criteria, you must get an Item Selector URL based on the criteria. The URL is needed to open the Item Selector dialog in your UI. In Java, you build the criteria and pass it in a call to get the Item Selector's URL.

First, get an Item Selector OSGi Service Component using Declarative Services.

```
import com.liferay.item.selector.ItemSelector;
import org.osgi.service.component.annotations.Reference;

@Reference
private ItemSelector _itemSelector
```

The component annotations are available in the `org.osgi.service.component.annotations` module.

To get a URL to the Item Selector, you must call its `getItemSelectorURL` method using the following parameters:

- `RequestBackedPortletURLFactory`: Factory that creates portlet URLs.
- `ItemSelectedEventName`: Unique arbitrary JavaScript event name that is triggered by the Item Selector when the element is selected.
- `ItemSelectorCriterion`: Criterion (or an array of criterion objects) that specifies the type of elements to make available in the Item Selector.

The following code demonstrates getting a URL to an Item Selector configured with criteria for images:

```
RequestBackedPortletURLFactory requestBackedPortletURLFactory =
    RequestBackedPortletURLFactoryUtil.create(request);

List<ItemSelectorReturnType> desiredItemSelectorReturnTypes =
    new ArrayList<>();
desiredItemSelectorReturnTypes.add(new URLItemSelectorReturnType());

ImageItemSelectorCriterion imageItemSelectorCriterion =
    new ImageItemSelectorCriterion();

imageItemSelectorCriterion.setDesiredItemSelectorReturnTypes(
    desiredItemSelectorReturnTypes);

PortletURL itemSelectorURL = _itemSelector.getItemSelectorURL(
    requestBackedPortletURLFactory, "sampleTestSelectItem",
    imageItemSelectorCriterion);
```

First it gets a factory to create the URL by invoking the `RequestBackedPortletURLFactoryUtil.create` method, passing it the current request object. The request can be an `HttpServletRequest` or `PortletRequest`.

Then it creates a list of return types expected for the image entity and a criterion for images. The return types list consists of a URL return type `URLItemSelectorReturnType`. The list is passed to the criterion's `setDesiredItemSelectorReturnTypes` method.

Lastly the method `getItemSelectorURL` is called to return a URL based on the criteria. The method requires a URL factory, an arbitrary event name, and a series of criterion (one, in this case).

Note: You can invoke the URL object's `toString` method to get its value.

An Item Selector can be configured to use any number of criterion. The criterion can use any number of return types.

The Item Selector's criterion order determines the selection view order. For example, if you pass the item selector criteria in this order: `ImageItemSelectorCriterion`, `VideoItemSelectorCriterion`, the Item Selector displays the image selection views first and then the ones for videos. If you reverse the order, it shows the video selection views first and the image selection views second.

Return type order is also significant. A view uses the first return type it supports from each criterion's return type list.

Now that you've got a URL to the Item Selector you've configured, you can start using the Item Selector in your UI.

Using the Item Selector Dialog

To open the Item Selector in your UI, you must use the `LiferayItemSelectorDialog` JavaScript component from AlloyUI's `liferay-item-selector-dialog` module. The component listens for the item selected event that you specified for the Item Selector URL. The event returns the selected element's information according to its return type.

Here are the steps for using the Item Selector dialog in a JSP:

1. Declare the AUI tag library.

   ```
   <%@ taglib prefix="aui" uri="http://liferay.com/tld/aui" %>
   ```

2. Add the `<aui:script>` tag and configure it to use the `liferay-item-selector-dialog` module:

```
<aui:script use="liferay-item-selector-dialog">

</aui:script>
```

3. Attach an event handler to the UI element you want to use to open the Item Selector dialog.

   For example, the configuration below creates a *Choose* button with the ID `chooseImage` with a click event and function attached.

```
<aui:button name="chooseImage" value="Choose" />

<aui:script use="liferay-item-selector-dialog">

    $('#<portlet:namespace />chooseImage').on(
    'click',
      function(event) {
        <!-- function logic goes here -->
      }
    );

</aui:script>
```

4. Inside the function, you must create a new instance of the `LiferayItemSelectorDialog` AlloyUI Component and configure it to use the Item Selector. Here's what you need to do in the function:

   - Set the function's `eventName` attribute to the item selected event name you specified in your Java code (the code that gets the Item Selector URL).
   - For the `on` attribute, implement a function that operates on the selected item change. The example code below demonstrates this.
   - Set the function's `title` attribute. This becomes the dialog's title.
   - Set the function's `url` attribute to the Item Selector URL you obtained earlier.
   - Call the dialog's `open` method.

The example code below and steps that follow show how to configure an Item Selector dialog to work with selected items.

```
<aui:button name="chooseImage" value="Choose" />

<%
String itemSelectorURL = GetterUtil.getString(request.getAttribute("itemSelectorURL"));
%>

<aui:script use="liferay-item-selector-dialog">

    $('#<portlet:namespace />chooseImage').on(
        'click',
        function(event) {
            var itemSelectorDialog = new A.LiferayItemSelectorDialog(
                {
                    eventName: 'ItemSelectedEventName',
                    on: {
                            selectedItemChange: function(event) {
                                var selectedItem = event.newVal;

                                if (selectedItem) {
                                    var itemValue = JSON.parse(
                                    selectedItem.value
                                    );
                                    itemSrc = itemValue.url;
                                    <!-- use item as needed -->
                                }
                            }
                    },
                    title: '<liferay-ui:message key="select-image" />',
                    url: '<%= itemSelectorURL.toString() %>'
                }
            );
            itemSelectorDialog.open();
        }
    );
</aui:script>
```

Here's what the code does:

1. This line creates a new instance of the Liferay Item Selector dialog:

   ```
   var itemSelectorDialog = new A.LiferayItemSelectorDialog(...)
   ```

2. The constructor sets the `eventName` attribute. This makes the dialog listen for the item selected event. The event name is the same as the one specified for the Item Selector URL generated earlier in the Java code.
3. When the user selects a new item, the `selectedItemChange` function fires, setting its variables for the newly selected item. The information available to parse depends on the return type(s) that were set.
4. As the comment indicates, the developer adds logic for using the selected element here.
5. The `LiferayItemSelectorDialog`'s `title` attribute sets the dialog's title.
6. The Item Selector URL retrieved previously in the Java code becomes the `url` attribute's value.
7. The `open` method opens the Item Selector dialog.

When the user clicks the *Choose* button, a new dialog opens, rendering the Item Selector with the views that support the criterion and return type(s) that were set.

Adding an Item Selector to your app is as easy as what's been demonstrated! It involves specifying criteria for the selectable items, applying them to an Item Selector, and configuring an Item Selector dialog to operate on the selected item. Using Item Selector API, you can give app users the power of choice!

Related Articles

Creating Custom Item Selector Views

Creating Custom Item Selector Entities

Front-End Taglibs

91.2 Creating Custom Item Selector Entities

Does your app require users to select an item that the Item Selector isn't configured for? No problem. You can create a new entity.

This tutorial explains how to create a new entity for the Item Selector.

Creating Item Selector Criterion

First, you must create a new criterion for your entity. Follow these steps to create an Item Selector criterion:

1. Create a class that extends the `BaseItemSelectorCriterion` class.

   The example below creates a class called `TaskItemSelectorCriterion`:

   ```
   public class TasksItemSelectorCriterion extends
   BaseItemSelectorCriterion {

   }
   ```

   This class specifies what kind of entity the user is selecting and what information the Item Selector should return. The methods inherited from the `BaseItemSelectorCriterion` class provide the logic for obtaining this information:

   ```
   public abstract class BaseItemSelectorCriterion
           implements ItemSelectorCriterion {

           @Override
           public List<ItemSelectorReturnType> getDesiredItemSelectorReturnTypes() {
                   return _desiredItemSelectorReturnTypes;
           }

           @Override
           public void setDesiredItemSelectorReturnTypes(
                   List<ItemSelectorReturnType> desiredItemSelectorReturnTypes) {

                   _desiredItemSelectorReturnTypes = desiredItemSelectorReturnTypes;
           }

           private List<ItemSelectorReturnType> _desiredItemSelectorReturnTypes;

   }
   ```

   Note that you can use this class to pass information to the view if needed. For example, the `JournalItemSelectorCriterion` class passes information about the primary key so the view can use it:

   ```
   public class JournalItemSelectorCriterion extends
   BaseItemSelectorCriterion {

           public JournalItemSelectorCriterion() {
           }

           public JournalItemSelectorCriterion(long resourcePrimKey) {
                   _resourcePrimKey = resourcePrimKey;
           }

           public long getResourcePrimKey() {
                   return _resourcePrimKey;
           }

           public void setResourcePrimKey(long resourcePrimKey) {
                   _resourcePrimKey = resourcePrimKey;
           }

           private long _resourcePrimKey;

   }
   ```

```
**Note:** Criterion fields should be serializable and should expose a
public empty constructor (as shown above).
```

---

2. Create an OSGi component class that implements the `BaseItemSelectorCriterionHandler` class. Each criterion requires a criterion handler, which is responsible for obtaining the proper selection view.

   The example below creates a criterion handler for the `TaskItemSelectorCriterion` class:

   ```
   @Component(service = ItemSelectorCriterionHandler.class)
   public class TaskItemSelectorCriterionHandler extends
   BaseItemSelectorCriterionHandler<TaskItemSelectorCriterion> {

       public Class <TaskItemSelectorCriterion>
       getItemSelectorCriterionClass() {
           return TasksItemSelectorCriterionHandler.class;
       }

       @Activate
       @Override
       protected void activate(BundleContext bundleContext) {
               super.activate(bundleContext);
       }

   }
   ```

   The `@Activate` and `@Override` tokens are required to activate this OSGi component.

Depending on the needs of your app, you may not need to create a return type. If your entity returns information that is already defined by an existing return type, you can use that return type instead.

You can view the default available criteria in the Item Selector Criterion and Return Types reference.

If, however, your entity returns information that is not covered by an existing return type, you'll need to create a new return type next.

Creating Item Selector Return Types

To create a return type, you must create a class that implements the `ItemSelectorReturnType` class.

The example below creates a new return type called `TaskItemSelectorReturnType`:

```
/**
* This return type should return the task ID and the user who
* created the task as a string.
*
* @author Joe Bloggs
*/
public class TaskItemSelectorReturnType implements ItemSelectorReturnType{

}
```

The `*ItemSelectorReturnType` class is used as an identifier by the Item Selector and does not return any information itself. You should determine the information you expect returned (an ID, a URL, a more complex object, etc.), and create a return type to handle that information. The return type class is an API that connects the return type to the Item Selector views. **Whenever the return type is used, the view must ensure that the proper information is returned.** It's recommended that you specify the information that the return type returns, as well as the format, as Javadoc (as shown in the `TaskItemSelectorReturnType` example above).

So far, you've created an API that you can use to create a selection view for your new entity. The entity's criterion and return type classes are used by your application to create the Item Selector URL. You can follow the Selecting Entities using the Item Selector tutorial to learn how to obtain the Item Selector URL.

The selection view is responsible for returning the proper entity information specified by the return type. Currently there isn't a selection view to select your entity. Follow the Creating Custom Item Selector Views tutorial to learn how to create your new view.

Now you know how to create an entity for the Item Selector!

Related Topics

Selecting Entities using the Item Selector

Creating Custom Item Selector Views

91.3 Creating Custom Item Selector Views

Have you found you need to create a new selection view for your app? No problem. Item Selector views are determined by the type of entity the user is selecting. The Item Selector can render multiple views for the same entity type. For example, when a user requires an image from the Item Selector, the selection views shown below are rendered:

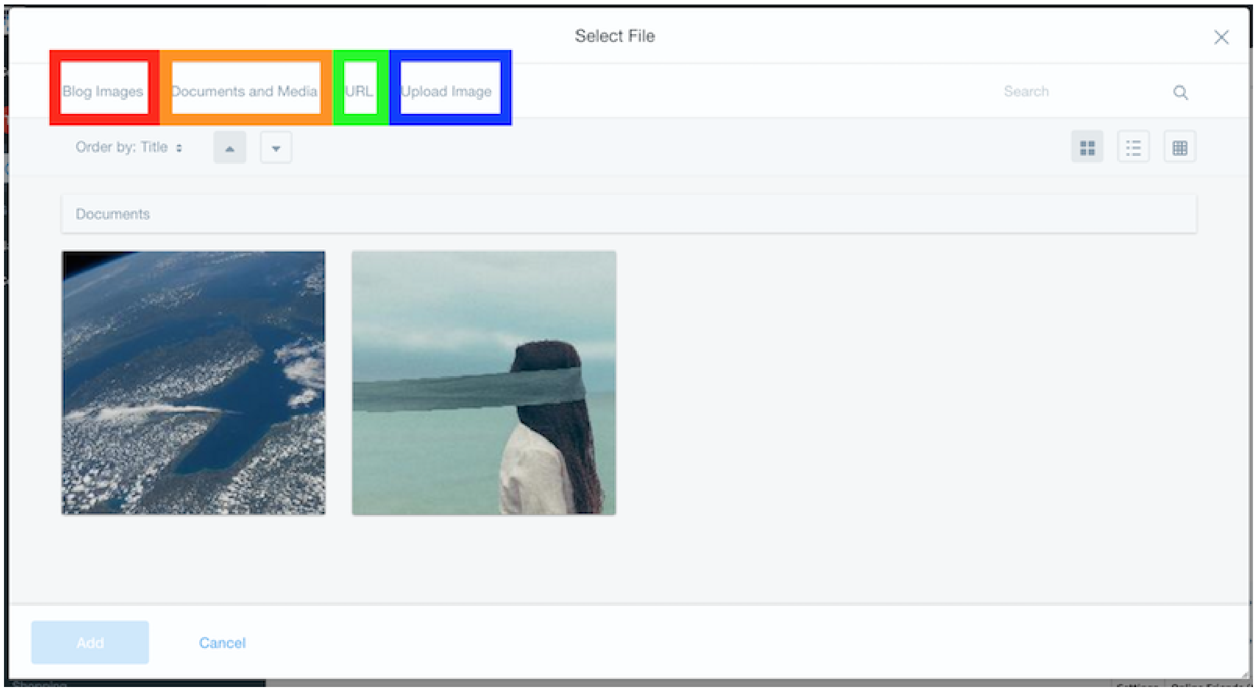


Figure 91.3: An entity type can have multiple selection views.

Each tab: *Blog Images*, *Documents and Media*, *URL*, and *Upload Image*, is a selection view for the Item Selector, each one represented by an `*ItemSelectorCriterion` class. The tabs in figure 1 are represented by the following `*ItemSelectorCriterion`:

- `BlogsItemSelectorCriterion` class: Blog Images View
- `ImageItemSelectorCriterion` class: Documents and Media View

- URLItemSelectorCriterion class: URL View
- UploadItemSelectorCriterion class: Upload Image View

The default selection views may provide everything you need for your application. If, however, your application requires a custom selection view, for instance to link to an external image provider, you can follow the steps outlined in this tutorial.

This tutorial covers how to create new selection views for the Item Selector.

Get started by configuring the module for your view next.

Configuring the Module

Follow these steps to prepare your module:

1. Add these dependencies to your module's build.gradle:

```
dependencies {
        compileOnly group: "com.liferay", name: "com.liferay.item.selector.api", version: "2.0.0"
        compileOnly group: "com.liferay", name: "com.liferay.item.selector.criteria.api", version: "2.0.0"
        compileOnly group: "com.liferay.portal", name: "com.liferay.portal.impl", version: "2.0.0"
        compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
        compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", version: "2.0.0"
        compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
        compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
        compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

2. Add your module's information to the bnd.bnd file. For example, the configuration below adds the information for a module called My Custom View.

```
Bundle-Name: My Custom View
Bundle-SymbolicName: com.liferay.docs.my.custom.view
Bundle-Version: 1.0.0
```

3. Add a Web-ContextPath to your bnd.bnd to point to the resources for your module. For example:

```
Include-Resource:\
        META-INF/resources=src/main/resources/META-INF/resources
Web-ContextPath: /my-custom-view
```

   If you don't have a Web-ContextPath your module won't know where your resources are. The Include-Resource header points to the relative path for the module's resources.

Now that your module is configured, you can create the view next.

Implementing the View

To create a new view you must first know what kind of entities you want to select in the new view: images, videos, users, etc. The kind of entities you choose determines the specific ItemSelectorCriterion you need to use. For example if you were selecting images, you must use the ImageItemSelectorCriterion.

Next, you need to know the type of information the entity can return when it's selected. For example, if the entity returns its URL, you would use URLItemSelectorReturnType for the return type.

For a full list of the available criterion and returns types that Liferay's apps and app suite's provide see the Item Selector Criterion and Return Types reference.

Once you've determined the kinds of entities you wish to select, follow these steps to create your selection view:

1. Create an Item Selector View Component class:

   ```
   @Component(
       property = {"item.selector.view.order:Integer=200"},
       service = ItemSelectorView.class
   )
   ```

   Note that the OSGi component is registered with the property `item.selector.view.order`. The Item Selector view order (the order of the rendered tab views) is prioritized according to these settings:

   - The criteria order specified in the `getItemSelectorURL` method of the application.
   - The `item.selector.view.order` property's value for multiple views with the same criteria. The lower the value is, the more priority it has, and the sooner it will appear in the order.

2. Implement the `ItemSelectorView` interface using the *criterion* the view requires. For example, the configuration below uses the `ImageItemSelectorCriterion` class to implement the view:

   ```
   public class SampleItemSelectorView
       implements ItemSelectorView<ImageItemSelectorCriterion> {

       @Override
       public Class<ImageItemSelectorCriterion> getItemSelectorCriterionClass()
       {
           return ImageItemSelectorCriterion.class;
       }

       @Override
       public ServletContext getServletContext() {
           return _servletContext;
       }

       @Override
       public List<ItemSelectorReturnType> getSupportedItemSelectorReturnTypes() {
           return _supportedItemSelectorReturnTypes;
       }

   }
   ```

   The implementation above also sets up some methods you'll use in the steps that follow. The `getSupportedItemSelectorReturnTypes` method returns a list of *ItemSelectorReturnTypes*. You'll populate this list in a later step to specify the return types the selection view supports.

```
**Note:** If you want your new selection view to be available only when
selecting the entity for something specific such as a blog entry, replace
the `*ItemSelectorCriterion` in your `*ItemSelectorView` class with the
`*ItemSelectorCriterion` class you wish to use, such as the
[`BlogsItemSelectorCriterion` class](@app-ref@/collaboration/latest/javadocs/com/liferay/blogs/item/selector/criterion/BlogsItemSelectorCriterion.html).
```

---

3. Configure the title, search options, and visibility settings for the selection view:

   An example configuration is shown below for a selection view called `Sample Selector`:

```
@Override
public String getTitle(Locale locale) {
    return "Sample Selector";
}

@Override
public boolean isShowSearch() {
    return false;
}

@Override
public boolean isVisible(ThemeDisplay themeDisplay) {
    return true;
}
```

The following methods are demonstrated above:

- `getTitle` method: returns the localized title of the tab to display in the Item Selector dialog.
- `isShowSearch()` method: returns whether the Item Selector view should show the search field.

```
**Note:** To implement search, return `true` for this method. The
`renderHTML` method, covered in the next section, indicates whether a user
performed a search based on the value of the `search` parameter. Then the
keywords the user searched can be obtained as follows:

    String keywords = ParamUtil.getString(request, "keywords");
```

```
- [`isVisible()` method](@app-ref@/collaboration/latest/javadocs/com/liferay/item/selector/ItemSelectorView.html#isVisible-com.liferay.portal.kernel.theme.T
):
returns whether the Item Selector view is visible. In most cases, you'll
want to set this to `true`. You can use this method to add conditional logic
to disable the view.
```

4. Next, set the render settings for your view, using the `renderHTML` method. The example below points to a JSP file to render the view:

```
@Override
public void renderHTML(
    ServletRequest request, ServletResponse response,
    ImageItemSelectorCriterion itemSelectorCriterion,
    PortletURL portletURL, String itemSelectedEventName,
    boolean search
)
throws IOException, ServletException {

    request.setAttribute(_ITEM_SELECTED_EVENT_NAME,
    itemSelectedEventName);

    ServletContext servletContext = getServletContext();

    RequestDispatcher requestDispatcher =
    servletContext.getRequestDispatcher("/sample.jsp");

    requestDispatcher.include(request, response);
}
```

The renderHTML methods passes the *ItemSelectorCriterion required to display the selection view. Next, the portletURL, used to invoke the Item Selector, is passed. Then the itemSelectedEventName is passed. This is the event name that the caller listens for. When an element is selected, the view fires a JavaScript event with this name. Finally, a search boolean is passed, specifying when the view should render search results. When the user performs a search, this boolean should be set to true.

Note that the itemSelectedEventName is passed as a request attribute, so it can be used in the view markup.

The view markup is specified this way:

```
RequestDispatcher requestDispatcher =
        servletContext.getRequestDispatcher("/sample.jsp");
```

Although the example uses JSPs, you can use another language such as FreeMarker to render the markup.

5. Use the @Reference annotation to reference your module's class for the setServletContext method.

   Below is an example configuration:

```
@Reference(
    target =
    "(osgi.web.symbolicname=com.liferay.item.selector.sample.web)",
    unbind = "-"
)
```

   public void setServletContext(ServletContext servletContext) { _servletContext = servletContext; }

   The target parameter is used to specify the available services for the servlet context. In this case, it specifies the com.liferay.selector.sample.web class as the default value, using the osgi.web.symbolicname property. Finally, the unbind = _ parameter specifies that there is no unbind method for this module. A method is defined to set the servlet context as well.

6. Finally, populate the _supportedItemSelectorReturnTypes list specified in step 2 with the return types that this view supports.

   The example below adds the URLItemSelectorReturnType class and FileEntryItemSelectorReturnType class to the list of supported return types, but you could use more return types if the view could return them. More return types means that the view is more reusable:

```
private static final List<ItemSelectorReturnType>
    _supportedItemSelectorReturnTypes =
    Collections.unmodifiableList(
        ListUtil.fromArray(
            new ItemSelectorReturnType[] {
                new FileEntryItemSelectorReturnType(),
                new URLItemSelectorReturnType()
            }));

 private ServletContext _servletContext;
```

   The servlet context variable is declared at the bottom of the file.

As a complete example, below is the full code for the FlickrItemSelectorView class:

```
public class FlickrItemSelectorView
        implements ItemSelectorView<FlickrItemSelectorCriterion> {

        @Override
        public Class<FlickrItemSelectorCriterion> getItemSelectorCriterionClass() {
                return FlickrItemSelectorCriterion.class;
        }

        @Override
        public List<ItemSelectorReturnType> getSupportedItemSelectorReturnTypes() {
                return _supportedItemSelectorReturnTypes;
        }

        @Override
        public String getTitle(Locale locale) {
                return FlickrItemSelectorView.class.getName();
        }

        @Override
        public boolean isShowSearch() {
                return false;
        }

        @Override
        public boolean isVisible(ThemeDisplay themeDisplay) {
                return true;
        }

        @Override
        public void renderHTML(
                        ServletRequest request, ServletResponse response,
                        FlickrItemSelectorCriterion flickrItemSelectorCriterion,
                        PortletURL portletURL, String itemSelectedEventName,
                        boolean search)
                throws IOException {

                PrintWriter printWriter = response.getWriter();

                printWriter.print(
                        "<html>" + FlickrItemSelectorView.class.getName() +
                        "</html>");
        }

        private static final List<ItemSelectorReturnType>
                _supportedItemSelectorReturnTypes =
                Collections.unmodifiableList(
                        ListUtil.fromArray(
                                new ItemSelectorReturnType[] {
                                        new TestURLItemSelectorReturnType()
                                }));

}
```

The diagram below illustrates how the Item Selector's API works (right-click to view larger image):
Once you've implemented your Item Selector view, you must create the view markup.

Writing your View Markup

You've implemented your view, specifying the criteria and return types, along with important configuration information, such as how to render the view. All that's left is to write the markup for your selection view.

Naturally, the markup for your selection view will vary greatly depending on the requirements of your app. You can use taglibs, AUI components, or even pure HTML and JavaScript if you prefer, to write your view. Regardless of the approach you choose to create your view, the view must do two key things:

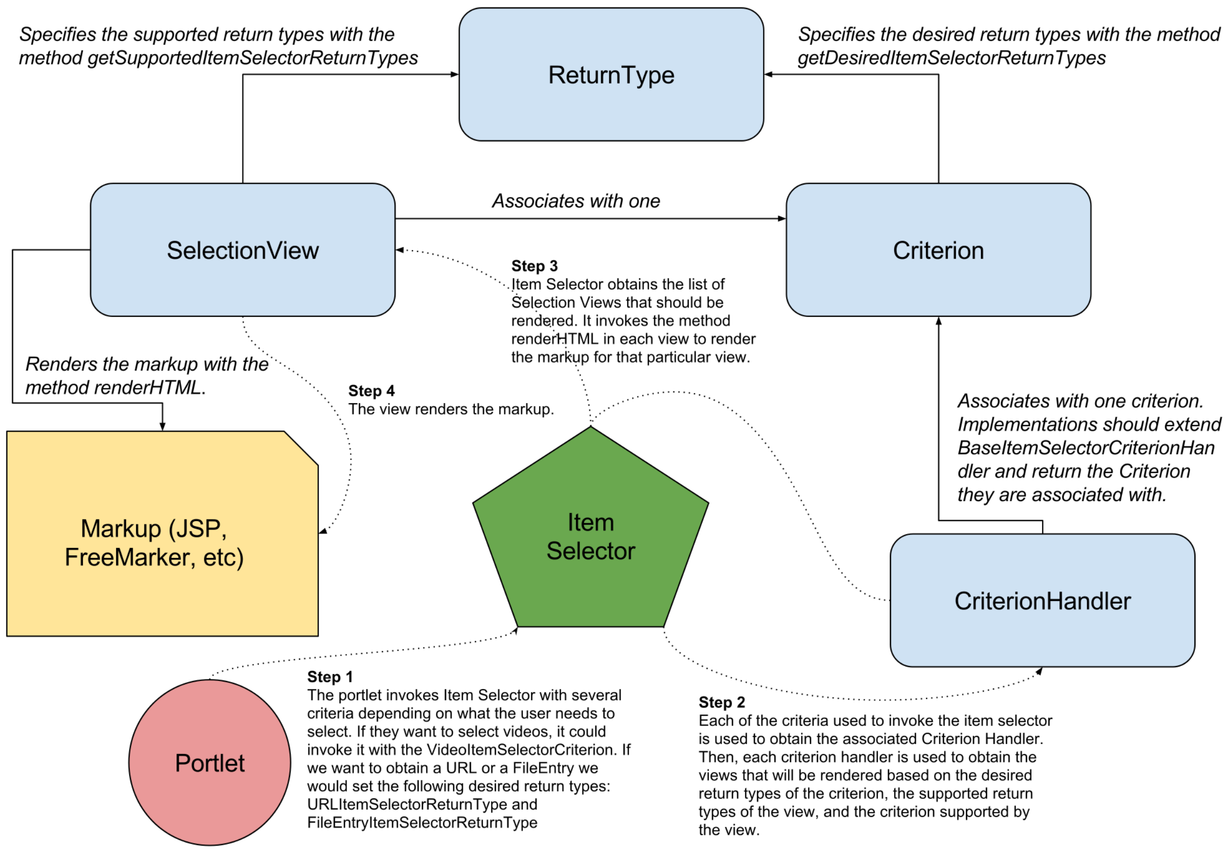

Specifies the supported return types with the method getSupportedItemSelectorReturnTypes
ReturnType
Specifies the desired return types with the method getDesiredItemSelectorReturnTypes
SelectionView
Associates with one
Criterion
Step 3
Item Selector obtains the list of Selection Views that should be rendered. It invokes the method renderHTML in each view to render the markup for that particular view.
Renders the markup with the method renderHTML.
Step 4
The view renders the markup.
Associates with one criterion. Implementations should extend BaseItemSelectorCriterionHandler and return the Criterion they are associated with.
Markup (JSP, FreeMarker, etc)
Item Selector
CriterionHandler
Step 1
The portlet invokes Item Selector with several criteria depending on what the user needs to select. If they want to select videos, it could invoke it with the VideoItemSelectorCriterion. If we want to obtain a URL or a FileEntry we would set the following desired return types: URLItemSelectorReturnType and FileEntryItemSelectorReturnType
Portlet
Step 2
Each of the criteria used to invoke the item selector is used to obtain the associated Criterion Handler. Then, each criterion handler is used to obtain the views that will be rendered based on the desired return types of the criterion, the supported return types of the view, and the criterion supported by the view.

Figure 91.4: Item Selector views are determined by the desired return types of the criterion, the supported return types of the view, and the criterion supported by the view.

- Render the entities for the user to select
- Contain JavaScript logic that passes the information specified by the Item Selector return type via a JavaScript event when an entity is selected.

If you're following the example in the last section, the JavaScript event name has been passed as a request attribute in the `renderHTML` method of the `*ItemSelectorView` class, so it can be used in the view markup as follows:

```
Liferay.fire(
        `<%= {ITEM_SELECTED_EVENT_NAME} %>',

        {
            data:{
                the-data-your-client-needs-according-to-the-return-type
            }
        }
);
```

Below is the complete `Layouts.jsp` view markup for the `com.liferay.layout.item.selector.web` module: Some Java imports are defined first:

```
<%
LayoutItemSelectorViewDisplayContext layoutItemSelectorViewDisplayContext =
(LayoutItemSelectorViewDisplayContext)request.getAttribute(
BaseLayoutsItemSelectorView.LAYOUT_ITEM_SELECTOR_VIEW_DISPLAY_CONTEXT);

LayoutItemSelectorCriterion layoutItemSelectorCriterion =
layoutItemSelectorViewDisplayContext.getLayoutItemSelectorCriterion();

Portlet portlet = PortletLocalServiceUtil.getPortletById(
company.getCompanyId(), portletDisplay.getId());
%>
```

Note that the `LayoutItemSelectorViewDisplayContext` is an optional class that contains additional information about the criteria and view, but it isn't required.

The snippet below imports a CSS file for styling and places it in the <head> of the page:

```
<liferay-util:html-top>
        <link href="<%= PortalUtil.getStaticResourceURL(
        request, application.getContextPath() + "/css/main.css",
        portlet.getTimestamp())
        %>" rel="stylesheet" type="text/css" />
</liferay-util:html-top>
```

You can learn more about using the `liferay-util` taglibs in the Using the Liferay Util Taglib tutorial.

This snippet creates the UI to display the layout entities. It uses the `liferay-layout:layouts-tree` taglib along with the Lexicon design language to create cards:

```
<div class="container-fluid-1280 layouts-selector">
        <div class="card-horizontal main-content-card">
                <div class="card-row card-row-padded">
                        <liferay-layout:layouts-tree
                                checkContentDisplayPage="<%= layoutItemSelectorCriterion.isCheckDisplayPage() %>"
                                draggableTree="<%= false %>"
                                expandFirstNode="<%= true %>"
                                groupId="<%= scopeGroupId %>"
                                portletURL="<%= layoutItemSelectorViewDisplayContext.getEditLayoutURL() %>"
                                privateLayout="<%= layoutItemSelectorViewDisplayContext.isPrivateLayout() %>"
                                rootNodeName="<%= layoutItemSelectorViewDisplayContext.getRootNodeName() %>"
                                saveState="<%= false %>"
                                selectedLayoutIds="<%= layoutItemSelectorViewDisplayContext.getSelectedLayoutIds() %>"
                                selPlid="<%= layoutItemSelectorViewDisplayContext.getSelPlid() %>"
                                treeId="treeContainer"
                        />
                </div>
        </div>
</div>
```

The configuration above renders the UI shown in the figure below:

This portion of the `aui:script` returns the path for the page:

```
<aui:script use="aui-base">
        var LString = A.Lang.String;

        var getChosenPagePath = function(node) {
                var buffer = [];

                if (A.instanceOf(node, A.TreeNode)) {
                        var labelText = LString.escapeHTML(node.get('labelEl').text());

                        buffer.push(labelText);

                        node.eachParent(
                                function(treeNode) {
```

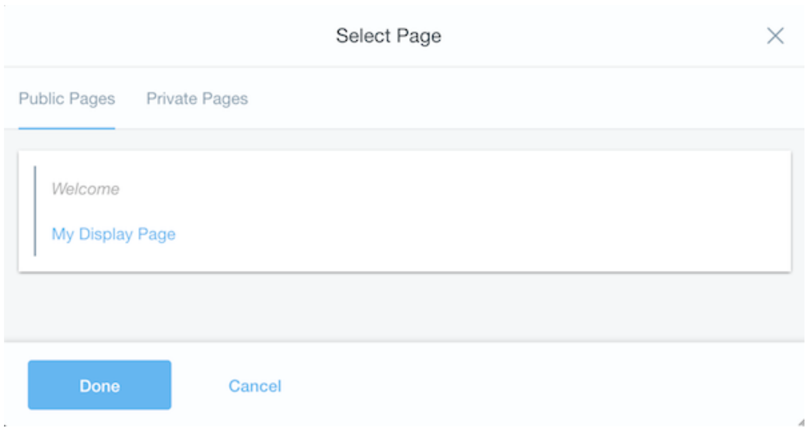


Figure 91.5: The Layouts Item Selector view uses Lexicon and Liferay Layout taglibs to create the UI.

```
                                var labelEl = treeNode.get('labelEl');

                                if (labelEl) {
                                        labelText = LString.escapeHTML(labelEl.text());

                                        buffer.unshift(labelText);
                                }
                        }
                );
        }

        return buffer.join(' > ');
};
```

The snippet below passes the return type data when the layout(entity) is selected. In particular, take note of the `url` and `uuid` variables, which retrieve the URL or UUID for the layout:

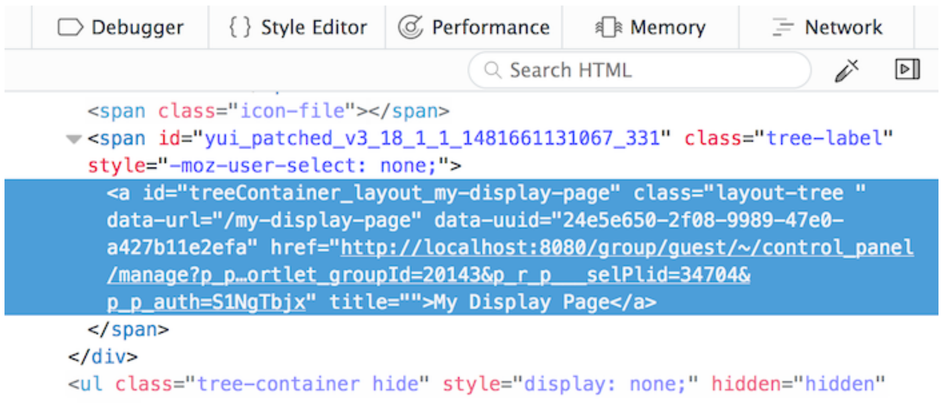
```
var setSelectedPage = function(event) {
        var disabled = true;

        var messageText = '<%= UnicodeLanguageUtil.get(request, "there-is-no-selected-page") %>';

        var lastSelectedNode = event.newVal;

        var labelEl = lastSelectedNode.get('labelEl');

        var link = labelEl.one('a');

        var url = link.attr('data-url');
        var uuid = link.attr('data-uuid');

        var data = {};

        if (link && url) {
                disabled = false;

                data.layoutpath = getChosenPagePath(lastSelectedNode);
```

This checks if the return type information is a URL or a UUID; then it sets the value for the JSON object's data attribute accordingly:

```
                <c:choose>
                        <c:when test="<%= Objects.equals(layoutItemSelectorViewDisplayContext.getItemSelectorReturnTypeName(), URLItemSelectorReturn
                                data.value = url;
```

```
                    </c:when>
                    <c:when test="<%= Objects.equals(layoutItemSelectorViewDisplayContext.getItemSelectorReturnTypeName(), UUIDItemSelectorRetur
                            data.value = uuid;
                    </c:when>
                </c:choose>
            }

            <c:if test="<%= Validator.isNotNull(layoutItemSelectorViewDisplayContext.getCkEditorFuncNum()) %>">
                data.ckeditorfuncnum: <%= layoutItemSelectorViewDisplayContext.getCkEditorFuncNum() %>;
            </c:if>
```

The data-url and data-uuid attributes are in the HTML markup for the Layouts Item Selector. The HTML markup for an instance of the Layouts Item Selector is shown below:



```
<span class="icon-file"></span>
<span id="yui_patched_v3_18_1_1_1481661131067_331" class="tree-label"
style="-moz-user-select: none;">
  <a id="treeContainer_layout_my-display-page" class="layout-tree "
  data-url="/my-display-page" data-uuid="24e5e650-2f08-9989-47e0-
  a427b11e2efa" href="http://localhost:8080/group/guest/~/control_panel
  /manage?p_p…ortlet_groupId=20143&p_r_p___selPlid=34704&
  p_p_auth=S1NgTbjx" title="">My Display Page</a>
</span>
</div>
<ul class="tree-container hide" style="display: none;" hidden="hidden"
```

Figure 91.6: The URL and UUID can be seen in the data-url and data-uuid attributes of the Layout Item Selector's HTML markup.

The last line adds the CKEditorFuncNum for the editor to the JSON object's data attribute.

The JavaScript trigger event, specified in the Item Selector return type, is fired, passing the data JSON object with the required return type information:

```
            Liferay.Util.getOpener().Liferay.fire(
                '<%= layoutItemSelectorViewDisplayContext.getItemSelectedEventName() %>',
                {
                    data: data
                }
            );
        };
```

Finally, the layout is set to the selected page:

```
        var container = A.one('#<portlet:namespace />treeContainerOutput');

        if (container) {
            container.swallowEvent('click', true);

            var tree = container.getData('tree-view');

            tree.after('lastSelectedChange', setSelectedPage);
        }
</aui:script>
```

Your new selection view is automatically rendered by the Item Selector in every portlet that uses the criterion and return types you defined, without modifying anything in those portlets.

Now you know how to create custom views for the Item Selector!

Related Topics

Selecting Entities Using the Item Selector
    Creating Custom Item Selector Entities

CHAPTER 92

ADAPTIVE MEDIA

The Adaptive Media app on Liferay Marketplace lets administrators tailor the quality of images in Liferay DXP to the device viewing those images. For information on using this app, see the Adaptive Media user guide.

But what if you want to leverage Adaptive Media in your own app? Then you're in the right place! The tutorials in this section explain how to use adapted images in your app. You'll also learn how to change Adaptive Media's image processing.

Onwards!

92.1 Displaying Adapted Images in Your App

To display adapted images in your apps, Adaptive Media offers a convenient taglib in the module `com.liferay.adaptive.media.image.taglib`. This taglib only has one mandatory attribute: `fileVersion`. This attribute indicates the file version of the adapted image that you want to display. You can also add as many attributes as needed, such as `class`, `style`, `data-sample`, and so on. Any attributes you add are then added to the adapted images in the markup the taglib renders.

This tutorial uses the Adaptive Media Samples app to show you how to use this taglib. When added to a page, this app displays all the adapted images from the current site's Documents and Media app, provided that Adaptive Media image resolutions and Documents and Media images exist.

Follow these steps to use the taglib:

1. Include the module taglib dependency in your project. If you're using Gradle, for example, you must add the following line in your project's `build.gradle` file:

   ```
   provided group: "com.liferay", name: "com.liferay.adaptive.media.image.taglib", version: "1.0.0"
   ```

   For example, the Adaptive Media Samples app's `build.gradle` file contains this taglib.

2. Declare the taglib in your JSP:

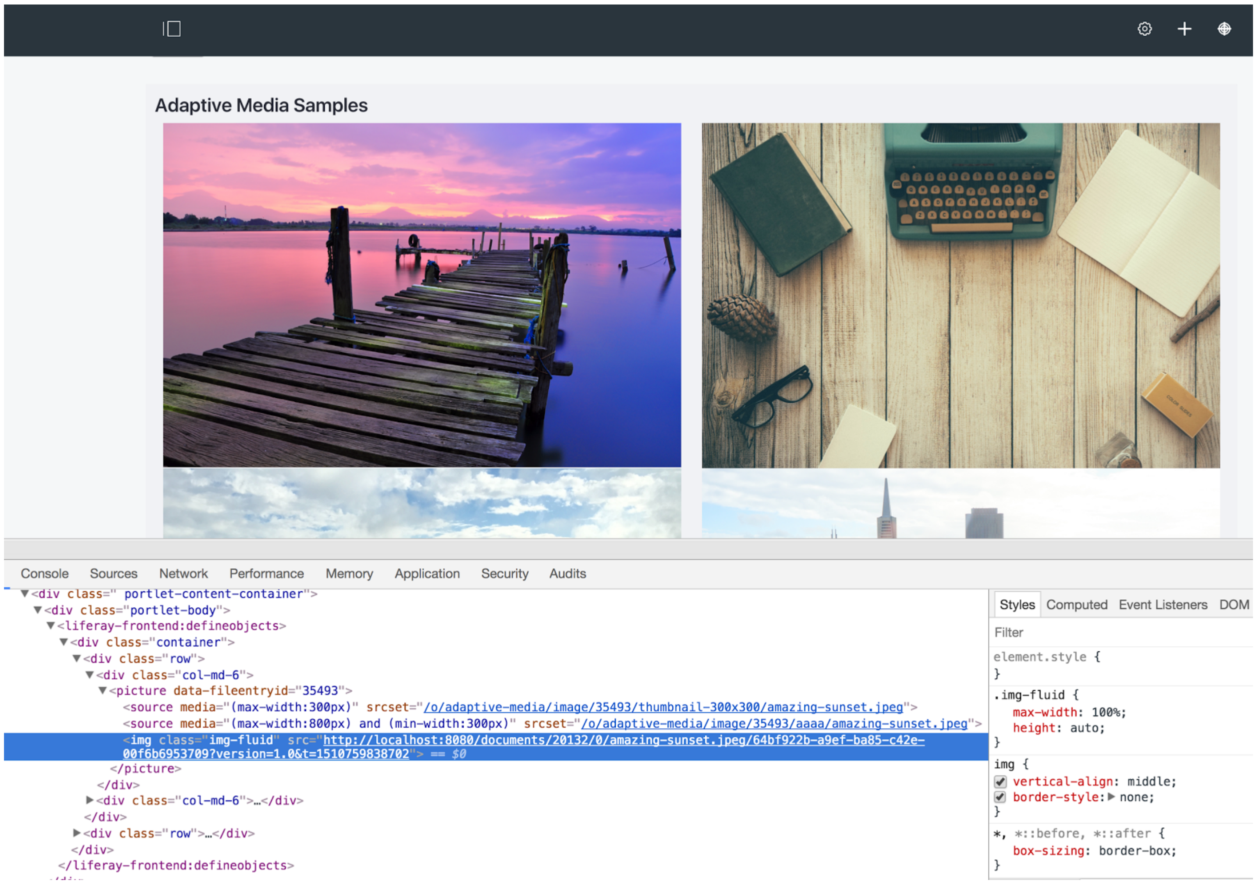
   ```
   <%@ taglib uri="http://liferay.com/tld/adaptive-media-image" prefix="liferay-adaptive-media" %>
   ```

   For example, the Adaptive Media Samples app's `init.jsp` declares all the taglibs the app needs.

3. Use the taglib wherever you want the adapted image to appear in your app's JSP files:

```
<liferay-adaptive-media:img class="img-fluid" fileVersion="<%= fileEntry.getFileVersion() %>" />
```

For example, the Adaptive Media Samples app's `view.jsp` uses the taglib to display the adapted images in a grid with the `col-md-6` column container class. Looking at the markup the app generates, you can see that it uses the `<picture>` tag as described in the article Creating Content with Adapted Images.



```
▼<div class=" portlet-content-container">
  ▼<div class="portlet-body">
    ▼<liferay-frontend:defineobjects>
      ▼<div class="container">
        ▼<div class="row">
          ▼<div class="col-md-6">
            ▼<picture data-fileentryid="35493">
                <source media="(max-width:300px)" srcset="/o/adaptive-media/image/35493/thumbnail-300x300/amazing-sunset.jpeg">
                <source media="(max-width:800px) and (min-width:300px)" srcset="/o/adaptive-media/image/35493/aaaa/amazing-sunset.jpeg">
                <img class="img-fluid" src="http://localhost:8080/documents/20132/0/amazing-sunset.jpeg/64bf922b-a9ef-ba85-c42e-
                00f6b6953709?version=1.0&t=1510759838702"> == $0
              </picture>
            </div>
          ▶<div class="col-md-6">…</div>
          </div>
        ▶<div class="row">…</div>
        </div>
      </liferay-frontend:defineobjects>
```

```
element.style {
}
.img-fluid {
   max-width: 100%;
   height: auto;
}
img {
✓ vertical-align: middle;
✓ border-style:▶ none;
}
*, *::before, *::after {
   box-sizing: border-box;
}
```

Figure 92.1: The Adaptive Media Samples app shows all the site's adapted images.

Well done! Now you know how to display adapted images in your app.

Related Topics

Finding Adapted Images

Changing Adaptive Media's Image Processing

Adapting Your Media Across Multiple Devices

92.2 Finding Adapted Images

In most cases, you can rely on the Adaptive Media taglib to display adapted images in your app. This taglib uses the file version you give it to query Adaptive Media's finder API and display the adapted image appropriate for the device making the request. If you need more control, however, you can write your own query with the API instead of using the taglib. For example, if you have an app that needs a specific image in a specific dimension, it's best to query Adaptive Media's finder API directly. You can then display the image however you like (e.g., with an HTML `<img>` tag).

Adaptive Media's finder API lets you write queries that get adapted images based on certain search criteria and filters. For example, you can get adapted images that match a file version or resolution, or are ordered by an attribute like image width. You can even get adapted images that match approximate attribute values (*fuzzy* attributes).

This tutorial shows you how to call Adaptive Media's API to get adapted images in your app. First, you'll learn how to construct such API calls.

Calling Adaptive Media's API

The entry point to Adaptive Media's API is the `AMImageFinder` interface. To use it, you must first inject the OSGi component in your class, which must also be an OSGi component, as follows:

```
@Reference
private AMImageFinder _amImageFinder;
```

This makes an `AMImageFinder` instance available. It has one method, `getAdaptiveMediaStream`, that returns a stream of `AdaptiveMedia` objects. This method takes a `Function` that creates an `AMQuery` (the query for adapted images) via `AMImageQueryBuilder`, which can search adapted images based on different attributes (e.g., width, height, order, etc.). The `AMImageQueryBuilder` methods you call depend on the exact query you want to construct.

For example, here's a general `getAdaptiveMediaStream` call:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.methodToCall(arg).done());
```

The argument to `getAdaptiveMediaStream` is a lambda expression that returns an `AMQuery` constructed via `AMImageQueryBuilder`. Note that `methodToCall(arg)` is a placeholder for the `AMImageQueryBuilder` method you want to call and its argument. The exact call depends on the criteria you want to use to select adapted images. The `done()` call that follows this, however, isn't a placeholder–it creates and returns the `AMQuery` regardless of which `AMImageQueryBuilder` methods you call.

For more information on creating `AMQuery` instances, see the Javadoc for `AMImageQueryBuilder`.

Next, you'll see specific examples of constructing calls that get adapted images.

Getting Adapted Images for a Specific File Version

To get adapted images for a specific file version, you must call the `AMImageQueryBuilder` method `forFileVersion` with a `FileVersion` object as an argument:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(fileVersion).done());
```

To get the adapted images for the latest approved file version, use the `forFileEntry` method with a `FileEntry` object:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileEntry(fileEntry).done());
```

Note that these calls only return the adapted images for enabled image resolutions. Adapted images for disabled resolutions aren't included in the stream. To retrieve all adapted images regardless of any image resolution's status, you must also call the withConfigurationStatus method with the constant AMImageQueryBuilder.ConfigurationStatus.ANY:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(fileVersion)
            .withConfigurationStatus(AMImageQueryBuilder.ConfigurationStatus.ANY).done());
```

To get adapted images for a specific file version when the image resolution is disabled, make the same call but instead use the constant AMImageQueryBuilder.ConfigurationStatus.DISABLED:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(fileVersion)
            .withConfigurationStatus(AMImageQueryBuilder.ConfigurationStatus.DISABLED).done());
```

Next, you'll learn how to get adapted images for a specific image resolution.

Getting the Adapted Images for a Specific Image Resolution

By providing an image resolution's UUID to AMImageFinder, you can get that resolution's adapted images. This UUID is defined when adding the resolution in the Adaptive Media app. To get a resolution's adapted images, you must pass that resolution's UUID to the forConfiguration method.

For example, this code gets the adapted images that match a file version, and belong to an image resolution with the UUID hd-resolution. It returns the adapted images regardless of whether the resolution is enabled or disabled:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinder.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(fileVersion)
            .forConfiguration("hd-resolution").done());
```

Next, you'll learn how to return adapted images in a specific order.

Getting Adapted Images in a Specific Order

It's also possible to define the order in which getAdaptiveMediaStream returns adapted images. To do this, call the orderBy method with your sort criteria just before calling the done() method. The orderBy method takes two arguments: the first specifies the image attribute to sort by (e.g., width/height), while the second specifies the sort order (e.g., ascending/descending).

For example, this code gets all the adapted images regardless of whether the image resolution is enabled, and puts them in ascending order by the image width:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinderImpl.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(_fileVersion)
            .withConfigurationStatus(AMImageQueryBuilder.ConfigurationStatus.ANY)
            .orderBy(AMImageAttribute.AM_IMAGE_ATTRIBUTE_WIDTH, AMImageQueryBuilder.SortOrder.ASC)
            .done());
```

The orderBy arguments AMImageAttribute.AM_IMAGE_ATTRIBUTE_WIDTH and AMImageQueryBuilder.SortOrder.ASC specify the image width and ascending sort, respectively. You can alternatively use AMImageAttribute.AM_IMAGE_ATTRIBUTE_HEIG to sort by image height, and AMImageQueryBuilder.SortOrder.DESC to perform a descending sort.

Next, you'll learn how to specify approximate attribute values when getting adapted images.

Getting Adapted Images with Fuzzy Attributes

Adaptive Media also lets you get adapted images that match *fuzzy attributes* (approximate attribute values). For example, fuzzy attributes let you ask for adapted images whose height is around 200px, or whose size is around 100kb. The API returns a stream with elements ordered by how close they are to the specified attribute. For example, imagine that there are four image resolutions that have adapted images with the heights 150px, 350px, 600px, and 900px. Searching for adapted images whose height is approximately 400px returns this order in the stream: 350px, 600px, 150px, 900px.

So how close, exactly, is *close*? It depends on the attribute. In the case of width, height, and length, a numeric comparison orders the images. In the case of content type, file name, or UUID, the comparison is more tricky because these attributes are strings and thus delegated to the Java `String.compareTo` method.

To specify a fuzzy attribute, call the `with` method with your search criteria just before calling the `done()` method. The `with` method takes two arguments: the image attribute, and that attribute's approximate value. For example, this code gets adapted images whose height (`AMImageAttribute.AM_IMAGE_ATTRIBUTE_HEIGHT`) is approximately 400px:

```
Stream<AdaptiveMedia<AMImageProcessor>> adaptiveMediaStream =
    _amImageFinderImpl.getAdaptiveMediaStream(
        amImageQueryBuilder -> amImageQueryBuilder.forFileVersion(_fileVersion)
            .with(AMImageAttribute.AM_IMAGE_ATTRIBUTE_HEIGHT, 400).done());
```

To search for image width instead, use `AMImageAttribute.AM_IMAGE_ATTRIBUTE_WIDTH` as the first argument to the `width` method.

Using the Adaptive Media Stream

Once you have the `AdaptiveMedia` stream, you can get the information you need from it. For example, this code prints the URI for each adapted image:

```
adaptiveMediaStream.forEach(
    adaptiveMedia -> {
        System.out.println(adaptiveMedia.getURI());
    }
);
```

You can also get other values and attributes from the `AdaptiveMedia` stream. Here are a few examples:

```
// Get the InputStream
adaptiveMedia.getInputStream()

// Get the content length
adaptiveMedia.getValueOptional(AMAttribute.getContentLengthAMAttribute())

// Get the image height
adaptiveMedia.getValueOptional(AMImageAttribute.AM_IMAGE_ATTRIBUTE_HEIGHT)
```

Awesome! Now you know how to find and use adapted images.

Related Topics

Displaying Adapted Images in Your App

Changing Adaptive Media's Image Processing

Adapting Your Media Across Multiple Devices

92.3 Changing Adaptive Media's Image Scaling

As described in the Adaptive Media user guide, Adaptive Media scales images to match the image resolutions defined by the Liferay DXP administrator. The default scaling is usually suitable, but you can also customize it to your needs. Before doing so, however, you should understand how this scaling works.

Understanding Image Scaling in Adaptive Media

Adaptive Media contains an extension point that lets you replace the way it scales images. The `AMImageScaler` interface defines Adaptive Media's image scaling logic. Out of the box, Adaptive Media provides two implementations of this interface:

- `AMDefaultImageScaler`: The default image scaler. It's always enabled and uses `java.awt` for its image processing and scaling.
- `AMGIFImageScaler`: A scaler that works only with GIF images. It depends on the installation of the external tool gifsicle in the Liferay DXP instance. This scaler must be enabled in *Control Panel → System Settings*.

You must register image scalers in Liferay DXP's OSGi container using the `AMImageScaler` interface. Each scaler must also set the `mime.type` property to the MIME type it handles. For example, if you set a scaler's MIME type to `image/jpeg`, then that scaler can only handle `image/jpeg` images. If you specify the special MIME type `*`, the scaler can process any image. Note that the `AMDefaultImageScaler` is registered using `mime.type=*`, while the `AMGIFImageScaler` is registered using `mime.type=image/gif`. Both scalers, like all scalers, implement `AMImageScaler`.

You can add as many image scalers as you need, even for the same MIME type. Even so, Adaptive Media uses only one scaler per image, using this process to determine the best one:

1. Select only the image scalers registered with the same MIME type as the image.
2. Select the enabled scalers from those selected in the first step (the `AMImageScaler` method `isEnabled()` returns `true` for enabled scalers).
3. Of the scalers selected in the second step, select the scaler with the highest `service.ranking`.

If these steps return no results, they're repeated, but the first step uses the special MIME type `*`. Also note that if an image scaler is registered for specific MIME types and has a higher `service.ranking`, it's more likely to be chosen than if it's registered for the special MIME type `*` or has a lower `service.ranking`.

Creating an Image Scaler

Now that you know how Adaptive Media scales images, you'll learn how to customize this scaling. As an example, you'll see a sample image scaler that customizes the scaling of PNG images.

Follow these steps to create a custom image scaler:

1. Create your scaler class to implement `AMImageScaler`. You must also annotate your scaler class with `@Component`, setting `mime.type` properties for each of the scaler's MIME types, and registering an `AMImageScaler` service. If there's more than one scaler for the same MIME type, you must also set the `@Component` annotation's `service.ranking` property. For your scaler to take precedence over other scalers of the same MIME type, its service ranking property must be higher than that of the other scalers. If `service.ranking` isn't set, it defaults to `0`.

```
**Note:** The `service.ranking` property isn't set for the image scalers
included with Adaptive Media (`AMDefaultImageScaler` and
`AMGIFImageScaler`). Their service ranking therefore defaults to `0`. To
replace either scaler, you must set your scaler to the same MIME type and
give it a service ranking higher than `0`.
```

```
For example, this sample image scaler scales PNG and x-PNG images, and has a
service ranking of `100`:

    @Component(
        immediate = true,
        property = {"mime.type=image/png", "mime.type=image/x-png", "service.ranking:Integer=100"},
        service = {AMImageScaler.class}
    )
    public class SampleAMPNGImageScaler implements AMImageScaler {...

This requires these imports:

    import com.liferay.adaptive.media.image.scaler.AMImageScaler;
    import org.osgi.service.component.annotations.Component;
```

2. Implement the `isEnabled()` method to return true when you want to enable the scaler. In many cases, you always want the scaler enabled, so you can simply return true in this method. This is the case with the example `SampleAMPNGImageScaler`:

   ```
   @Override
   public boolean isEnabled() {
       return true;
   }
   ```

   This method gets more interesting when the scaler depends on other tools or features. For example, the `isEnabled()` method in `AMGIFImageScaler` determines whether gifsicle is enabled. This scaler must only be enabled when the tool it depends on, gifsicle, is also enabled:

   ```
   @Override
   public boolean isEnabled() {
       return _amImageConfiguration.gifsicleEnabled();
   }
   ```

3. Implement the `scaleImage` method. This method contains the scaler's business logic, and must return an `AMImageScaledImage` instance. For example, the example `scaleImage` implementation in `SampleAMPNGImageScaler` uses `AMImageConfigurationEntry` to get the maximum height and width values for the scaled image, and `FileVersion` to get the image to scale. The scaling is done with the help of a private inner class, assuming that the methods `_scalePNG`, `_getScalePNGHeight`, `_getScalePNGWidth`, and `_getScalePNGSize` implement the actual scaling:

   ```
   @Override
   public AMImageScaledImage scaleImage(FileVersion fileVersion,
       AMImageConfigurationEntry amImageConfigurationEntry) {

       Map<String, String> properties = amImageConfigurationEntry.getProperties();

       int maxHeight = GetterUtil.getInteger(properties.get("max-height"));
       int maxWidth = GetterUtil.getInteger(properties.get("max-width"));
   ```

```
        try {
            InputStream inputStream =
                _scalePNG(fileVersion.getContentStream(false), maxHeight, maxWidth);

            int height = _getScalePNGHeight();
            int width = _getScalePNGWidth();
            long size = _getScalePNGSize();

            return new AMImageScaledImageImpl(inputStream, height, width, size);
        }
        catch (PortalException pe) {
            throw new AMRuntimeException.IOException(pe);
        }
    }

    private class AMImageScaledImageImpl implements AMImageScaledImage {

        @Override
        public int getHeight() {
            return _height;
        }

        @Override
        public InputStream getInputStream() {
            return _inputStream;
        }

        @Override
        public long getSize() {
            return _size;
        }

        @Override
        public int getWidth() {
            return _width;
        }

        private AMImageScaledImageImpl(InputStream inputStream, int height,
            int width, long size) {

            _inputStream = inputStream;
            _height = height;
            _width = width;
            _size = size;
        }

        private final int _height;
        private final InputStream _inputStream;
        private final long _size;
        private final int _width;

    }
```

This requires these imports:

```
import com.liferay.adaptive.media.exception.AMRuntimeException;
import com.liferay.adaptive.media.image.configuration.AMImageConfigurationEntry;
import com.liferay.adaptive.media.image.scaler.AMImageScaledImage;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.repository.model.FileVersion;
import com.liferay.portal.kernel.util.GetterUtil;
import java.io.InputStream;
import java.util.Map;
```

Great! Now you know how to write your own image scalers.

Related Topics

Displaying Adapted Images in Your App
    Finding Adapted Images
    Adapting Your Media Across Multiple Devices

Chapter 93

Liferay Forms

Many modern websites rely on forms for their functionality. Liferay's Forms application is maturing rapidly, offering rich out-of-the-box functionality. Many use cases, from the simplest to the most complex, can be met using the Forms application as it is. However, why not make the application behave exactly as you'd like it to? Just because something is good, doesn't mean it's perfect for your specific use case. Liferay's Forms solution can be adapted to your specific needs. You can even develop forms using its API.

In this section, learn to extend the Forms application's functionality and leverage its APIs.

- Build your own field types for Liferay's Forms application.
  - Create field types that look just like the built-in field types (both their source code and their UI appearance).
  - Add custom configuration options to your form field types.
- Use annotations to leverage the Dynamic Data Mapping (DDM) API and build reusable forms quickly.
  - Create forms using annotations.
  - Create fields and reusable fieldsets using annotations.
  - Configure form rules using annotations.

Chapter 94

Form Field Types

The Forms application contains many highly configurable field types out-of-the-box. Most use cases will be met with one of the existing field types.

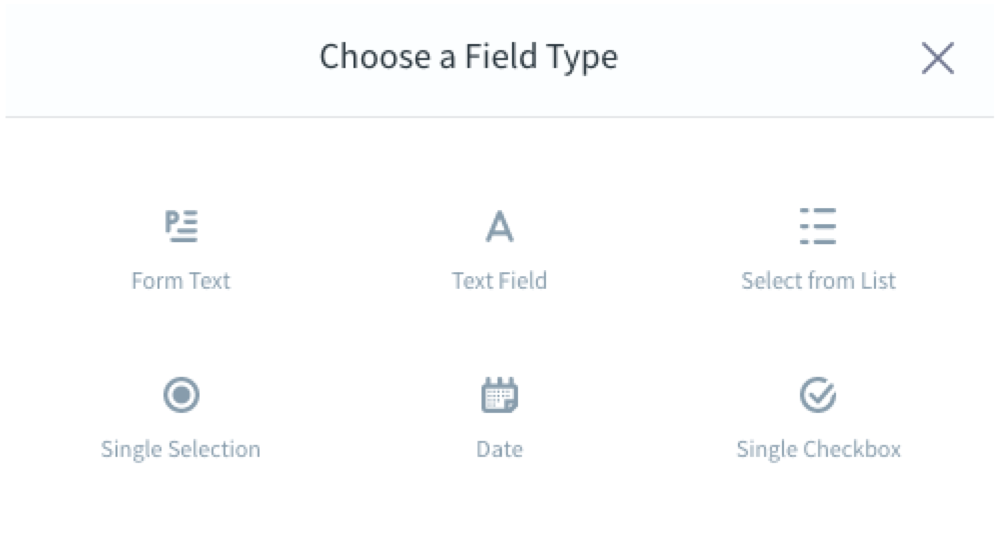


Figure 94.1: The Forms application has useful out-of-the-box field types, but you can add your own if you need to.

If you're reading this, however, your use case probably wasn't met with the default field types. For example, perhaps you need a color picker field. You could create a select field that lists the color options, but some users don't know that *gamboge* is the color of spicy mustard (maybe a little darker), and anyway, seeing colors is much more interesting than listing them. Another example is a dedicated *time* field. You can use a text field and add a tip to tell users something like *enter the time in the format hour*`:minute`, but some users will still enter something indecipherable, like *8:88*. Instead, add a *time* field to Liferay DXP's Forms application. You can think of other uses where it's best to break free of the mold of existing field types and create your own that serve your needs best. Keep reading to find out how.

In these tutorials, learn to

- create a module that adds a *Time* form field type with a timepicker
- add custom configuration options to your field types

Example project: The source code for the example *time* project developed in these tutorials can be downloaded for your convenience. Click here to begin downloading the source code zip file.

Before getting started, learn what Liferay DXP's field types consist of.

94.1 Anatomy of a Field Type Module

The `dynamic-data-mapping-type-*` modules in Liferay DXP's source code (inside the *Forms and Workflow* application suite) are good templates to follow when developing your own field types. For example, look at the directory structure of the `dynamic-data-mapping-type-paragraph` module (version 2.0.7) in the Forms and Workflow application suite:

```
bnd.bnd
build.gradle
src
└── main
    ├── java
    │   └── com
    │       └── liferay
    │           └── dynamic
    │               └── data
    │                   └── mapping
    │                       └── type
    │                           └── paragraph
    │                               └── internal
    │                                   ├── ParagraphDDMFormFieldRenderer.java
    │                                   ├── ParagraphDDMFormFieldType.java
    │                                   └── ParagraphDDMFormFieldTypeSettings.java
    └── resources
        ├── content
        │   ├── Language.properties
        │   └── Language_xx_XX.properties
        │   └── ...
        └── META-INF
            └── resources
                ├── config.js
                ├── paragraph_field.js
                ├── paragraph.soy
                └── paragraph.soy.js
```

Custom field type modules are nearly identical in structure to those included in Liferay DXP, as presented above. You won't need a `*TypeSettings` class in your initial module (see the tutorial on adding settings to your form field types to learn more about `*TypeSettings`), and the `*.soy.js` is generated from the `*.soy` file at compile time. These are the Java classes and resources you'll need to create:

- `*DDMFormFieldRenderer.java`: Controls the rendering of the template. Sets the language, declares the namespace, and loads the template resources on activation of the Component. Extending the abstract class that implements the `DDMFormFieldRenderer` makes your work here easier.
- `*DDMFormFieldType.java`: Define the form field type in the backend. If you extend the abstract class that implements the interface, you automatically include the default form configuration options for your form field type. In that case, override the interface's `getName` method and you're done. To see the default configuration options your form field type will inherit, look at the `DefaultDDMFormFieldTypeSettings` class in the `dynamic-data-mapping-form-field-type` module.
- `config.js`: Autogenerated if you use Blade CLI, `config.js` defines the dependencies of all declared JavaScript components.

- `[name-of-field-type]_field.js`: The JavaScript file that models your field.
- `[name-of-field-type].soy`: The template that defines the appearance of the field.
- `Language_xx.properties`: Define any terms that need to be translated into different languages.

In addition to the Java classes, Soy templates, and JavaScript files, Liferay DXP applications contain a `bnd.bnd` file to manage the module's metadata, and a `build.gradle` file to manage its dependencies and build properties. This example follows those patterns.

94.2 Creating Form Field Types

Liferay's Forms application does not contain a dedicated time field out-of-the-box. For ease of use and to ensure proper data is collected, you can develop a time field and learn how Liferay DXP's field types work at the same time.

There are several steps involved in creating a form field type:

1. Specify the OSGi metadata
2. Configure your build script and dependencies
3. Create a `DDMFormFieldType` component
4. Implement a `DDMFormFieldType`
5. Render the field type

Note: To avoid manually creating your own project, use BladeCLI. If you have Blade CLI on your machine, there's a template for creating form fields you can leverage using the following command syntax:

```
blade create -t form-field [ADDITIONAL OPTIONS] [PROJECT NAME]
```

See the BladeCLI documentation for more information, such as the answer to yuor question, What are those additional options?"

Using Blade CLI, you get a project skeleton with much of the boilerplate filled in, and you can focus on coding without delay.

Start by setting up the project's metadata.

Specifying OSGi Metadata

First specify the necessary OSGi metadata in a `bnd.bnd` file (see here for more information). Here's what it would look like for a module in a folder called `dynamic-data-mapping-type-time`:

```
Bundle-Name: Liferay Dynamic Data Mapping Type Time
Bundle-SymbolicName: com.liferay.dynamic.data.mapping.type.time
Bundle-Version: 1.0.0
Liferay-JS-Config: /META-INF/resources/config.js
Web-ContextPath: /dynamic-data-mapping-type-time
```

First name the bundle with a reader-friendly `Bundle-Name` and a unique `Bundle-SymbolicName` (it's common to use the root package of your module's Java classes), then set the version. Point to the JavaScript configuration file (`config.js`) that defines JavaScript modules added by your module (you'll get to that later), and set the Web Context Path so your module's resources are made available upon module activation.

Next add your dependencies to a `build.gradle` file.

Specifying Dependencies

If you're using Gradle to manage your dependencies (as Liferay DXP modules do), add this to your `build.gradle` file:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins", version: "3.0.23"
    }

    repositories {
        mavenLocal()

        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.plugin"

dependencies {
    compile group: "com.liferay", name: "com.liferay.dynamic.data.mapping.api", version: "3.2.0"
    compile group: "com.liferay", name: "com.liferay.dynamic.data.mapping.form.field.type", version: "2.0.5"
    compile group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.compendium", version: "5.0.0"
}

repositories {
    mavenLocal()

    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}

classes {
    dependsOn buildSoy
}

transpileJS {
    soySrcIncludes = ""
    srcIncludes = "**/*.es.js"
}

wrapSoyAlloyTemplate {
    enabled = true
    moduleName = "liferay-ddm-form-field-time-template"
    namespace = "ddm"
}
```

Along with the regular Java `dependencies`, there's some JavaScript dependency configuration you need to include here. It's all boilerplate and can be copied directly into your module's `build.gradle` if you follow the conventions presented here.

Next craft the OSGi Component that marks your class as an implementation of `DDMFormFieldType`.

Creating a DDMFormFieldType Component

If you're creating a *Time* field type, define the Component at the top of your `*DDMFormFieldType` class like this:

```
@Component(
  immediate = true,
  property = {
    "ddm.form.field.type.display.order:Integer=8",
    "ddm.form.field.type.icon=star-o",
    "ddm.form.field.type.js.class.name=Liferay.DDM.Field.Time",
    "ddm.form.field.type.js.module=liferay-ddm-form-field-time",
    "ddm.form.field.type.label=time-field-type-label",
    "ddm.form.field.type.name=time"
  },
  service = DDMFormFieldType.class
)
```

Define the field type's properties (`property=...`) and declare that you're implementing the `DDMFormFieldType` service (`service=...`).

`DDMFormFieldType` Components can have several properties:

`ddm.form.field.type.display.order` Integer that defines the field type's position in the *Choose a Field Type* dialog of the form builder.

`ddm.form.field.type.icon` The icon to be used for the field type. Choosing one of the Lexicon icons makes your form field blends in with the existing form field types.

`ddm.form.field.type.js.class.name` The field type's JavaScript class name–the JavaScript file is used to define the field type's behavior.

`ddm.form.field.type.js.module` The name of the JavaScript module–provided to the Form engine so the module can be loaded when needed.

`ddm.form.field.type.label` The field type's label. Its localized value appears in the *Choose a Field Type* dialog.

`ddm.form.field.type.name` The field type's name must be unique. Each Component in a field type module references the field type name, and it's used by OSGi service trackers to filter the field's capabilities (for example, rendering and validation).

Next code the *`DDMFormFieldType` class.

Implementing DDMFormFieldType

Implementing the field type in Java is made easier because of `BaseDDMFormFieldType`, an abstract class you can leverage in your code.

After extending `BaseDDMFormFieldType`, override the `getName` method by specifying the name of your new field type:

```
public class TimeDDMFormFieldType extends BaseDDMFormFieldType {
    @Override
    public String getName() {
        return "time";
    }
}
```

That's all there is to defining the field type. Next determine how your field type is rendered.

Rendering Field Types

Before you get to the frontend coding necessary to render your field type, there's another Component to define and a Java class to code.

The Component only has one property, `ddm.form.field.type.name`, and then you declare that you're adding a `DDMFormFieldRenderer` implementation to the OSGi framework:

```
@Component(
    immediate = true,
    property = "ddm.form.field.type.name=time",
    service = DDMFormFieldRenderer.class
)
```

There's another abstract class to leverage, this time `BaseDDMFormFieldRenderer`. It gives you a default implementation of the `render` method, the only required method for implementing the API. The form engine calls the render method for every form field type present in a form, and returns the plain HMTL of the rendered field type. The abstract implementation also includes some utility methods. Here's what the time field's `DDMFormFieldRenderer` looks like:

```
public class TimeDDMFormFieldRenderer extends BaseDDMFormFieldRenderer {

    @Override
    public String getTemplateLanguage() {
        return TemplateConstants.LANG_TYPE_SOY;
    }

    @Override
    public String getTemplateNamespace() {
        return "ddm.time";
    }

    @Override
    public TemplateResource getTemplateResource() {
        return _templateResource;
    }

    @Activate
    protected void activate(Map<String, Object> properties) {
        _templateResource = getTemplateResource("/META-INF/resources/time.soy");
    }

    private TemplateResource _templateResource;

}
```

Here you're setting the templating language (Soy closure templates), the template namespace (`ddm.time`), and pointing to the location of the templates within your module (`/META-INF/resource/time.soy`).

Note: Closure templates are a templating system for building UI elements. Liferay DXP developers chose to build the Forms UI with closure templates because they enable a smooth, responsive repainting of the UI as a user enters data. With closure templates there's no need to reload the entire page from the server side when the UI is updated by the user: only the relevant portion of the page is updated from the server. This makes for a smooth user experience.

Now it's time to write the template you referenced in the renderer: `time.soy` in the case of the time field type.

Create

```
src/main/resources/META-INF/resources/time.soy
```

and populate it with these contents:

```
{namespace ddm}

/**
 * Prints the DDM form time field.
 *
 * @param label
 * @param name
 * @param readOnly
 * @param required
 * @param showLabel
 * @param tip
 * @param value
 */
{template .time autoescape="deprecated-contextual"}
    <div class="form-group liferay-ddm-form-field-time" data-fieldname="{$name}">
        {if $showLabel}
            <label class="control-label">
                {$label}

                {if $required}
                    <span class="icon-asterisk text-warning"></span>
                {/if}
            </label>

            {if $tip}
                <p class="liferay-ddm-form-field-tip">{$tip}</p>
            {/if}
        {/if}

        <input class="field form-control" id="{$name}" name="{$name}" {if $readOnly}readonly{/if} type="text" value="{$value}">
    </div>
{/template}
```

There are three important things to do in the template:

1. Define the template namespace. The template namespace allows you to define multiple templates for your field type by adding the namespace as a prefix.

   ```
   {namespace ddm}
   ```

2. Describe the template parameters. The template above uses some of the parameters as flags to display or hide some parts of the HTML (for example, the `$required` parameter). If you extend `BaseDDMFormFieldRenderer`, all the listed parameters are passed by default.

   ```
   /**
    * Prints the DDM form time field.
    *
    * @param label
    * @param name
    * @param readOnly
    * @param required
    * @param showLabel
    * @param tip
    * @param value
    */
   ```

3. Write the template logic (everything encapsulated by the `{template}...{/template}` block). In the above example the template does these things:

   - Checks whether to show the label of the field, and if so, adds it.
   - Checks if the field is required, and adds `icon-asterisk` if it is.
   - Checks if a tip is provided, and displays it.
   - Provides the markup for the time field in the <input> tag. In this case a text input field is defined.

Once you have your template defined, write the JavaScript file modeling your field. Call it `time_field.js` and give it these contents:

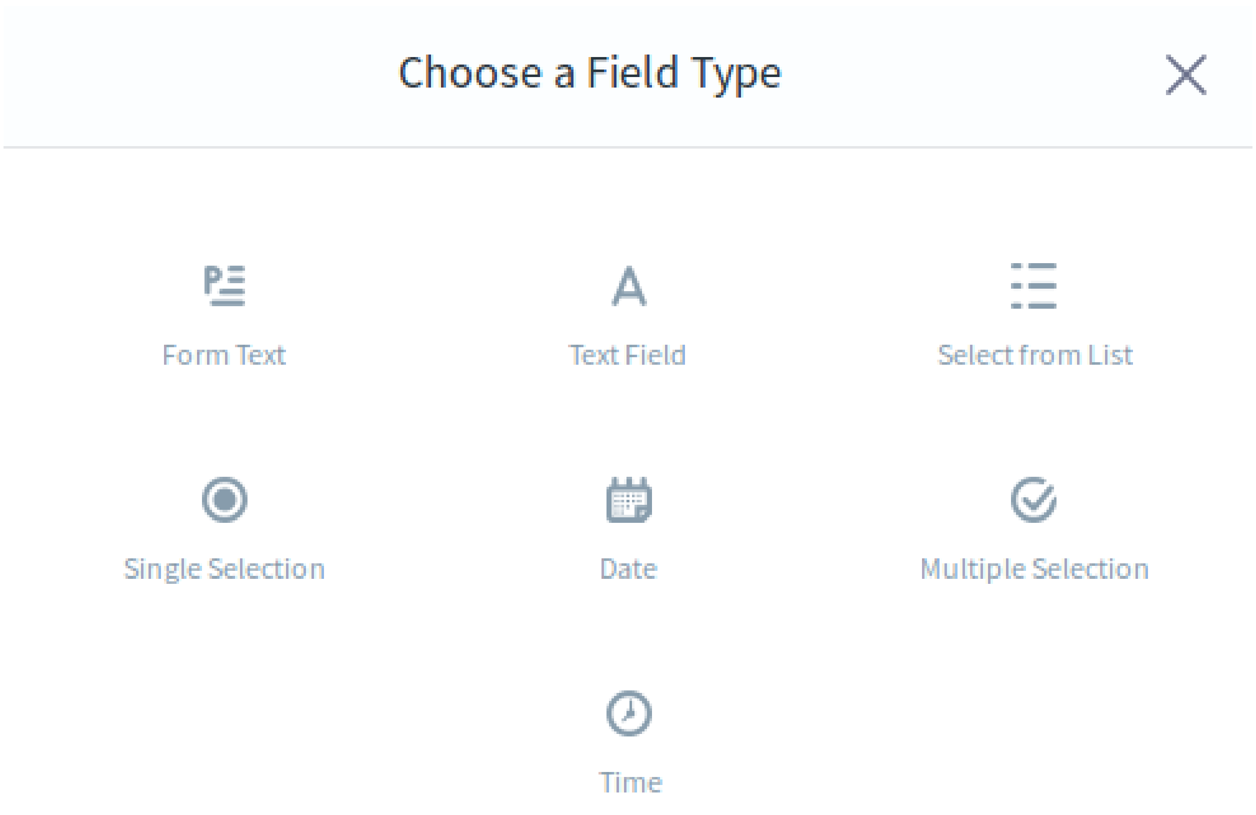
```
AUI.add('liferay-ddm-form-field-time', function(A) {
    var TimeField = A.Component.create({
        ATTRS : {
            type : {
                value : 'time'
            }
        },

        EXTENDS : Liferay.DDM.Renderer.Field,
        NAME : 'liferay-ddm-form-field-time',

        prototype : {}
    });

    Liferay.namespace('DDM.Field').Time = TimeField;
}, '', {
    requires : [ 'liferay-ddm-form-renderer-field' ]
});
```

The JavaScript above creates a component called `TimeField`. The component extends `Liferay.DDM.Renderer.Field`, which gives you automatic injection of the default field parameters.

All that's left to do is create the `config.js` file:

```
;
(function() {
    AUI().applyConfig({
        groups : {
            'field-time' : {
                base : MODULE_PATH + '/',
                combine : Liferay.AUI.getCombine(),
                modules : {
                    'liferay-ddm-form-field-time' : {
                        condition : {
                            trigger : 'liferay-ddm-form-renderer'
                        },
                        path : 'time_field.js',
                        requires : [ 'liferay-ddm-form-renderer-field' ]
                    },
                    'liferay-ddm-form-field-time-template' : {
                        condition : {
                            trigger : 'liferay-ddm-form-renderer'
                        },
                        path : 'time.soy.js',
                        requires : [ 'soyutils' ]
                    }
                },
                root : MODULE_PATH + '/'
            }
        }
    });
})();
```

This file is entirely boilerplate, and you'll never need anything different if you follow the conventions described above. In fact, if you use Blade CLI to generate a field type module, you won't need to modify anything in this file. So what is the `config.js` file for? It's a JavaScript file that defines the dependencies of the declared JavaScript components (`requires...`), and where the files are located (`path...`). The `config.js` is used by the Alloy loader when it satisfies dependencies for each JavaScript component. For more information about the Alloy loader see the tutorial on its usage.



[

If you build and deploy your new field type module, you'll see that you get exactly what you described in the `time.soy` file: a single text input field. Of course, that's not what you want! You need a time picker.

Adding Behavior to the Field

If you want to do more than simply provide a text input field, define the behavior in the `time_field.js` file. To add an AlloyUI timepicker, first specify that your component requires the `aui-timepicker` in the `requires...` block:

```
{
    requires: ['aui-timepicker','liferay-ddm-form-renderer-field']
}
```

Since you're now changing the default rendering of the field, overwrite the base `render` logic and instantiate the time picker. This occurs in the `prototype` block:

```
prototype: {
    render: function() {

        var instance = this;

        TimeField.superclass.render.apply(instance, arguments);

        instance._timePicker = new A.TimePicker(
            {
                trigger: instance.getInputSelector(),
                popover: {
                    zIndex: 1
            }
        }
    );
}
```

Invoke the original render method–it prints markup required by the Alloy time picker. Then instantiate the time picker, passing the field type input as a `trigger`. See the Alloy documentation for more information.

Now when the field is rendered, there's a real time picker.

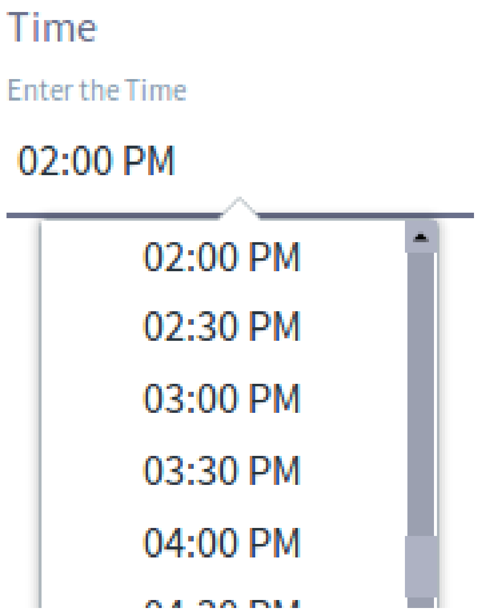


Figure 94.2: The Alloy UI Timepicker in action.

Now you know how to create a new field type and define its behavior. Currently, the field type only contains the default settings it inherits from its superclasses. If that's not sufficient, create additional settings for your field type. See the next tutorial (not yet written) to learn how.

94.3 Adding Settings to Form Field Types

Once you develop a Form Field Type, you might need to add settings to it. For example, your time field might need to be configured to accept different time formats. Here you'll learn how to add settings to form field types by adding a *mask* and a *placeholder* to the Time field type created in the previous tutorial.

Note: To learn more about using masks with the AUI Timepicker, go here. The mask just sets the format the timepicker uses to display the time choices. Use the strftime format to pick the mask you want.

To add settings to form field types, you'll use these steps:

- Write an interface that extends the default field type configuration, `DefaultDDMFormFieldTypesettings`.
- Update the `*FormFieldRenderer` so it makes the new configuration options available to the JavaScript component and/or the Soy template for rendering.
- Update the JavaScript component (defined in `time_field.js` in our example) to configure the new settings and their default values.

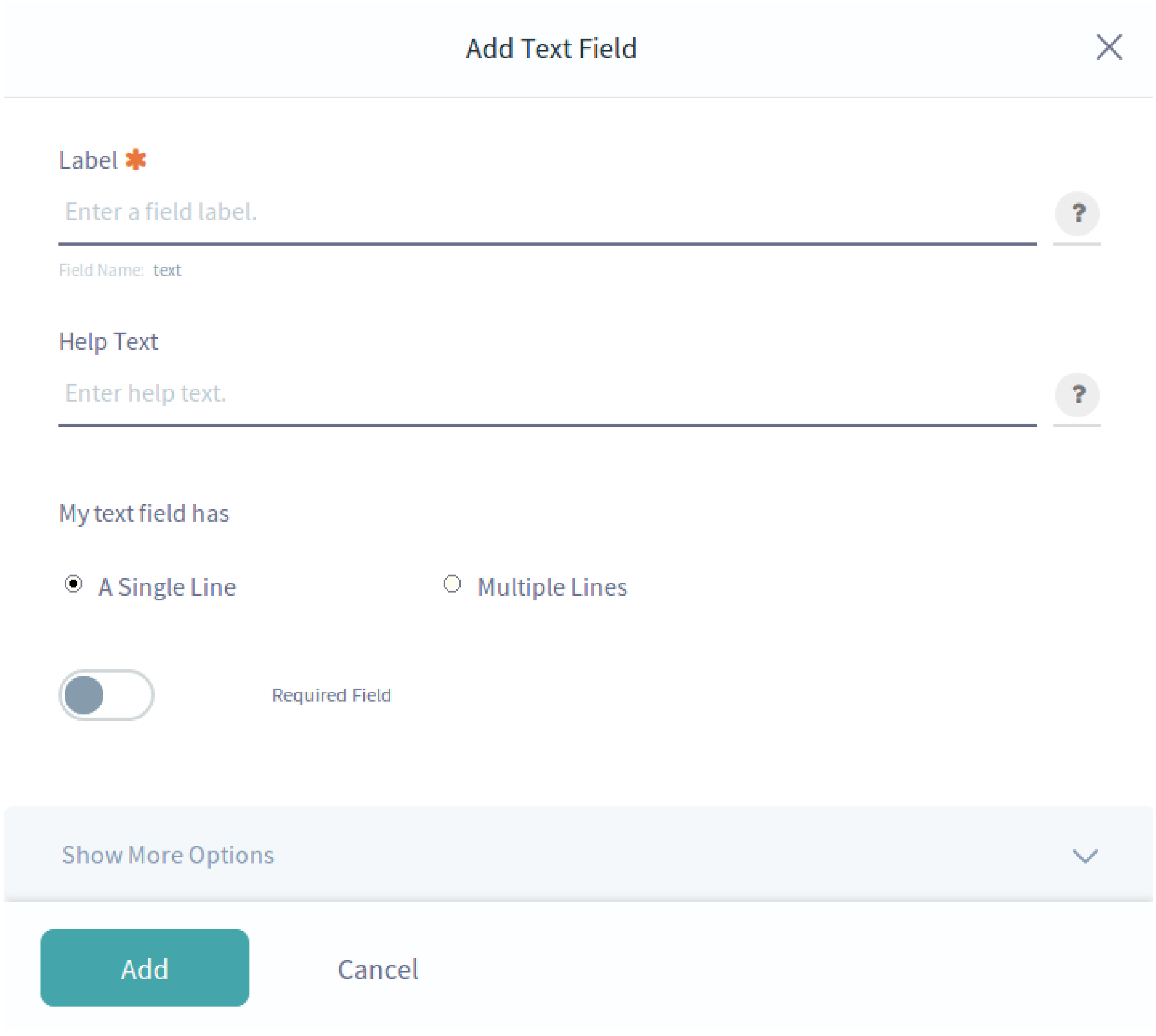
- Update the Soy template to include any settings that need to be rendered in a form (the placeholder, in our example).

Get started by crafting the interface that controls what settings your field has.

Extending the Default Type Settings

To add type settings, you need a `*TypeSettings` class that extends `DefaultDDMFormFieldTypeSettings`. Since this example works with a Time field type, call it `TimeDDMFormFieldTypeSettings`.

This class sets up the *Add [Field Type]* configuration form.


Add Text Field

Label *

Enter a field label.

Field Name: text

Help Text

Enter help text.

My text field has

[✓ A Single Line] Multiple Lines

Required Field

Figure 94.3: Like your custom field types, the text field type's settings are configured in a Java interface.

Here's what it looks like:

```
package com.liferay.docs.ddm.time;
```

```
import ...

@DDMForm
@DDMFormLayout(
    paginationMode = com.liferay.dynamic.data.mapping.model.DDMFormLayout.TABBED_MODE,
    value = {
        @DDMFormLayoutPage(
            title = "basic",
            value = {
                @DDMFormLayoutRow(
                    {
                        @DDMFormLayoutColumn(
                            size = 12,
                            value = {"label", "required", "tip", "mask", "placeholder"}
                        )
                    }
                )
            }
        ),
        @DDMFormLayoutPage(
            title = "properties",
            value = {
                @DDMFormLayoutRow(
                    {
                        @DDMFormLayoutColumn(
                            size = 12,
                            value = {
                                "predefinedValue", "visibilityExpression",
                                "fieldNamespace", "indexType", "localizable",
                                "readOnly", "dataType", "type", "name",
                                "showLabel", "repeatable"
                            }
                        )
                    }
                )
            }
        )
    }
)
public interface TimeDDMFormFieldTypeSettings
    extends DefaultDDMFormFieldTypeSettings {

    @DDMFormField(label = "%mask", predefinedValue="%I:%M %p")
    public String mask();

    @DDMFormField(label = "%placeholder")
    public String placeholder();

}
```

Would you look at that! Most of the work you need to do is in the class's annotations.

In this class you're setting up a dynamic form with all the settings your form field type needs. The form layout presented here gives your form the look and feel of a native form field type. See the note below for more information on the DDM annotations used in this form.

One thing to note is that all the default settings must be present in your settings form. Note the list of settings present for each tab (each `@DDMFormLayoutPage`) above. If you need to make one of the default settings unusable in the settings form for your field type, configure a *hide rule* for the field. Form field rules are configured using the `@DDMFormFieldRule` annotation. More information on configuring form rules will be written soon.

Your interface is extending the `DefaultDDMFormfieldTypeSettings` class. That's why the default settings are available to use in the class annotation, without setting them up in the class, as was necessary for the mask and placeholder.

DDM Annotations: The `@DDMForm` annotation on this class allows the form engine to convert the interface definition into a dynamic form. This makes it really intuitive to lay out your settings form.

For now, here are brief explanations for the annotations used in the above example:

`@DDMForm` Instantiates a new `DDMForm`. Creates a dynamic form from the annotation.

`@DDMFormLayout` Takes two variables: `paginationMode` and `value`. The pagination mode is a String that controls how the layout pages are displayed. The `pagionationMode` can be `TABBED_MODE`, `SINGLE_PAGE_MODE`, `SETTINGS_MODE`, or `WIZARD_MODE`. Under `value`, specify any `@DDMFormLayoutPages` that you want to use.

`@DDMFormLayoutPage` The sections of the type settings form. Takes two variables: `title` and `value`, where title is a String value that names the section of the form and value is one or more `@DDMFormLayoutRows`.

Note: The title of the layout pages are `basic` and `properties` for all of Liferay DXP's field types: in future versions of the Forms application, the localized value of the key you specify here will be the heading for the form section (the layout page is a section of the form). In the current version of Liferay DXP (at the time of this writing, DE DXP SP1 and CE 7.0 GA3), these are not displayed. To remain consistent with the Forms application's default fields, it's best to follow the standard approach and use `basic` and `properties`.

`@DDMFormLayoutRow` Use this to lay out the number of columns you want in the row. Most settings forms have just one row and one column.

`@DDMFormLayoutColumn` Use this to lay out the columns your settings form needs. Most settings forms have one row and one column. Each column accepts two argument, `size` and `value`.

`@DDMFormField` Use this annotation to add new fields to the settings form. In this example, the `mask` and `placeholder` settings are configured with this annotation.

Once your `*TypeSettings` class is finished, move on to update the `*Renderer` class for your form field type.

Updating the Renderer Class

To send the new configuration settings to the Soy template so they can be displayed to the end user, you need to modify the `*DDMFormFieldRenderer`.

Add this method to `TimeDDMFormFieldRenderer`:

```
@Override
protected void populateOptionalContext(
    Template template, DDMFormField ddmFormField,
    DDMFormFieldRenderingContext ddmFormFieldRenderingContext) {

    template.put(
        "placeholder", (String)ddmFormField.getProperty("placeholder"));

    template.put(
        "mask", (String)ddmFormField.getProperty("mask"));

    }
```

The `populateOptionalContext` method takes three parameters: The template object, the `DDMFormField`, and the `DDMFormFieldRenderingContext`. The `DDMFormField` represents the definition of the field type instance: you can use this object to access the configurations set for the field type (the mask and placeholder settings in our case). The `DDMFormFieldRenderingContext` object contains extra information about the form such as the user's locale, the HTTP request and response objects, the portlet namespace, and more (all of its included properties can be found here.

You're putting the new settings into the template object, which is just an extension of a Map that takes a String and an Object (in this case the Object is the property configured in the `@DDMFormField` in your `*TypeSettings` class, retrieved by the name of the field: `placeholder` and `mask`, respectively.

Now the JavaScript component and the Soy template can access the new settings. Next, update the JavaScript Component so it handles these properties and can use them, whether passing them to the template context (similar to the `*Renderer`, only this time for client-side rendering), or using them to configure the behavior of the JavaScript component itself.

Note: Remember that the Soy template can be used for server side or client side rendering. By defining the settings you're adding in both the Java Renderer and the JavaScript Renderer, you're allowing for the best possible user experience. For example, if a form builder is in the form builder, configuring a form field type, the configuration they enter can be directly passed to the template, and become visible in the UI, almost instantly. However, when the user clicks into a form field initially to begin editing, the rendering occurs from the server side.

Next configure the JavaScript component to include the new settings.

Adding Settings to the JavaScript Component

The JavaScript component needs to know about the new settings. First configure them as attributes of the component:

```
ATTRS: {
    mask: {
        value: '%I:%M %p'
    },
    placeholder: {
        value: ''
    },
    type: {
        value: 'time'
    }
},
```

The mask setting has a default value of `%I:%M %p`, and the placeholder is blank. Now that the new settings are declared as attributes of the component, make the JavaScript component pass the placeholder configuration to the Soy template on the client side. Just like in the Java renderer, pass the placeholder configuration to the template context. In this case, override the `getTemplateContext()` method to pass in the placeholder configuration. Add this to the `prototype` section of the JavaScript component definition:

```
getTemplateContext: function() {
    var instance = this;

return A.merge(
    TimeField.superclass.getTemplateContext.apply(instance, arguments),
        {
        placeholder: instance.get('placeholder')
        }
    );
},
```

Then in the component's render method, add the mask as an attribute of the AUI Timepicker using `mask: instance.get('mask')`.

```
        render: function() {
         var instance = this;

         TimeField.superclass.render.apply(instance, arguments);

         instance._timePicker = new A.TimePicker(
             {
                 trigger: instance.getInputSelector(),
                 mask: instance.get('mask'),
                 popover: {
                     zIndex: 1
                 }
             }
         );
    }
```

Now the field type JavaScript component is configured to include the settings. All you have left to do is to update the Soy template so the placeholder can be rendered in the form with the time field.

Updating the Soy Template

After all that, adding the placeholder setting to your Soy template's logic is simple.

The whole template is included below, but the only additions are in the commented section (adds the placeholder to the list of parameters–the ? indicates that the placeholder is not required), and then in the <input> tag, where you use the parameter value to configure the placeholder HTML property with the proper value.

```
{namespace ddm}

/**
 * Prints the DDM form time field.
 *
 * @param label
 * @param name
 * @param? placeholder
 * @param readOnly
 * @param required
 * @param showLabel
 * @param tip
 * @param value
 */
{template .time autoescape="deprecated-contextual"}
    <div class="form-group liferay-ddm-form-field-time" data-fieldname="{$name}">
        {if $showLabel}
            <label class="control-label">
                {$label}

                {if $required}
                    <span class="icon-asterisk text-warning"></span>
                {/if}
            </label>

            {if $tip}
                <p class="liferay-ddm-form-field-tip">{$tip}</p>
            {/if}
        {/if}

        <input class="field form-control" id="{$name}" name="{$name}" placeholder="{$placeholder}" {if $readOnly}readonly{/if} type="text" value="{$value}">
    </div>
{/template}
```

Why isn't the mask parameter added to the Soy template? The mask is not needed in the template because it's only used in the JavaScript for configuring the behavior of the timepicker. You don't need the dynamic rendering of the soy template to take the mask setting and configure it in the form. The maskv set by the form builder is captured in the rendering of the timepicker itself.

Now when you build the project and deploy your time field, you have a fully developed *time* form field type, complete with the proper JavaScript behavior and with additional settings.

Chapter 95

Search

Liferay stores its information in a database. If you need to search for data, why not search the database directly? Why add the complexity of a search engine? Database table merges are expensive! Documents in a search index often contain searchable fields from multiple tables in the database.

Searching with a search engine provides access to features such as relevance and scoring. Database searches do not support features like fuzzy searching or any type of relevancy. Moreover, when searching with a search engine, you can apply algorithms such as "More Like This" to obtain similar content. Search engines also support geolocation, faceting of search results, and multi-lingual searching.

95.1 Basic Search Concepts

Indexing: During indexing, a document is sent to the search engine. This document contains a collection of fields of various types (string, etc). The search engine processes each field within the document. For each field, the search engine determines whether it needs to simply store the field or if it needs to undertake special analysis (index time analysis). Index time analysis can be configured for each field (see Mapping Definitions).

For fields requiring analysis, the search engine first tokenizes the value to obtain individual words or tokens. Following tokenization, the search engine passes each token through a series of analyzers. Analyzers perform different functions. Some remove common words or stop words (e.g., "the", "and", "or") while others perform operations like lowercasing all characters.

Searching: Searching involves sending a search query and obtaining results (a.k.a. hits) from the search engine. The search query may be comprised of both queries and filters (more on this later). Each query or filter specifies a field to search within and the value to match against. Upon receiving the search query, the search engine iterates through each field within the nested queries and filters. During this process, the engine may perform special analysis prior to executing the query (search time analysis). Search time analysis can be configured for each field (see Mapping Definitions).

Mapping Definitions

Most search engines can be semi-intelligent in automatically deciphering how to process documents passed to them. However, there are many instances where it's desirable to explicitly configure how a field should be processed.

Mappings allow users to control how a search engine processes a given field. For instance, for all field names that end in “es_ES”, we want to process the field values as Spanish, removing any common Spanish words like “si”.

In Elasticsearch and Solr, the two supported search engines for Liferay Portal, we define mappings using `liferay-type-mappings.json` and `schema.xml`, respectively.

The Elasticsearch mapping JSON file can be seen here: https://github.com/liferay/liferay-portal/blob/7.0.6-ga7/modules/apps/foundation/portal-search/portal-search-elasticsearch/src/main/resources/META-INF/mappings/liferay-type-mappings.json

The Solr `schema.xml` can be seen here: https://github.com/liferay/liferay-portal/blob/7.0.6-ga7/modules/apps/portal-search-solr/portal-search-solr/src/main/resources/META-INF/resources/schema.xml

These are default mapping files that are shipped with the product. You can further customize these mappings to fit your needs. For example, you might want to use a special analyzer for a custom inventory number field.

Liferay Search Infrastructure

Search engines already provide native APIs. Why does Liferay provide search infrastructure to wrap search engines? Liferay’s search infrastructure ensures that documents are indexed with fields Liferay needs:

`entryClassName`, `entryClassPK`, `assetTagNames`, `assetCategories`, `companyId`, `groupId`, staging status, etc.

Liferay’s search infrastructure ensures that the proper set of filters are added to search queries to scope results. Liferay’s search infrastructure also provides capabilities like permission checking and creating hit summaries for display.

95.2 Liferay Search API

Liferay Portal’s Search API allows users to build a search query, execute it, and obtain search hits that match the query.

Queries and Filters

Elasticsearch and Solr do not make API level distinctions between queries and filters. However, Liferay’s API explicitly provides two sets of APIs, one for queries and one for filters.

A *filter* asks a yes or no question for every document. A *query* asks the same yes or no question AND how well (score) a document matches the specified criteria. For instance, a filter might ask is the status field equal to staging or live. A query might ask if the document’s content field field contains the words “Liferay”, “Content”, “Management”, and how relevant the content of the document is to the search terms.

With respect to performance, filters are much faster since the documents that match a filter can be easily cached. Queries not only match documents but also calculate scores. Liferay uses filters and queries together so that filters can reduce the number of matched documents before the query examines them for scoring.

Liferay’s Search API supports the following types of queries:

Full text queries:

- MatchQuery: Full text matching, scored by relevance.
- MultiMatchQuery: MatchQuery over several fields.
- StringQuery: Uses Lucene query syntax

Term queries:

- TermQuery: Exact matching on keyword fields and indexed terms

- TermRangeQuery: TermQuery with a range
- WildcardQuery: Wildcard (* and ?) matching on keyword fields and indexed terms
- FuzzyQuery: Scrambles characters in input before matching

Compound queries:

- BooleanQuery: Allows a combo of several query types. Individual queries are added as clauses with SHOULD | MUST | MUST_NOT.
- DisMaxQuery

Other queries:

- MoreLikeThisQuery
- MatchAllQuery: Matches all documents

Liferay's Search API supports the following types of filters:
Term filters:

- TermFilter
- TermsFilter
- PrefixFilter
- ExistsFilter
- MissingFilter
- RangeTermFilter

Compound filters:

- BooleanFilter

Geo filters: (Geolocation filters help filter documents based on the latitude and longitude fields)

- GeoDistanceFilter
- GeoDistanceRangeFilter
- GeoBoundingBoxFilter
- GeoPolygonFilter

Other filters:

- QueryFilter: Turns any query into a filter. E.g., can a BooleanQuery into a BooleanFilter
- MatchAllFilter: Matches all documents

Aggregations

Aggregations help summarize search results. Individual aggregations can be used to create more complex aggregations. Facets are a type of aggregation. In addition to facets, Liferay also provides group by and statistics aggregations.
Facets:

- Date Range Facet
- Modified Date Facet

- MultiValue Facet
- Range Facet
- Scope Facet
- Simple Facet

Statistics:
Stats provides general statistics for a desired field within the returned search results:

- count
- max
- mean
- min
- missing
- standard deviation
- sum
- sum of squares

GroupBy:
GroupBy is a powerful feature that allows you to group search results based on a particular field. For example, suppose you wish to group the search results based on the asset type (e.g., web content article, document, blog post, etc.). To do so, you would create a search query that contains a GroupBy aggregation with the field "entryClassName".
Other attributes you can specify:

- The maximum number of results in each group
- Special sorting for the grouped results

Indexers

There is an Indexer for each asset in the portal (e.g., DLFileEntryIndexer). This allows each asset to control what fields are indexed and what filters are applied to the search query.
Generally, when you create an asset that requires indexing, you would implement a new Indexer by extending `com.liferay.portal.kernel.search.BaseIndexer<T>`.
For more information, consult the Javadocs for `com.liferay.portal.kernel.search.Indexer<T>` and `com.liferay.portal.kernel.search.BaseIndexer<T>`: @platform-ref@/7.0-latest/javadocs

IndexerPostProcessor

The IndexerPostProcessor allows developers to customize

- Search queries before they are sent to the search engine
- Documents before they are sent to the search engine
- Summaries for results before they are returned to the end users

This is the preferred way to customize existing Indexers.
Follow these steps to add a new IndexerPostProcessor:

1. Implement the interface `com.liferay.portal.kernel.search.IndexerPostProcessor`.
2. Publish it to the OSGi registry with the property `indexer.class.name`

`postProcessContextQueryBooleanFilter` allows the developer to customize the filters created by the `Indexer.getFacetBooleanFilter`. These filters are generally applied to the fields:

- entryClassName
- relatedClassName
- relatedEntryClassNames
- permissions related fields (e.g., roleId, groupId, etc.).

`postProcessFullQuery` allows the developer to customize the overall search query which includes

- Filters for any default facets, including those for

    - asset category ids
    - asset tag names
    - entry class names
    - folderIds
    - groupIds
    - layoutUUIDs
    - userId

- The keyword search queries. By default, this includes searches for the fields

    - description
    - title
    - userName
    - keyword
    - searchable Expando fields
    - localized fields for assetCategoryTitles

HitsProcessor

`com.liferay.portal.kernel.search.HitsProcessor` allows developers to preprocess the results from the search engine before they are returned to the user. This allows for features like

- spell checking
- suggesting related queries
- indexing search queries that have returned high quality search results

HitsProcessors are stored in a HitsProcessorRegistry and sorted by their `sort.order`. Essentially, we have a chain of responsibility held by the HitsProcessorRegistry.

By default, the HitsProcessor order is:

1. `CollatedSpellCheckHitsProcessor`

    - Performs a spell check if the minimum score for search results is less than a given threshold
    - Number of results defined in portal.properties (index.search.collated.spell.check.result.scores.threshold)

2. `AlternateKeywordQueryHitsProcessor`

    - Automatically issue a query using the suggested keywords from the `CollatedSpellCheckHitsProcessor`.

3. `QueryIndexingHitsProcessor`

    - If query indexing is enabled (`index.search.query.indexing.enabled` in `portal.properties`), then index the search query if the number of hits has exceeded a configured quantity (`index.search.query.indexing.threshold` in `portal.properties`).

4. `QuerySuggestionHitsProcessor`

    - If number of results returned has not met a given threshold (`index.search.query.suggestion.scores.threshold` in `portal.properties`), then suggest other potential queries that previous searches have yielded more results (`index.search.query.suggest.max` in `portal.properties`).

Suggestions

Suggestions are a powerful feature where the search engine can suggest "similar" results for a given query. For instance, suppose you have a blog entry with the title "Liferay Portal Content Management" and you would like to find other content with similar titles.

`com.liferay.portal.kernel.search.IndexSearcher` provides methods to access suggestion capabilities. It implements `com.liferay.portal.kernel.search.suggest.QuerySuggester`.

The QuerySuggester provides facilities for

- Spell Checking
- Related search queries
- General Suggester requests

Spell Checking

For Elasticsearch, spell checking heavily relies on the suggester API: - Dictionary words are analyzed by their language specific analyzer and indexed. - `TermSuggester` is used to provide suggestions for words based on specific `StringDistance` algorithms.

Solr's implementation of `Suggester` is less flexible and sophisticated. Solr's spell checking algorithm is based strictly on NGrams and does not handle Asian languages very well.

Note that using the search engine's spell checking functionality doesn't guarantee returned results. Instead, spell checking seeks to ensure that the query is correct.

Similar Search Queries

Like spell checking, similar search queries has a more robust implementation in Elasticsearch. The Elasticsearch implementation uses phrase suggesters on indexed keyword search queries.

Solr's similar search queries implementation is again based on tokenized NGrams.

Other Suggesters

You can also send custom Suggester requests and get SuggesterResults back from the search engine by calling `QuerySuggester.suggest(SearchContext, Suggester)`.

95.3 Search Adapter API

Search adapters convert Liferay Portal's API to the underlying search engine's API. This pluggable architecture allows customers to more easily integrate with other search engines. Liferay ships with two adapters: an Elasticsearch adapter and a Solr adapter.

The search adapter API has 2 primary interfaces:

- `IndexSearcher`: invoked for all search operations
- `IndexWriter`: used when adding, updating, or deleting documents from the search engine.

95.4 Transactional Search

Search engines do not operate within a traditional JTA/JTS transaction. In place of "real" transactions, Liferay buffers indexing operations (delete, update) until either the surrounding transaction has been committed or we have exceeded the max buffer size. The buffered indexer requests are abandoned in the event of transaction rollback. This gives us some semblance of transactional control, except in scenarios where we have large batches of commits (e.g., exceeds `maxBufferSize`).

When `maxBufferSize` has been exceeded, the search infrastructure executes buffered indexer requests to free up space in the buffer.

Buffered `IndexerRequests` always execute in FIFO order. There is no collation of `IndexerRequests` in the buffer.

You can activate / deactivate and set the buffer size by configuring `com.liferay.portal.search.configuration.IndexerRegis`
By default, the buffering is activated and the max buffer size is 200.

For a list of buffered methods, see `com.liferay.portal.kernel.search.Indexer`. All methods annotated with `@Bufferable` are subject to potential buffering.

95.5 Customizing Liferay Search

There are several extension points available for users to customize. The most obvious is the ability to add a new search engine adapter.

Adding a new Search Engine Adapter

To add a new search engine adapter, developers must create the following components and publish them to Liferay's OSGi registry:

1. Implement a new IndexSearcher that should convert the Liferay Search objects to the underlying search engine's dialects:

    - QueryTranslator: Translates Liferay Queries to the native search engine's queries.
    - FilterTranslator: Translates Liferay filters into native search engine's filters.
    - GroupByTranslator: Translates the GroupBy aggregation to the search engine's group by top hits aggregation.
    - StatsTranslator: Translates Stats request to the appropriate search engine's statistics aggregation.
    - SuggesterTranslator: Translates suggestion requests to the appropriate search engine's suggester API.

2. Implement a new IndexWriter that should

    - Convert the Liferay Document to a format understood by the underlying search engine's Document Format.
    - Use the search engine's API to update, add, and delete documents.

3. Implement a new SearchEngineConfigurator that should extend `AbstractSearchEngineConfigurator` to perform proper wiring.

4. Implement a new SearchEngine that

    - Should extend from `BaseSearchEngine` to perform any search engine specific initialization.
    - Should be published to Liferay's OSGi registry along with the property `search.engine.id=[searchEngineId]`.

Customizing IndexerRequestBufferOverflowHandler

`IndexerRequestBufferOverflowHandler` controls how the search infrastructure handles situations where buffered indexer requests has exceeded the configured maximum buffer size.

To customize, implement an `IndexerRequestBufferOverflowHandler` and publish it to Liferay's OSGi registry.

Customizing HitsProcessors

`com.liferay.portal.kernel.hits.HitsProcessor` objects are held in a `com.liferay.portal.kernel.hits.HitsProcessorRegistry` To add a new HitsProcessor, simply implement the interface and publish to the OSGi registry with the property `sort.order`.

Chapter 96

Application Security

Nothing has received more attention on the web in recent years than security. Liferay has a robust security model for you to use in your applications, and it supports a wide variety of security features. In this section, you can learn about these features:

- Resources, Roles, and Permissions
- Custom SSO Providers
- Authentication Pipelines
- Service Access Policies
- Authentication Verifiers

96.1 Adding Permissions to Resources

Public bulletin boards are great. Anyone can inform others of just about anything. On the other hand, *anyone* can post just about *anything* on the bulletin board. Some of this content might not be relevant to the community. Other content might be inappropriate. Thus, you sometimes need a way to restrict who can post or access content.

Fortunately, no matter what your portlet does, access to it and to its content can be controlled with permissions. Read on to learn about Liferay's permissions system and how to add permissions to your application.

Liferay's Permission System

Liferay's permission system uses a flexible mechanism that defines the actions that a given user can perform within the context of Liferay or a specific application. Liferay developers break down the operations that can be performed in Liferay or in a certain application into distinct *actions*. The act of granting the ability to perform an action to a specific role is the act of granting a *permission*. In Liferay, permissions are not granted to directly to users. Instead, permissions are granted to roles. Roles, in turn, can be assigned to specific users, sites, organizations, or user groups.

Developers need to define the different types of operations that are required to suit the business logic of their applications. They don't need to worry about which users will receive which permissions. Once the actions have been determined and configured, portal administrators can grant permissions to perform those actions to users, sites, organizations, or user groups by assigning roles. Administrators can use the portal's administration tools to grant permissions to roles or they can use the permissions UIs of individual portlets.

In this tutorial, you'll learn how to use Liferay's permissions system to provide Liferay administrators the same level of control over permissions that they have over the out-of-the-box Liferay applications.

Before proceeding, make sure you understand these critical terms:

Action: An operation that can be performed by a Liferay user. For example, actions that be performed on the Bookmarks application include `ADD_TO_PAGE`, `CONFIGURATION`, and `VIEW`. Actions that can be performed with respect to Bookmarks entry entity include `ADD_ENTRY`, `DELETE`, `PERMISSIONS`, `UPDATE`, and `VIEW`.

Resource: A generic representation of any application or entity in the portal on which an action can be performed. Resources are used for permission checking. For example, resources within a Liferay instance could include the RSS application with instance ID `hF5f`, a globally scoped Wiki page, a Bookmarks entry of the site X, and a Message Boards post with the ID `5052`.

Permission: An action that can be performed on a resource. In Liferay's database, resources and actions are saved in pairs. (Each entry in the `ResourceAction` table contains both the name of a portlet or entity and the name of an action.) For example, the `VIEW` action with respect to *viewing the Bookmarks application* is associated with the `com_liferay_bookmarks_web_portlet_BookmarksPortlet` portlet ID. The `VIEW` actions with respect to *viewing a Bookmarks Folder* or *viewing a Bookmarks entry* are associated with the `com.liferay.bookmarks.model.BookmarksFolder` and `com.liferay.bookmarks.model.BookmarksEntry` entities, respectively.

There are two kinds of resources in Liferay: *portlet resources* and *model resources*. Portlet resources represent portlet applications. The names of portlet resources typically correspond to the IDs of the portlets themselves. For example, the fully qualified name of the Bookmarks portlet class is `com.liferay.bookmarks.web.portlet.BookmarksPortlet`. Its ID is defined in the `BookmarksPortletKeys` class like this:

```
public static final String BOOKMARKS =
    "com_liferay_bookmarks_web_portlet_BookmarksPortlet";
```

This `BOOKMARKS` string is used when declaring portlet resources in `default.xml` files, as discussed below.

There are two kinds of resources in Liferay: *portlet resources* and *model resources*. Portlet resources represent portlets. The names of portlet resources are the portlet IDs from the portlets' `portlet.xml` files (or in the case of core portlets, Liferay's `portlet-custom.xml`). Model resources refer to entities within Liferay. The names of model resources are the fully qualified class names of the entities they represent. In the XML displayed below, permission implementations are first defined for the *portlet* resource and then for the *model* resources.

Model resources represent entities within Liferay, such as bookmarks folders or bookmarks entries. The names of model resources are the fully qualified class names of the entities they represent. In the `default.xml` files displayed below, permission implementations are first defined for the *portlet* resource and then for the *model* resources.

Note: For each resource, there are four scopes to which the permissions can be applied: company, group, group-template, or individual. See the Javadoc of ResourcePermissionImpl for more information.

You can add permissions to your custom portlets using four easy steps (also known as *DRAC*):

1. Define all resources and their permissions.

2. Register all defined resources in the permissions system. This is also known as *adding resources*.

3. Associate the necessary permissions with resources.

4. Check permission before returning resources.

Define All Resources and Permissions

Here you'll learn the first step, in which you define your resources and their actions. The Bookmarks application is used here to demonstrate how to define portlet resources and model resources. Open the `default.xml` file in `bookmarks-web` module of the Bookmarks application: default.xml. There, you'll see the following mapping of resources to actions:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action Mapping 7.0.0//EN" "http://www.liferay.com/dtd/liferay-
resource-action-mapping_7_0_0.dtd">

<resource-action-mapping>
    <portlet-resource>
        <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksAdminPortlet</portlet-name>
        <permissions>
            <supports>
                <action-key>ACCESS_IN_CONTROL_PANEL</action-key>
                <action-key>CONFIGURATION</action-key>
                <action-key>VIEW</action-key>
            </supports>
            <site-member-defaults>
                <action-key>VIEW</action-key>
            </site-member-defaults>
            <guest-defaults>
                <action-key>VIEW</action-key>
            </guest-defaults>
            <guest-unsupported>
                <action-key>ACCESS_IN_CONTROL_PANEL</action-key>
                <action-key>CONFIGURATION</action-key>
            </guest-unsupported>
        </permissions>
    </portlet-resource>
    <portlet-resource>
        <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksPortlet</portlet-name>
        <permissions>
            <supports>
                <action-key>ADD_TO_PAGE</action-key>
                <action-key>CONFIGURATION</action-key>
                <action-key>VIEW</action-key>
            </supports>
            <site-member-defaults>
                <action-key>VIEW</action-key>
            </site-member-defaults>
            <guest-defaults>
                <action-key>VIEW</action-key>
            </guest-defaults>
            <guest-unsupported>
                <action-key>CONFIGURATION</action-key>
            </guest-unsupported>
        </permissions>
    </portlet-resource>
</resource-action-mapping>
```

This `default.xml` defines portlet resources. The bookmarks service module also contains a default.xml file:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action Mapping 7.0.0//EN" "http://www.liferay.com/dtd/liferay-
resource-action-mapping_7_0_0.dtd">

<resource-action-mapping>
    <model-resource>
        <model-name>com.liferay.bookmarks</model-name>
        <portlet-ref>
            <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksPortlet</portlet-name>
            <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksAdminPortlet</portlet-name>
        </portlet-ref>
        <root>true</root>
        <weight>1</weight>
        <permissions>
            <supports>
                <action-key>ADD_ENTRY</action-key>
                <action-key>ADD_FOLDER</action-key>
                <action-key>PERMISSIONS</action-key>
                <action-key>SUBSCRIBE</action-key>
                <action-key>VIEW</action-key>
            </supports>
            <site-member-defaults>
                <action-key>ADD_ENTRY</action-key>
                <action-key>SUBSCRIBE</action-key>
                <action-key>VIEW</action-key>
            </site-member-defaults>
            <guest-defaults>
                <action-key>VIEW</action-key>
            </guest-defaults>
            <guest-unsupported>
                <action-key>ADD_ENTRY</action-key>
                <action-key>ADD_FOLDER</action-key>
                <action-key>PERMISSIONS</action-key>
                <action-key>SUBSCRIBE</action-key>
            </guest-unsupported>
        </permissions>
    </model-resource>
    <model-resource>
        <model-name>com.liferay.bookmarks.model.BookmarksFolder</model-name>
        <portlet-ref>
            <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksPortlet</portlet-name>
            <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksAdminPortlet</portlet-name>
        </portlet-ref>
        <weight>2</weight>
        <permissions>
            <supports>
                <action-key>ACCESS</action-key>
                <action-key>ADD_ENTRY</action-key>
                <action-key>ADD_SUBFOLDER</action-key>
                <action-key>DELETE</action-key>
                <action-key>PERMISSIONS</action-key>
                <action-key>SUBSCRIBE</action-key>
                <action-key>UPDATE</action-key>
                <action-key>VIEW</action-key>
            </supports>
            <site-member-defaults>
                <action-key>ADD_ENTRY</action-key>
                <action-key>SUBSCRIBE</action-key>
                <action-key>VIEW</action-key>
            </site-member-defaults>
            <guest-defaults>
                <action-key>VIEW</action-key>
            </guest-defaults>
            <guest-unsupported>
                <action-key>ADD_ENTRY</action-key>
                <action-key>ADD_SUBFOLDER</action-key>
                <action-key>UPDATE</action-key>
            </guest-unsupported>
        </permissions>
    </model-resource>
    <model-resource>
        <model-name>com.liferay.bookmarks.model.BookmarksEntry</model-name>
        <portlet-ref>
            <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksPortlet</portlet-name>
            <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksAdminPortlet</portlet-name>
        </portlet-ref>
        <weight>3</weight>
        <permissions>
            <supports>
                <action-key>DELETE</action-key>
                <action-key>PERMISSIONS</action-key>
                <action-key>SUBSCRIBE</action-key>
                <action-key>UPDATE</action-key>
                <action-key>VIEW</action-key>
            </supports>
            <site-member-defaults>
                <action-key>SUBSCRIBE</action-key>
                <action-key>VIEW</action-key>
            </site-member-defaults>
            <guest-defaults>
                <action-key>VIEW</action-key>
            </guest-defaults>
            <guest-unsupported>
                <action-key>SUBSCRIBE</action-key>
                <action-key>UPDATE</action-key>
            </guest-unsupported>
        </permissions>
    </model-resource>
</resource-action-mapping>
```

This `default.xml` file defines model resources.

The `<portlet-resource>` tag is used to define actions that can be taken with respect to the portlet window. For the Bookmarks application, such actions include these:

- `ADD_TO_PAGE`: *Add* the application to a page
- `CONFIGURATION`: *Access* the application's Configuration window
- `VIEW`: *View* the application

All the supported actions are defined in the `<supports>` tag, a sub-tag of the `<permissions>` tag (which is itself a sub-tag of the `<portlet-resource>` tag:

```
<supports>
    <action-key>ADD_TO_PAGE</action-key>
    <action-key>CONFIGURATION</action-key>
    <action-key>VIEW</action-key>
</supports>
```

The default permissions for site members are defined in the `<site-member-defaults>` tag. In the case of the Bookmarks application, site members can view any Bookmarks application in the site:

```
<site-member-defaults>
    <action-key>VIEW</action-key>
</site-member-defaults>
```

Similarly, the default permissions for guests are defined in the `<guest-defaults>` tag. Guests can also view any Bookmarks application in the site:

```
<guest-defaults>
    <action-key>VIEW</action-key>
</guest-defaults>
```

The `<guest-unsupported>` tag specifies permissions forbidden to guests. By default, guests cannot access the Bookmarks application in the Control Panel nor can they access the Bookmarks's configuration window:

```
<guest-unsupported>
    <action-key>ACCESS_IN_CONTROL_PANEL</action-key>
    <action-key>CONFIGURATION</action-key>
</guest-unsupported>
```

With respect to model resources, guests are forbidden from adding bookmarks entries and bookmarks folders. Guests are also not permitted to edit permissions or subscribe. Remember, these are just default permissions that can be changed by administrators. See the following entry in the first `<model-resource>` tag:

```
<guest-unsupported>
    <action-key>ADD_ENTRY</action-key>
    <action-key>ADD_FOLDER</action-key>
    <action-key>PERMISSIONS</action-key>
    <action-key>SUBSCRIBE</action-key>
</guest-unsupported>
```

The `<model-resource>` tag is used to define actions that can be performed with respect to models, also known as entities. There are two kinds of actions in Liferay: *top-level actions* and *resource actions*. Top-level actions are not applied to a particular resource. For example, the action of adding a new entity is not applied to a particular resource, so it's considered a top-level action. The first `<model-resource>` tag defines adding bookmark entry and bookmark folder resources as top-level actions:

```
<supports>
    <action-key>ADD_ENTRY</action-key>
    <action-key>ADD_FOLDER</action-key>
</supports>
```

The second and third `<model-resource>` tags define resource actions that can be applied to the `BookmarksFolder` and `BookmarksEntry` entities, respectively. For example, the permissions for the following actions are defined with respect to the resource associated with the `BookmarksEntry` entity:

```
<supports>
    <action-key>DELETE</action-key>
    <action-key>PERMISSIONS</action-key>
    <action-key>SUBSCRIBE</action-key>
    <action-key>UPDATE</action-key>
    <action-key>VIEW</action-key>
</supports>
```

Similarly, the permissions for the following actions are defined with respect to the resource associated with the `BookmarksFolder` entity:

```
<supports>
    <action-key>ACCESS</action-key>
    <action-key>ADD_ENTRY</action-key>
    <action-key>ADD_SUBFOLDER</action-key>
    <action-key>DELETE</action-key>
    <action-key>PERMISSIONS</action-key>
    <action-key>SUBSCRIBE</action-key>
    <action-key>UPDATE</action-key>
    <action-key>VIEW</action-key>
</supports>
```

In each `<model-resource>` tag, notice that the model name must be defined. The `<model-name>` must be either the fully-qualified name of a package or of an entity class. For example, `com.liferay.bookmarks` is the name of a package and `com.liferay.bookmarks.model.BookmarksEntry` is the name of an entity class. Using a package is the recommended convention for permissions that refer to top-level actions:

```
<model-name>com.liferay.bookmarks</model-name>
```

The `ADD_ENTRY` and `ADD_FOLDER` permissions are defined this way since they're top-level actions. For resource actions, the entity class is specified:

```
<model-name>com.liferay.bookmarks.model.BookmarksEntry</model-name>
```

The <portlet-ref> element comes next and contains a <portlet-name> sub-tag.

```
<portlet-ref>
    <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksPortlet</portlet-name>
    <portlet-name>com_liferay_bookmarks_web_portlet_BookmarksAdminPortlet</portlet-name>
</portlet-ref>
```

The value of <portlet-name> references the name of the portlet to which the model resource belongs. It's possible for a model resource to belong to multiple portlets referenced with multiple <portlet-name> elements. This is the case here since both the Bookmarks application and the Bookmarks Admin application can be used to perform actions on bookmark entries.

The <supports>, <site-member-defaults>, <guest-defaults>, and <guest-unsupported> tags work the same way in the <model-resource> tag as they do in the <portlet-resource> tag. The <supports> tag lets you specify a list of supported actions that require permission to perform. The <site-member-defaults> tag and the <guest-defaults> tags define default permissions for site members and guests, respectively. And the <guest-unsupported> tag specifies permissions forbidden to guests.

After defining resource permissions for your portlet, you need to point Liferay to the default.xml file that contains your definitions. In Liferay's core, there are multiple permissions XML definition files for various core Liferay portlets in the portal/portal-impl/src/resource-actions directory. The default.xml file in that folder contains pointers to the definition files of the various applications. This excerpt from Liferay's default.xml references the resource permission definition files for all built-in Liferay portlets:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action Mapping 7.0.0//EN" "http://www.liferay.com/dtd/liferay-
resource-action-mapping_7_0_0.dtd">

<resource-action-mapping>
    <resource file="resource-actions/portal.xml" />
    <resource file="resource-actions/announcements.xml" />
    <resource file="resource-actions/asset.xml" />
    <resource file="resource-actions/blogs.xml" />
    ...
</resource-action-mapping>
```

Your application's permissions XML file should be named default.xml and should be placed in a directory in your module's classpath. src/resource-actions is the standard location. Once your project's default.xml file has been created, you should create a properties file named portlet.properties that contains a reference to your permissions XML file. In your portlet.properties file, create a property named resource.actions.configs with the relative path to your portlet's resource-action mapping file (e.g. default.xml) as its value. Here's what this property specification might look like:

```
resource.actions.configs=resource-actions/default.xml
```

Your permissions XML file must contain a root resource-action-mapping element. Check out a copy of the Liferay source code from the Liferay Portal repository to see how resources and permissions are defined for core Liferay portlets. Start by looking at the definition files found in the portal-impl/src/resource-actions directory. For a simple example of defining permissions in the context of a portlet plugin, check out the Liferay Plugins repository and examine the portlet sample-permissions-portlet.

CHAPTER 97

AUTHENTICATION PIPELINES

The authentication process in Liferay DXP is a pipeline through which users can be validated by one or several systems. Liferay DXP's flexibility and extensibility makes it possible for you to make it authenticate users to anything you wish, rather than be limited by what it supports out of the box.

Here's how authentication works under most circumstances:

1. Users provide their credentials to the Login Portlet to begin an authenticated session in a browser.

2. Alternatively, credentials are provided to Liferay DXP's API endpoints, where they are sent in an HTTP BASIC Auth header.

3. Alternatively, credentials can be provided by another system. These are managed by `AutoLogin` components.

4. Credentials are checked by default against the database, but they can be delegated to other systems instead of or in addition to it. This is called an *Authentication Pipeline*. You can add `Authenticators` to the pipeline to support any system.

5. You can also customize Liferay DXP's Login Portlet to support whatever user interface any of these systems need. This gives you full flexibility over the entire authentication process.

You can also support an authentication mechanism and/or accept credentials from a system that Liferay DXP doesn't yet support. If you don't like the user interface for signing in, you can replace it with your own.

This set of tutorials guides you through these customizations. You'll discover three kinds of customizations:

- Auto Login: the easiest of the three, this lets you authenticate to Liferay DXP using credentials provided in the HTTP header from another system.

- Authentication Pipelines: if you need to check credentials against other systems instead of or in addition to Liferay DXP's database, you can create a pipeline.

- Custom Login Portlet: if you want to change the user's sign-in experience completely, you can implement your own Login portlet.

Read on to discover how to customize your users' sign-in experience.

97.1 Auto Login

While Liferay DXP supports a wide variety of authentication mechanisms, you may use a home-grown system or some other product to authenticate users. To do so, you can write an Auto Login component to support your authentication system.

Auto Login components can check if the request contains something (a cookie, an attribute) that can be associated with a user in any way. If the component can make that association, it can authenticate that user to Liferay DXP.

Creating an Auto Login Component

Create a Declarative Services component. The component should implement the `com.liferay.portal.kernel.security.auto.lo` interface. Here's an example template:

```
import com.liferay.portal.kernel.security.auto.login.AutoLogin;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osgi.service.component.annotations.Component;

@Component(immediate = true)
public class MyAutoLogin implements Autologin {

    public String[] handleException(
            HttpServletRequest request, HttpServletResponse response,
            Exception e)
        throws AutoLoginException {

        /* This method is no longer used in the interface and can be
      left empty */

    }

    public String[] login(
            HttpServletRequest request, HttpServletResponse response)
        throws AutoLoginException {

        /* Your Code Goes Here */

    }

}
```

As you can see, you have access to the `HttpServletRequest` and the `HttpServletResponse` objects. If your sign-on solution places anything here that identifies a user, such as a cookie, an attribute, or a parameter, you can retrieve it and take whatever action you need to retrieve the user information and authenticate that user to Liferay DXP.

For example, say that there's a request attribute that contains the encrypted value of a user key in Liferay DXP. This can only be there if the user has authenticated with a third party system that knew the value of the user key, encrypted it, and added it as a request attribute. You could write code that reads the value, decrypts it using the same pre-shared key, and uses the value to look up and authenticate the user.

The `login` method is where this all happens. This method must return a `String` array with three items in this order:

- The user ID
- The user password

- A boolean flag that's true if the password is encrypted and false if it's not (Boolean.TRUE.toString() or Boolean.FALSE.toString()).

Sending redirects is an optional AutoLogin feature. Since AutoLogins are part of the servlet filter chain, you have two options. Both are implemented by setting attributes in the request. Here are the attributes:

- AutoLogin.AUTO_LOGIN_REDIRECT: This key causes AutoLoginFilter to stop the filter chain's execution and redirect immediately to the location specified in the attribute's value.
- AutoLogin.AUTO_LOGIN_REDIRECT_AND_CONTINUE: This key causes AutoLoginFilter to set the redirect and continue executing the remaining filters in the chain.

Auto Login components are useful ways of providing an authentication mechanism to a system that Liferay DXP doesn't yet support. You can write them fairly quickly to provide the integration you need.

Related Topics

Password-Based Authentication Pipelines

Writing a Custom Login Portlet

97.2 Password-Based Authentication Pipelines

By default, once a user submits credentials to Liferay DXP, those credentials are checked against Liferay DXP's database, though you can also delegate authentication to an LDAP server. To use some other system in your environment instead of or in addition to checking credentials against Liferay DXP's database, you can write an Authenticator and insert it as a step in Liferay DXP's authentication pipeline.

Because the Authenticator is checked by the Login Portlet, you can't use this approach if the user must be redirected to the external system or needs a token to authenticate. In those cases, you should use an Auto Login or an Auth Verifier.

Authenticators let you do these things:

- Log into Liferay DXP with a username and password maintained in an external system
- Make secondary user authentication checks
- Perform additional processing when user authentication fails

Read on to learn how to create an Authenticator.

Anatomy of an Authenticator

Authenticators are implemented for various steps in the authentication pipeline. Here are the steps:

1. auth.pipeline.pre: Comes before default authentication to the Liferay DXP database. In this step, you can instruct Liferay DXP to skip credential validation against the Liferay DXP database. Implemented by Authenticator.
2. Default (optional) authentication to the Liferay DXP database.
3. auth.pipeline.post: Further (secondary, tertiary) authentication checks. Implemented by Authenticator.
4. auth.failure: Perform additional processing after authentication fails. Implemented by AuthFailure.

To create an Authenticator, create a module and add a component that implements the interface:

```
@Component(
    immediate = true, property = {"key=auth.pipeline.post"},
    service = Authenticator.class
)
public class MyCustomAuth implements Authenticator {

    public int authenticateByEmailAddress(
            long companyId, String emailAddress, String password,
            Map<String, String[]> headerMap, Map<String, String[]> parameterMap)
        throws AuthException {

return Authenticator.SUCCESS;
}

    public int authenticateByScreenName(
            long companyId, String screenName, String password,
            Map<String, String[]> headerMap, Map<String, String[]> parameterMap)
        throws AuthException {

return Authenticator.SUCCESS;
    }

    public int authenticateByUserId(
            long companyId, long userId, String password,
            Map<String, String[]> headerMap, Map<String, String[]> parameterMap)
        throws AuthException {

return Authenticator.SUCCESS;
    }
}
```

This example has been stripped down so you can see its structure. First, note the @Component annotation's contents:

- immediate = true: sets the component to start immediately
- key=auth.pipeline.post: sets the Authenticator to run in the auth.pipeline.post phase. To run the auth.pipeline.pre phase, substitute auth.pipeline.pre.
- service = Authenticator.class: implements the Authenticator service. All Authenticators must do this.

The three methods below the annotation run based on how you've configured authentication: by email address (the default), by screen name, or by user ID. All the methods throw an AuthException in case the Authenticator is unable to perform its task–perhaps if the system it's authenticating against is unavailable or if some dependency can't be found. The methods in this barebones example return success in all cases. If you deploy its module, it has no effect. Naturally, you'll want to provide more functionality. Next is an example that shows you how to do that.

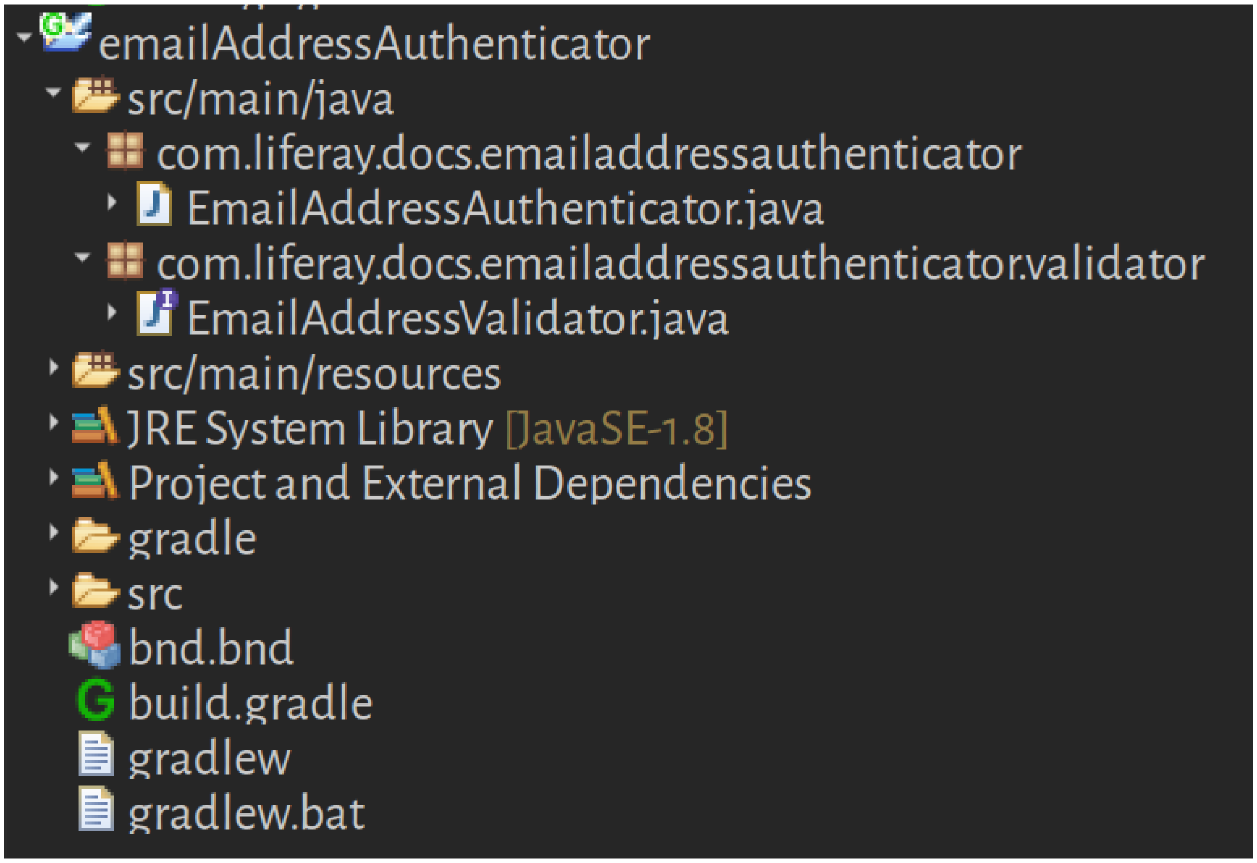
Creating an Authenticator

This example is an Authenticator that only allows users whose email addresses end with *@liferay.com* or *@example.com*. You can implement this using one module that does everything. If you think other modules might be able to use the functionality that validates the email addresses, you might create two modules: one to implement the Authenticator and one to validate email addresses. This example shows the two module approach.

To create an Authenticator, create a module for your implementation. The most appropriate Blade template for this is the service template. Once you have the module, creating the Activator is straightforward:

1. Add the @Component annotation to bind your Activator to the appropriate authentication pipeline phase.
2. Implement the Authenticator interface and provide the functionality you need.
3. Deploy your module. If you're using Blade CLI, do this via blade deploy.

For this example, you'll do this twice: once for the email address validator module and once for the Authenticator itself. The Authenticator project contains the interface for the validator, and the validator project contains the implementation. Here's what the Authenticator module structure looks like:



```
emailAddressAuthenticator
  src/main/java
    com.liferay.docs.emailaddressauthenticator
      EmailAddressAuthenticator.java
    com.liferay.docs.emailaddressauthenticator.validator
      EmailAddressValidator.java
  src/main/resources
  JRE System Library [JavaSE-1.8]
  Project and External Dependencies
  gradle
  src
  bnd.bnd
  build.gradle
  gradlew
  gradlew.bat
```

Figure 97.1: The Authenticator module contains the validator's interface and the authenticator.

Since the Authenticator is the most relevant, examine it first:

```
package com.liferay.docs.emailaddressauthenticator;

import java.util.Map;

import com.liferay.docs.emailaddressauthenticator.validator.EmailAddressValidator;
import com.liferay.portal.kernel.log.Log;
import com.liferay.portal.kernel.log.LogFactoryUtil;
import com.liferay.portal.kernel.security.auth.AuthException;
import com.liferay.portal.kernel.security.auth.Authenticator;
import com.liferay.portal.kernel.service.UserLocalService;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;
import org.osgi.service.component.annotations.ReferenceCardinality;
import org.osgi.service.component.annotations.ReferencePolicy;

@Component(
    immediate = true,
    property = {"key=auth.pipeline.post"},
    service = Authenticator.class
)
public class EmailAddressAuthenticator implements Authenticator {

    @Override
    public int authenticateByEmailAddress(long companyId, String emailAddress,
            String password, Map<String, String[]> headerMap,
            Map<String, String[]> parameterMap) throws AuthException {

        return validateDomain(emailAddress);
    }

    @Override
    public int authenticateByScreenName(long companyId, String screenName,
            String password, Map<String, String[]> headerMap,
            Map<String, String[]> parameterMap) throws AuthException {

        String emailAddress =
            _userLocalService.fetchUserByScreenName(companyId, screenName).getEmailAddress();

        return validateDomain(emailAddress);
    }

    @Override
    public int authenticateByUserId(long companyId, long userId,
            String password, Map<String, String[]> headerMap,
            Map<String, String[]> parameterMap) throws AuthException {

        String emailAddress =
            _userLocalService.fetchUserById(userId).getEmailAddress();

        return validateDomain(emailAddress);
    }

    private int validateDomain(String emailAddress) throws AuthException {

        if (_emailValidator == null) {

            String msg = "Email address validator is unavailable, cannot authenticate user";
            _log.error(msg);

            throw new AuthException(msg);
        }

        if (_emailValidator.isValidEmailAddress(emailAddress)) {
            return Authenticator.SUCCESS;
        }
        return Authenticator.FAILURE;
    }

    @Reference
    private volatile UserLocalService _userLocalService;

    @Reference(
        policy = ReferencePolicy.DYNAMIC,
        cardinality = ReferenceCardinality.OPTIONAL
    )
    private volatile EmailAddressValidator _emailValidator;

    private static final Log _log = LogFactoryUtil.getLog(EmailAddressAuthenticator.class);
}
```

This time, rather than stubs, the three authentication methods contain functionality. The authenticateByEmailAddress method directly checks the email address provided by the Login Portlet. The other two methods, authenticateByScreenName and authenticateByUserId call Liferay DXP's UserLocalService to look up the user's email address before checking it. This service is injected by the OSGi container because of the @Reference annotation. Note that the validator is also injected in this same manner, though it's configured not to fail if the implementation can't be found. This allows this module to start regardless of its dependency on the validator implementation. In this case, this is safe because the error is handled by throwing an AuthException and logging the error.

Why would you want to do it this way? To err gracefully. Because this is an auth.pipeline.post Authenticator, you presumably have other Authenticators checking credentials before this one. If this one isn't working, you want to inform administrators with an error message rather than catastrophically failing and preventing users from logging in.

The only other Java code in this module is the Interface for the validator:

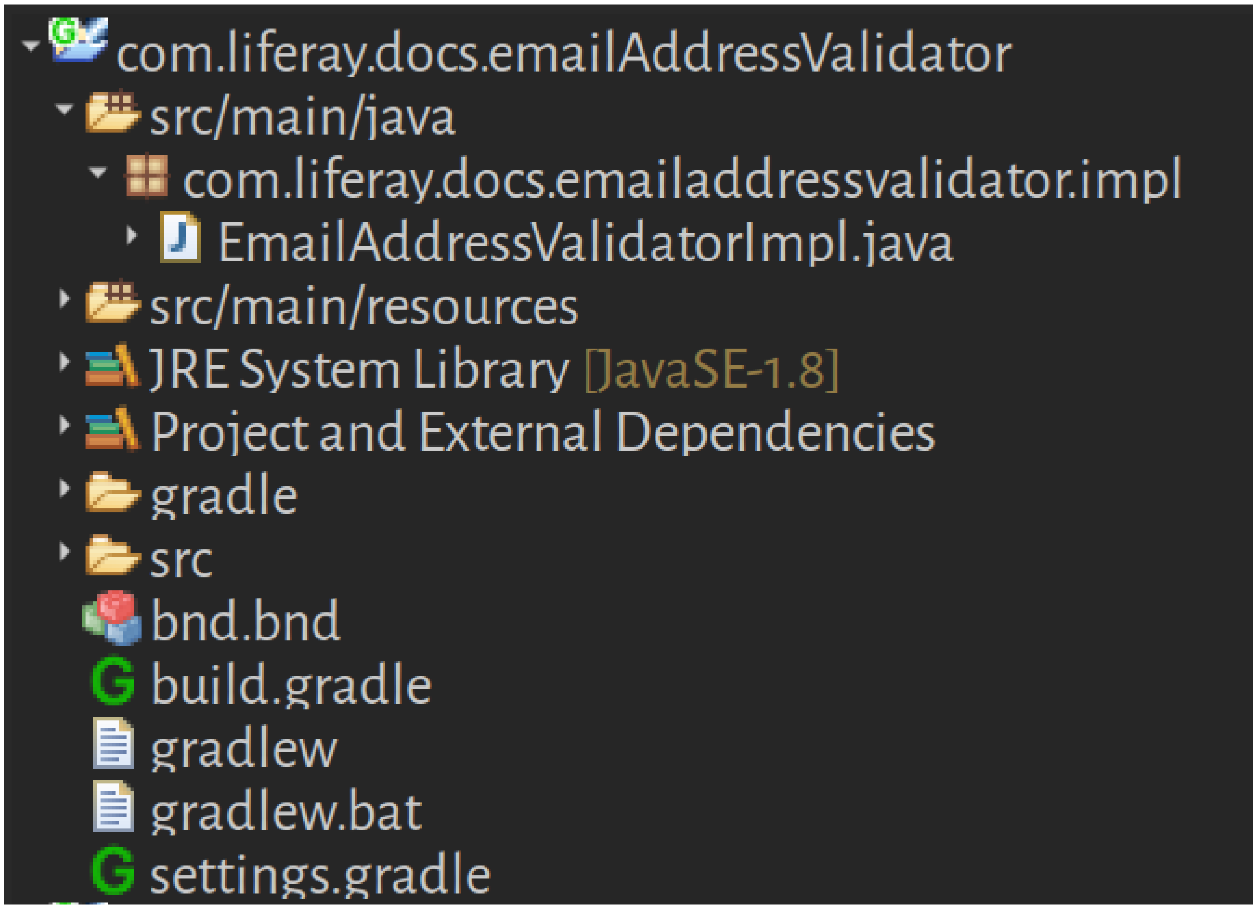
```
package com.liferay.docs.emailaddressauthenticator.validator;

import aQute.bnd.annotation.ProviderType;

@ProviderType
public interface EmailAddressValidator {

    public boolean isValidEmailAddress(String emailAddress);
}
```

This defines a single method for checking the email address.

Next, you'll address the validator module.

This module contains only one class. It implements the Validator interface:

```
package com.liferay.docs.emailaddressvalidator.impl;

import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import org.osgi.service.component.annotations.Component;
import com.liferay.docs.emailaddressauthenticator.validator.EmailAddressValidator;

@Component(
    immediate = true,
    property = {
    },
    service = EmailAddressValidator.class
)
public class EmailAddressValidatorImpl implements EmailAddressValidator {

    @Override
    public boolean isValidEmailAddress(String emailAddress) {

        if (_validEmailDomains.contains(
            emailAddress.substring(emailAddress.indexOf('@')))) {

            return true;
        }
        return false;
    }

    private Set<String> _validEmailDomains =
        new HashSet<String>(Arrays.asList(new String[] {"@liferay.com", "@example.com"}));
}
```



com.liferay.docs.emailAddressValidator
src/main/java
com.liferay.docs.emailaddressvalidator.impl
EmailAddressValidatorImpl.java
src/main/resources
JRE System Library [JavaSE-1.8]
Project and External Dependencies
gradle
src
bnd.bnd
build.gradle
gradlew
gradlew.bat
settings.gradle

Figure 97.2: The validator project implements the Validator Interface and depends on the authenticator module.

This code checks to make sure that the email address is from the *@liferay.com* or *@example.com* domains. The only other interesting part of this module is the Gradle build script, because it defines a compile-only dependency between the two projects. This is divided into two files: a `settings.gradle` and a `build.gradle`.

The `settings.gradle` file defines the location of the project (the `Authenticator`) the validator depends on:

```
include ':emailAddressAuthenticator'
project(':emailAddressAuthenticator').projectDir = new File(settingsDir, '../com.liferay.docs.emailAddressAuthenticator')
```

Since this project contains the interface, it must be on the classpath at compile time, which is when `build.gradle` is running:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins", version: "3.0.23"
    }

    repositories {
        mavenLocal()

        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}
```

```
apply plugin: "com.liferay.plugin"

dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "org.osgi", name: "org.osgi.compendium", version: "5.0.0"

    compileOnly project(":emailAddressAuthenticator")
}

repositories {
    mavenLocal()

    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Note the line in the dependencies section that refers to the `Authenticator` project defined in `settings.gradle`.

When these projects are deployed, the `Authenticator` you defined runs, enforcing logins for the two domains specified in the validator.

If you want to examine these projects further, you can download them in this ZIP file.

Related Topics

Auto Login

Writing a Custom Login Portlet

97.3 Writing a Custom Login Portlet

If you need to customize your users' authentication experience completely, you can write your own Login Portlet. The mechanics of this on the macro level are no different from writing any other portlet, so if you need to familiarize yourself with that, please see the portlets section of tutorials.

This tutorial shows only the relevant parts of a Liferay MVC Portlet that authenticates the user. You'll learn how to call Liferay DXP's authentication pipeline and then redirect the user to a location of your choice.

Authenticating to Liferay DXP

You can use the example project in this ZIP file as a starting point for your own.

It has only one view, which is used for logging in or showing the user who is already logged in:

```
<%@ include file="/init.jsp" %>

<p>
    <b><liferay-ui:message key="myloginportlet_MyLogin.caption"/></b>
</p>

<c:choose>
    <c:when test="<%= themeDisplay.isSignedIn() %>">

        <%
        String signedInAs = HtmlUtil.escape(user.getFullName());

        if (themeDisplay.isShowMyAccountIcon() && (themeDisplay.getURLMyAccount() != null)) {
            String myAccountURL = String.valueOf(themeDisplay.getURLMyAccount());

            signedInAs = "<a class=\"signed-in\" href=\"" + HtmlUtil.escape(myAccountURL) + "\">" + signedInAs + "</a>";
        }
        %>

        <liferay-ui:message arguments="<%= signedInAs %>" key="you-are-signed-in-as-x" translateArguments="<%= false %>" />
    </c:when>
    <c:otherwise>

        <%
        String redirect = ParamUtil.getString(request, "redirect");
        %>

        <portlet:actionURL name="/login/login" var="loginURL">
            <portlet:param name="mvcRenderCommandName" value="/login/login" />
        </portlet:actionURL>

        <aui:form action="<%= loginURL %>" autocomplete='on' cssClass="sign-in-form" method="post" name="loginForm">

            <aui:input name="saveLastPath" type="hidden" value="<%= false %>" />
            <aui:input name="redirect" type="hidden" value="<%= redirect %>" />

            <aui:input autoFocus="true" cssClass="clearable" label="email-address" name="login" showRequiredLabel="<%= false %>" type="text" value="">
                <aui:validator name="required" />
            </aui:input>

            <aui:input name="password" showRequiredLabel="<%= false %>" type="password">
                <aui:validator name="required" />
            </aui:input>

            <aui:button-row>
                <aui:button cssClass="btn-lg" type="submit" value="sign-in" />
            </aui:button-row>

        </aui:form>
    </c:otherwise>
</c:choose>
```

Note that in the form, authentication by email address (Liferay DXP's default setting) is hard-coded, as this is an example project. The current page is sent as a hidden field on the form so the portlet can redirect the user to it, but you can of course set this to any value you want.

The portlet handles all processing of this form using a single Action Command (imports left out for brevity):

```
@Component(
    property = {
        "javax.portlet.name=MyLoginPortlet",
        "mvc.command.name=/login/login"
    },
    service = MVCActionCommand.class
)
public class MyLoginMVCActionCommand extends BaseMVCActionCommand {

    @Override
    protected void doProcessAction(ActionRequest actionRequest,
            ActionResponse actionResponse) throws Exception {

        ThemeDisplay themeDisplay = (ThemeDisplay)actionRequest.getAttribute(
            WebKeys.THEME_DISPLAY);

        HttpServletRequest request = PortalUtil.getOriginalServletRequest(
            PortalUtil.getHttpServletRequest(actionRequest));

        HttpServletResponse response = PortalUtil.getHttpServletResponse(
            actionResponse);

        String login = ParamUtil.getString(actionRequest, "login");
        String password = actionRequest.getParameter("password");
        boolean rememberMe = ParamUtil.getBoolean(actionRequest, "rememberMe");
        String authType = CompanyConstants.AUTH_TYPE_EA;

        AuthenticatedSessionManagerUtil.login(
            request, response, login, password, rememberMe, authType);

        actionResponse.sendRedirect(themeDisplay.getPathMain());
    }

}
```

The only tricky/unusual code here is the need to grab the `HttpServletRequest` and the `HttpServletResponse`. This is necessary to call Liferay DXP's API for authentication. At the end of the Action Command, the portlet sends a redirect that sends the user to the same page. You can of course make this any page you want.

Implementing your own login portlet gives you complete control over the authentication process.

Related Topics

Password-Based Authentication Pipelines

Auto Login

97.4 Service Access Policies

Service access policies are a layer of web service security on top of Liferay DXP's remote services. Together with the permissions layer, service access policies limit remote service access by remote client applications. This forms an additional security layer that protects user data from unauthorized access and modification.

To connect to a Liferay DXP instance, remote clients must authenticate with credentials in that instance. This grants the remote client the permissions assigned to those credentials in the Liferay DXP installation. Service access policies are a layer of security on top of this: they further limit the remote client's access to the remote services specified in the policy. Without such policies, authenticated remote clients are treated like users: they can call any remote API and read or modify data on behalf of the authenticated user. Since remote clients are often intended for a specific use case, granting them access to everything the user has permissions for poses a security risk.

For example, consider a mobile app (client) that displays a user's appointments from the Liferay Calendar app. This client app doesn't need access to the API that updates the user profile, even though the user has such permissions on the server. The client app doesn't even need access to the Calendar API methods that create, update, and delete appointments. It only needs access to the remote service methods for finding and retrieving appointments. A service access policy on the server can restrict the client's access to only these service methods. Otherwise, once authenticated it would have access to all the remote services the user has permission to access when logged in: services that create, update, and delete calendar appointments, as well as those that can update user data or other system entities the user can access. Since the client doesn't perform these operations, having access to them is a security risk if the mobile device is lost or stolen or the client app is compromised by an attacker.

How Service Access Policies Work

When a remote client issues a request to a web service, the request contains the user's credentials or an authorization token. An authentication module in Liferay DXP recognizes the client based on the credentials/token and grants the appropriate service access policy to the request. The service access policy authorization layer then processes all granted policies and lets the request access the remote service(s) permitted by the policy.

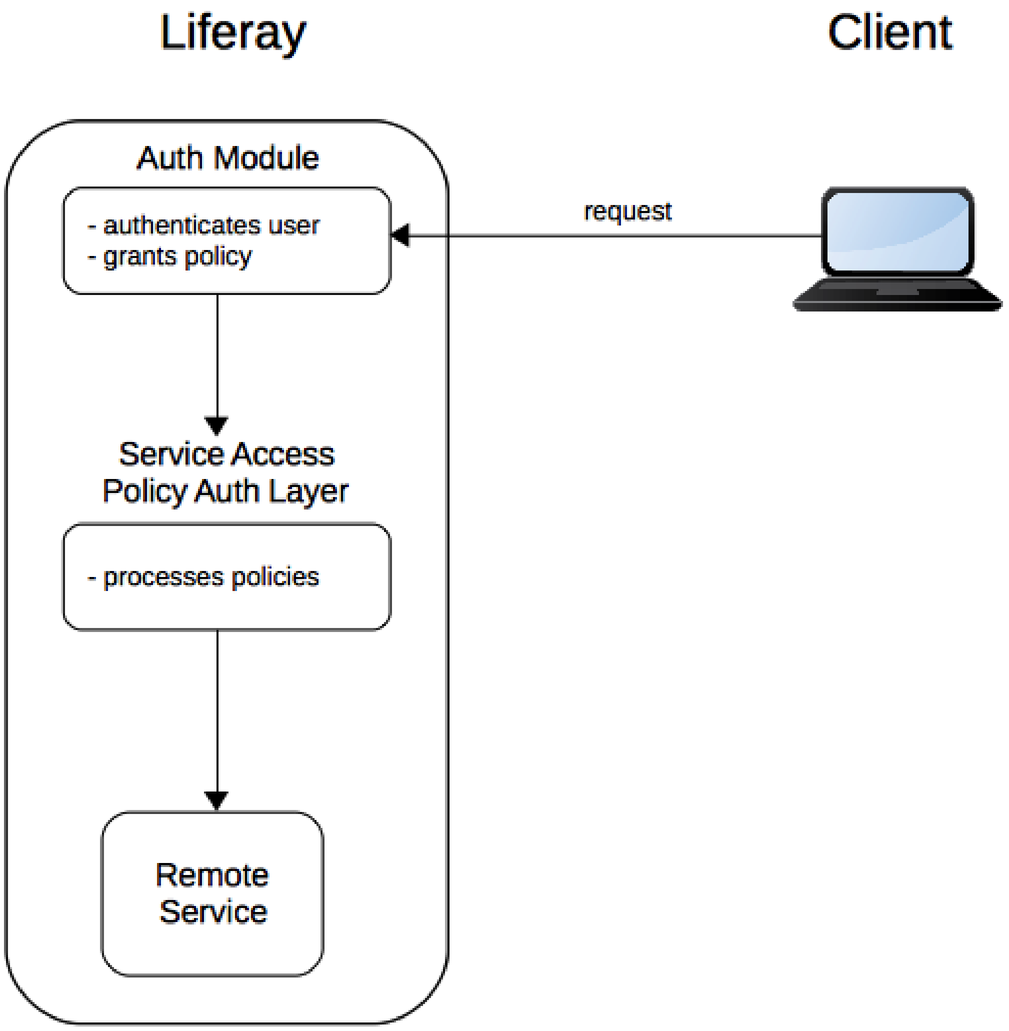


Figure 97.3: The authorization module maps the credentials or token to the proper Service Access Policy.

Service Access policies are created in the Control Panel by administrators. If you want to start creating policies yourself, see this article on service access policies that documents creating them in the UI.

There may be cases, however, when your server-side Liferay app needs to use the service access policies API. For example, your app may:

- use custom remote API authentication (tokens) and require certain services to be available for clients using the tokens.
- require its services be made available to guest users, with no authentication necessary.
- contain a remote service authorization layer that needs to drive access to remote services based on granted privileges.

API Overview

Liferay provides an Interface and a `ThreadLocal` if you don't want to roll your own policies. If you want to get low level, an API is provided that Liferay itself has used to implement Liferay Sync.

1. The Interface and `ThreadLocal` are available in the package `com.liferay.portal.kernel.security.service.access.policy` This package provides classes for basic access to policies. For example, you can use the singleton `ServiceAccessPolicyManagerUtil` to obtain Service Access Policies configured in the system. You can also use the `ServiceAccessPolicyThreadLocal` class to set and obtain Service Access Policies granted to the current request thread.

   At this level, you can get a list of the configured policies to let your app/client choose a policy for accessing services. Also, apps like OAuth can offer a list of available policies during the authorization step in the OAuth workflow and allow the user to choose the policy to assign to the remote application. You can also grant a policy to a current request thread. When a remote client accesses an API, something must tell the Liferay instance which policies are assigned to this call. This something is in most cases an `AuthVerifier` implementation. For example, in the case of the OAuth app, an `AuthVerifier` implementation assigns the policy chosen by the user in the authorization step.

2. The API ships with the product as OSGi modules:

- `com.liferay.portal.security.service.access.policy.api.jar`
- `com.liferay.portal.security.service.access.policy.service.jar`
- `com.liferay.portal.security.service.access.policy.web.jar`

  These OSGi modules are active in Liferay DXP by default, and you can use them to manage Service Access Policies programmatically. You can find their source code [here in GitHub](https://github.com/liferay/liferay-portal/tree/master/modules/apps/foundation/portal-security). Each module publishes a list of packages and services that can be consumed by other OSGi modules.

You can use both tools to develop a token verification module (a module that implements custom security token verification for use in authorizing remote clients) for your app to use. For example, this module may contain a JSON Web Token implementation for Liferay DXP's remote API. A custom token verification module must use the Service Access Policies API during the remote API/web service call to grant the associated policy during the request. The module:

- can use `com.liferay.portal.security.service.access.policy.api.jar` and `com.liferay.portal.security.service.acce` to create policies programmatically.

- should use the method ServiceAccessPolicyThreadLocal.addActiveServiceAccessPolicyName() to grant the associated policy during a web service request.

- can use ServiceAccessPolicyManagerUtil to display list of supported policies when authorizing the remote application, to associate the token with an existing policy.

Example

Liferay Sync's sync-security module is an example of such a module. It uses com.liferay.portal.security.service.access.pol to create the SYNC_DEFAULT and SYNC_TOKEN policies programmatically. For service calls to Sync's remote API, these policies grant access to Sync's com.liferay.sync.service.SyncDLObjectService#getSyncContext and com.liferay.sync.service.*, respectively. Here's the code in the sync-security module that defines and creates these policies:

```
@Component(immediate = true)
public class SyncSAPEntryActivator {

    // Define the policies
    public static final Object[][] SAP_ENTRY_OBJECT_ARRAYS = new Object[][] {
        {
            "SYNC_DEFAULT",
            "com.liferay.sync.service.SyncDLObjectService#getSyncContext", true
        },
        {"SYNC_TOKEN", "com.liferay.sync.service.*", false}
    };

    ...

    // Create the policies
    protected void addSAPEntry(long companyId) throws PortalException {
            for (Object[] sapEntryObjectArray : SAP_ENTRY_OBJECT_ARRAYS) {
                String name = String.valueOf(sapEntryObjectArray[0]);
                String allowedServiceSignatures = String.valueOf(
                    sapEntryObjectArray[1]);
                boolean defaultSAPEntry = GetterUtil.getBoolean(
                    sapEntryObjectArray[2]);

                SAPEntry sapEntry = _sapEntryLocalService.fetchSAPEntry(
                    companyId, name);

                if (sapEntry ≠ null) {
                    continue;
                }

                Map<Locale, String> map = new HashMap<>();

                map.put(LocaleUtil.getDefault(), name);

                _sapEntryLocalService.addSAPEntry(
                    _userLocalService.getDefaultUserId(companyId),
                    allowedServiceSignatures, defaultSAPEntry, true, name, map,
                    new ServiceContext());
            }
    }

    ...

}
```

Click here to see the entire SyncSAPEntryActivator class. This class creates the policies when the module starts. Note that this module is included and enabled in Liferay DXP by default. You can access these and other policies in *Control Panel → Configuration → Service Access Policy*.

The sync-security module must then grant the appropriate policy when needed. Since every authenticated call to Liferay Sync's remote API requires access to com.liferay.sync.service.*, the module must grant the SYNC_TOKEN policy to such calls. The module does this with the method ServiceAccessPolicyThreadLocal.addActiveServiceAccessPolicyName, as shown in this code snippet:

```
if ((permissionChecker ≠ null) && permissionChecker.isSignedIn()) {
    ServiceAccessPolicyThreadLocal.addActiveServiceAccessPolicyName(
        String.valueOf(
            SyncSAPEntryActivator.SAP_ENTRY_OBJECT_ARRAYS[1][0]));
}
```

Now every authenticated call to Sync's remote API, regardless of authentication method, has access to com.liferay.sync.service.*. To see the full code example, click here.

Nice! Now you know how to integrate your apps with the Service Access Policies in Liferay DXP.

97.5 Using JSR Roles in a Portlet

Roles in Liferay DXP are the primary means for granting or restricting access to content. If you've decided *not* to use Liferay's permissions system, you can use the basic system offered by the JSR 168, 286, and 362 specifications that map Roles in a portlet to Roles provided by the portal.

JSR Portlet Security

The portlet specification defines a means to specify Roles used by portlets in their docroot/WEB-INF/portlet.xml descriptors. The Role names themselves, however, are not standardized. When these portlets run in Liferay DXP, the Role names defined in the portlet must be mapped to Roles that exist in the Portal.

For example, consider a Guestbook project that contains two portlets: The Guestbook portlet and the Guestbook Admin portlet. The WAR version of the Guestbook project's portlet.xml file references the *administrator*, *guest*, *power-user*, and *user* Roles:

```
<?xml version="1.0"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http:/
app_2_0.xsd http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd" version="2.0">

<portlet>
    <portlet-name>guestbook</portlet-name>
    <display-name>Guestbook</display-name>
    <portlet-class>
        com.liferay.docs.guestbook.portlet.GuestbookPortlet
    </portlet-class>
    <init-param>
        <name>view-template</name>
        <value>/html/guestbook/view.jsp</value>
    </init-param>
    <expiration-cache>0</expiration-cache>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
        <title>Guestbook</title>
        <short-title>Guestbook</short-title>
        <keywords></keywords>
    </portlet-info>
    <security-role-ref>
        <role-name>administrator</role-name>
    </security-role-ref>
    <security-role-ref>
        <role-name>guest</role-name>
    </security-role-ref>
    <security-role-ref>
        <role-name>power-user</role-name>
    </security-role-ref>
    <security-role-ref>
        <role-name>user</role-name>
    </security-role-ref>
</portlet>
<portlet>
    <portlet-name>guestbook-admin</portlet-name>
    <display-name>Guestbook Admin</display-name>
    <portlet-class>
        com.liferay.docs.guestbook.portlet.GuestbookAdminPortlet
    </portlet-class>
    <init-param>
        <name>view-template</name>
        <value>/html/guestbookadmin/view.jsp</value>
    </init-param>
    <expiration-cache>0</expiration-cache>
    <supports>
        <mime-type>text/html</mime-type>
        <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
        <title>Guestbook Admin</title>
        <short-title>Guestbook Admin</short-title>
        <keywords></keywords>
    </portlet-info>
    <security-role-ref>
        <role-name>administrator</role-name>
    </security-role-ref>
    <security-role-ref>
        <role-name>guest</role-name>
    </security-role-ref>
    <security-role-ref>
        <role-name>power-user</role-name>
    </security-role-ref>
    <security-role-ref>
        <role-name>user</role-name>
    </security-role-ref>
</portlet>
```

An OSGi-based `guestbook-web` module project defines Roles without an XML file, in the portlet class's `@Component` annotation:

```
@Component(
    immediate = true,
    property = {
        "com.liferay.portlet.display-category=category.sample",
        "com.liferay.portlet.instanceable=true",
        "javax.portlet.init-param.template-path=/",
        "javax.portlet.init-param.view-template=/view.jsp",
        "javax.portlet.name=" + GuestbookPortletKeys.Guestbook,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
```

If you are using an OSGi-based MVC Portlet, you must use Liferay's permissions system, as the only way to map JSR-362 Roles to Liferay Roles is to place them in the Liferay WAR file's `portlet.xml`.

Your `portlet.xml` Roles must be mapped to specific Roles that have been created. These mappings allow Liferay DXP to resolve conflicts between Roles with the same name that are from different portlets (e.g. portlets from different developers).

Note: Each Role named in a portlet's `<security-role-ref>` element is given permission to add the portlet to a page.

Mapping Portlet Roles to Portal Roles

To map the Roles to Liferay DXP, you must use the `docroot/WEB-INF/liferay-portlet.xml` Liferay-specific configuration file. For an example, see the mapping defined in the Guestbook project's `liferay-portlet.xml` file.

```
<role-mapper>
    <role-name>administrator</role-name>
    <role-link>Administrator</role-link>
</role-mapper>
<role-mapper>
    <role-name>guest</role-name>
    <role-link>Guest</role-link>
</role-mapper>
<role-mapper>
    <role-name>power-user</role-name>
    <role-link>Power User</role-link>
</role-mapper>
<role-mapper>
    <role-name>user</role-name>
    <role-link>User</role-link>
</role-mapper>
```

If a portlet definition references the Role `power-user`, that portlet is mapped to the Liferay Role called *Power User* that's already in Liferay's database.

As stated above, there is no standardization with portal Role names. If you deploy a portlet with Role names different from the above default Liferay names, you must add the names to the `system.roles` property in your `portal-ext.properties` file:

```
system.roles=my-role,your-role,our-role
```

This prevents Roles from being created accidentally.

Once Roles are mapped to the portal, you can use methods as defined in the portlet specification:

- `getRemoteUser()`
- `isUserInRole()`
- `getUserPrincipal()`

For example, you can use the following code to check if the current User has the `power-user` Role:

```
if (renderRequest.isUserInRole("power-user")) {
    // ...
}
```

By default, Liferay doesn't use the `isUserInRole()` method in any built-in portlets. Liferay uses its own permission system directly to achieve more fine-grained security. If you don't intend on deploying your portlets to other portal servers, we recommend using Liferay's permission system, because it offers a much more robust way of tailoring your application's permissions.

Related Topics

Liferay Permissions
    Asset Framework
    Portlets
    Understanding ServiceContext

Chapter 98

Internationalization

Liferay DXP makes it easier than ever to localize content and design apps for different locales. You can centralize messages (language keys) and have manual and automatic translation, including localizing forms and setting text directionality from left-to-right or right-to-left. Customizing messages in apps is easy too. Internationalization is a snap with Liferay DXP!

98.1 Localizing Your Application

If you're writing a Liferay Application, you're probably a genius who is also really cool. Which means your application is going to be used throughout the entire world. At least, it will if the messages that appear to its users can be translated into their language. Thankfully, Liferay makes it easy to support translation of your application's language keys.

Note: Even if you don't think your application needs to be translated into multiple languages, use the localization pattern presented here for any messages displayed in your user interface. It's much easier to change the messages by updating a language properties file than by finding every instance of a message and replacing it in your JSPs and Java classes.

You just need to create a default language properties file (`Language.properties`) and one for each translation you'd like to support (for example, `Language_fr.properties` for your French translation), and put them in the correct location in your application. Use the two letter locale that corresponds to the language you want to translate in your file names (for example, `Language_es.properties` provides a Spanish translation for each key).

Application localization topics:

- What are Language Keys?
- What Locales are Available By Default?
- Where do I Put Language Files?
- Creating a Language Module
- Using a Language Module
- Using Global Language Properties

What are Language Keys?

Each language property file holds key/value pairs. The key is the same in all the language property files, while the value is translated in each file. You specify the key in your user interface code, and the appropriately translated message is returned automatically for your users, depending on the locale being used in Liferay. If you have Liferay running locally, append the URL with a supported locale to see how Liferay's language keys are translated (for example, enter `localhost:8080/es`).

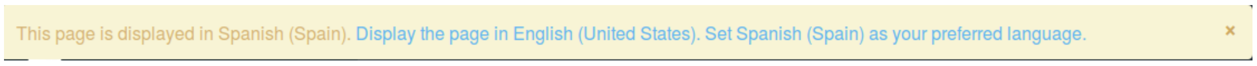

This page is displayed in Spanish (Spain). Display the page in English (United States). Set Spanish (Spain) as your preferred language. ×

Figure 98.1: Append the locale to your running Liferay's URL and see Liferay's translation power in action.

Language keys are just keys you'll use in place of a hard coded, fully translated String value in your user interface code. For example, you can use a language key in your JSP via a `<liferay-ui:message />` tag.

The tag might be set up like this if you're not considering the need to translate your application's messages:

```
<liferay-ui:message key="Howdy, Partner!" />
```

In that case you'll get a properly capitalized and punctuated message in your application.

Alternatively, you can specify a simple key instead of the final value:

```
<liferay-ui:message key="howdy-partner" />
```

That way you can provide a translation of the key in a default language properties file (`Language.properties`):

```
howdy-partner=Howdy, Partner!
```

You'll get the same output in your application with either method above, but you have the flexibility to add additional language properties files that provide translations for your application's keys if you use the language properties approach. Use a key in your UI code, then provide the value (or translation) in your language properties file. You just need to make sure there's a locale that corresponds to your translation.

The values from your default `Language.properties` file will appear if no locale is specified. If a locale is specified, Liferay will try to find a file that corresponds to the locale. For example, if a Spanish translation is sought, a `Language_es.properties` file must be present to provide the proper values. If it isn't, the default language properties (from the `Language.properties` file) will be used.

What Locales are Available By Default?

There are a bunch of locales available by default in Liferay. Look in the `portal.properties` file file to find them.

```
locales=ar_SA,eu_ES,bg_BG,ca_AD,ca_ES,zh_CN,zh_TW,hr_HR,cs_CZ,da_DK,nl_NL,
    nl_BE,en_US,en_GB,en_AU,et_EE,fi_FI,fr_FR,fr_CA,gl_ES,de_DE,el_GR,
    iw_IL,hi_IN,hu_HU,in_ID,it_IT,ja_JP,ko_KR,lo_LA,lt_LT,nb_NO,fa_IR,
    pl_PL,pt_BR,pt_PT,ro_RO,ru_RU,sr_RS,sr_RS_latin,sl_SI,sk_SK,es_ES,
    sv_SE,tr_TR,uk_UA,vi_VN
```

To provide a translation for one of these locales, specify the locale in the file name where the translated keys will be (for example, `Langauge_es.properties` holds the Spanish translation).

Where do I Put Language Files?

In an application with only one module that holds all your application's views (for example, all its JSPs) and portlet components, just create a src/main/resources/content folder in that module, and place your Language.properties and Language_xx.properties files there.

After that, make sure any portlet components (the @Component annotation in your -Portlet classes) in the module include this property:

```
"javax.portlet.resource-bundle=content.Language"
```

Providing translated language properties files and specifying the javax.portlet.resource-bundle property in your portlet component is all you need to do to have your language keys translated. Then, when the locale is changed in Liferay DXP, your application's language keys will be automatically translated.

In a more complicated, well-modularized application, you might have language keys spread over multiple modules providing portlet components and JSP files. Moreover, there might be a fair number of duplicated language keys between the modules. Thankfully you don't need to maintain language properties files in each module.

Creating a Language Module

If you're crazy about modularity (and you should be), you might have an application with multiple modules that provide the view layer. These modules are often called web modules.

```
my-application/
my-application-web/
my-admin-application-web/
my-application-content-web/
my-application-api/
my-application-service/
```

Each of these modules can have language keys and translations to maintain, and there will probably be duplicate keys. You don't want to end up with different values for the same key, and you don't want to maintain language keys in multiple places. In this case, you need to go even crazier with modularity and create a new module, which we'll call a language module.

In the root project folder (the one that holds your service, API, and web modules), create a new module to hold your app's language keys. For example, here's the folder structure of a language module called my-application-lang.

```
my-application-lang/
    bnd.bnd
    src/
        main/
            resources/
                content/
                    Language.properties
                    Language_ar.properties
                    Language_bg.properties
                    ...
```

In the language module, create a src/main/resources/content folder. Put your language properties files here. A Language.properties file might look like this:

```
application=My Application
add-entity=Add Entity
```

Create any translations you want, adding the translation locale ID to the language file name. File Language_es.properties might look like this:

```
my-app-title=Mi Aplicación
add-entity=Añadir Entity
```

On building the language module, Liferay DXP's ResourceBundleLoaderAnalyzerPlugin detects the content/Language.properties file and adds a resource bundle *capability* to the module. A capability is a contract a module declares to Liferay DXP's OSGi framework. Capabilities let you associate services with modules that provide them. In this case, Liferay DXP registers a ResourceBundleLoader service for the resource bundle capability.

Next, you'll configure a web module to use the language module resource bundle.

Using a Language Module

A module or traditional Liferay plugin can use a resource bundle from another module and optionally include its own resource bundle. OSGi manifest headers Require-Capability and Provide-Capability make this possible, and it's especially easy in modules generated from Liferay project templates. Instructions for using a language module are divided into these environments:

- Using a Language Module from a Module
- Using a Language Module from a Traditional Plugin

If you're using bnd with Maven or Gradle, you need only specify Liferay's -liferay-aggregate-resource-bundle: bnd instruction–at build time, Liferay's bnd plugin converts the instruction to Require-Capability and Provide-Capability parameters automatically. Both approaches are demonstrated.

Using a Language Module from a Module

Modules generated from Liferay project templates have a Liferay bnd build time instruction called -liferay-aggregate-resource-bundles. It lets you use other resource bundles (e.g., including their language keys) along with your own. Here's how to do it:

1. Open your module's bnd.bnd file.

2. Add the -liferay-aggregate-resource-bundles: bnd instruction and assign it the bundle symbolic names of modules whose resource bundles to aggregate with the current module's resource bundle.

   ```
   -liferay-aggregate-resource-bundles: \
       [bundle.symbolic.name1],\
       [bundle.symbolic.name2]
   ```

For example, a module that uses resource bundles from modules com.liferay.docs.l10n.myapp1.lang and com.liferay.docs.l10n.myapp2.lang would set this in its bnd.bnd file:

```
-liferay-aggregate-resource-bundles: \
    com.liferay.docs.l10n.myapp1.lang,\
    com.liferay.docs.l10n.myapp2.lang
```

The current module's resource bundle is prioritized over those of the listed modules.

The Shared Language Key sample project is a working example that demonstrates aggregating resource bundles. You can deploy it in Gradle, Maven, and Liferay Workspace build environments.

At build time, Liferay's bnd plugin converts the bnd instruction to `Require-Capability` and `Provide-Capability` parameters automatically. In traditional Liferay plugins, you must specify the parameters manually.

Note: You can always specify the `Require-Capability` and `Provide- Capability` OSGi manifest headers manually, as the next section demonstrates.

Using a Language Module from a Traditional Plugin

To use a language module, from a traditional Liferay plugin you must specify the language module using `Require-Capability` and `Provide-Capability` OSGi manifest headers in the plugin's `liferay-plugin-package.properties` file.

Follow these steps to configure your traditional plugin to use a language module:

1. Open the plugin's `liferay-plugin-package.properties` file and add a `Require-Capability` header that filters on the language module's resource bundle capability. For example, if the language module's symbolic name is `myapp.lang`, you'd specify the requirement like this:

   ```
   Require-Capability: liferay.resource.bundle;filter:="(bundle.symbolic.name=myapp.lang)"
   ```

2. In the same `liferay-plugin-package.properties` file, add a `Provide-Capability` header that adds the language module's resource bundle *as* this plugin's (the `myapp.web` plugin) own resource bundle:

   ```
   Provide-Capability:\
   liferay.resource.bundle;resource.bundle.base.name="content.Language",\
   liferay.resource.bundle;resource.bundle.aggregate:String="(bundle.symbolic.name=myapp.lang)";bundle.symbolic.name=myapp.web;resource.bundle.base.nam
   servlet.context.name=myapp-web
   ```

In this case, the `myapp.web` plugin solely uses the language module's resource bundle—the resource bundle aggregate only includes language module `myapp.lang`.

Aggregating resource bundles comes into play when you want to use your a language module's resource bundle *in addition to* your plugin's resource bundle. These instructions show you how to do this, while prioritizing your current plugin's resource bundle over the language module resource bundle. In this way, the language module's language keys compliment your plugin's language keys.

For example, a portlet whose bundle symbolic name is `myapp.web` uses keys from language module `myapp.lang`, in addition to its own. The portlet's `Provide-Capability` and `Web-ContextPath` OSGi headers accomplish this.

```
Provide-Capability:\
liferay.resource.bundle;resource.bundle.base.name="content.Language",\
liferay.resource.bundle;resource.bundle.aggregate:String="(bundle.symbolic.name=myapp.web),(bundle.symbolic.name=myapp.lang)";bundle.symbolic.name=myapp.web
servlet.context.name=myapp-web
```

Let's examine the example `Provide-Capability` header.

1. `liferay.resource.bundle;resource.bundle.base.name="content.Language"` declares that the module provides a resource bundle whose base name is `content.language`.

2. The `liferay.resource.bundle;resource.bundle.aggregate:String=...` directive specifies the list of bundles whose resource bundles are aggregated, the target bundle, the target bundle's resource bundle name, and this service's ranking:

   - `"(bundle.symbolic.name=myapp.web),(bundle.symbolic.name=myapp.lang)"`: The service aggregates resource bundles from bundles `bundle.symbolic.name=myapp.web` (the current module) and `bundle.symbolic.name=myapp.lang`. Aggregate as many bundles as desired. Listed bundles are prioritized in descending order.
   - `bundle.symbolic.name=myapp.web;resource.bundle.base.name="content.Language"`: Override the `myapp.web` bundle's resource bundle named `content.Language`.
   - `service.ranking:Long="4"`: The resource bundle's service ranking is `4`. The OSGi framework applies this service if it outranks all other resource bundle services that target `myapp.web`'s `content.Language` resource bundle.
   - `servlet.context.name=myapp-web`: The target resource bundle is in servlet context `myapp-web`.

Now the language keys from the aggregated resource bundles compliment your plugin's language keys. Did you know that Liferay DXP's core language keys are also available to your module? They're up next.

Using Global Language Properties

If you have Liferay DXP's source code, you can check out Liferay DXP's core language properties by looking in the `portal-impl/src/main/content` folder. Otherwise, you can look in the `portal-impl.jar` that's in your Liferay bundle.

```
liferay-portal/portal-impl/src/content/Language_xx.properties
```

```
[Liferay Home]/tomcat-[version]/webapps/ROOT/WEB-INF/lib/portal-impl.jar
```

These keys are available at runtime, so when you use any of Liferay DXP's default keys in your user interface code, they're automagically swapped out for the appropriately translated value. Using Liferay DXP's keys where possible saves you time and ensures that your application follows Liferay's UI conventions.

If you want to generate language files for each supported locale automatically, or to configure your application to generate translations automatically using the Microsoft Translator API, check out the tutorial Automatically Generating Language Files.

98.2 Automatically Generating Language Files

If you already have a `Language.properties` file that holds language keys for your user interface messages, or even a language module that holds these keys, you're in the right place.

- Instead of manually creating a language properties file for each locale that's supported by Liferay, you can get them all automatically generated for you with one command. The same command also propagates the keys from the default language file to all translation files.

- You can also generate automatic translations using Microsoft's Translator Text API.

Generating Language Files for Supported Locales

If you want to generate files automatically for all locales supported by Liferay, you must make a small modification to your application's build file.

1. Make sure your module's build includes the `com.liferay.lang.builder` plugin, by putting the plugin in build script classpath.

2. Make sure you have a default `Language.properties` file in `src/main/resources/content`.

3. Run the `gradle buildLang` task from your project's root directory to generate default translation files.

   The generated files contain automatic copies of all the keys and values in your default `Language.properties` files. That way you don't have to copy your lanugage keys manually into all of the files. Run the `buildLang` task each time you change the default language file.

   When the task completes, it prints `BUILD SUCCESSFUL` with this log output:

   ```
   Translation is disabled because credentials are not specified
   ```

   See the next section to learn how to turn translation on and provide credentials.

Here's what a configuration of the `com.liferay.lang.builder` plugin looks in a `build.gradle` file:

```
buildscript {
    dependencies {
        classpath 'com.liferay:com.liferay.gradle.plugins.lang.builder:latest.release'
    }

    repositories {
        maven {
            url "http://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.lang.builder"

repositories {
    maven {
        url "http://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Now you can start translating your application's messages. If you want to configure your app to generate automatic translations using the Microsoft Translator Text API, keep reading.

Translating Language Keys Automatically

If you've configured the `com.liferay.lang.builder` plugin in your app, you're almost there. Now you have to configure Microsoft's Translator Text API so you can generate automatic translations of your language keys. You cannot, however, use Liferay's Lang Builder to automatically translate language keys containing HTML (e.g., `<em>`, `<b>`, `<code>`, etc.). Language keys containing HTML are automatically *copied* to all supported language files.

Note: These translations are best used as a starting point. A machine translation can't match the accuracy of a real person who is fluent in the language. Then again, if you only speak English and you need a Hungarian translation, this is better and faster than your attempts at a manual translation.

---

1. Generate a translation subscription key for the Microsoft Translator Text API. Follow the instructions here.

2. Make sure the buildLang task knows to use your subscription key for translation by setting the translateSubscriptionKey property:

   ```
   buildLang {
       translateSubscriptionKey = "my-key"
   }
   ```

   For security reasons you probably don't want to pass them directly in your application's build script. Instead, pass the credentials to a property that's stored in your local build environment, and pass the property into your application's build script.

   ```
    buildLang {
      translateSubscriptionKey = langTranslateSubscriptionKey
   }
   ```

So what would the complete buildLang configuration look like if you followed all the steps above?

```
buildscript {
    dependencies {
        classpath 'com.liferay:com.liferay.gradle.plugins.lang.builder:latest.release'
    }

    repositories {
        maven {
            url "http://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.lang.builder"

buildLang {
   translateSubscriptionKey = langTranslateSubscriptionKey
}

repositories {
    maven {
        url "http://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Great! You now know how to generate language files and provide automatic translations of your language keys.

98.3 Using Liferay's Language Settings

For a given locale, you can override Liferay's core UI messages. Modifying Language key values in Liferay provides a lot of localization flexibility in itself, but we're always looking for new ways to give you more control. There are language settings in Liferay's Language_xx.properties files that give you even more localization options.

- In the add and edit user forms, configure the name fields that are displayed and the field values available in select fields. For example, leave out the middle name field if you want, or alter the prefix selections.

- Control the directionality of content and messages in Liferay (left to right or right to left).

To see how these settings are configured, open Liferay's core `Language.properties` file in one of two ways:

1. From Liferay's source code, navigate to

   ```
   liferay-portal/portal-impl/src/content/Language.properties
   ```

2. From a Liferay bundle's `portal-impl.jar`.

   ```
   [Liferay Home]/tomcat-[version]/webapps/ROOT/WEB-INF/lib/portal-impl.jar
   ```

   Just open the `content` folder in the Jar to find the language files.

The first section in the `Language.properties` file is labeled *Language settings*:

```
##
## Language settings
##

lang.dir=ltr
lang.line.begin=left
lang.line.end=right
lang.user.name.field.names=prefix,first-name,middle-name,last-name,suffix
lang.user.name.prefix.values=Dr,Mr,Ms,Mrs
lang.user.name.required.field.names=last-name
lang.user.name.suffix.values=II,III,IV,Jr,Phd,Sr
```

Note: To use the language settings mentioned here, you need a module, which is like a magic carpet on which your code and resources ride triumphantly into Liferay's OSGi runtime. Refer to the tutorial on overriding language keys to set up a module with the following characteristics:

- Contains an implementation of `ResourceBundle` that is registered in the OSGi runtime.
- Contains a `Language.properties` file for the locale whose properties you want to override.

The user name properties are used to customize certain fields of the add and edit user forms based on a user's locale.

Localizing User Names

Liferay's customers come from all over the world, and we recognize that naming conventions are different between locales. Liferay's engineers have made several of the user name fields configurable in Liferay where user name information is entered or edited.

- Remove certain name fields and make others appear more than once. Some locale's need more than one last name, for example.

  ```
  lang.user.name.field.names=prefix,first-name,middle-name,last-name,suffix
  ```

- Change the prefix and suffix values for a locale.

```
lang.user.name.prefix.values=Dr,Mr,Ms,Mrs
lang.user.name.suffix.values=II,III,IV,Jr,Phd,Sr
```

- Specify which fields are required.

```
lang.user.name.required.field.names=last-name
```

Note: A user's first name is mandatory in Liferay. Because of this, take these two points into consideration when configuring a locale's user name settings:

1. The `first-name` field can't be removed from the field names list.

   ```
   lang.user.name.field.names=prefix,first-name,middle-name,last-name,suffix
   ```

2. Because first name is required, it's always implicitly included in the *required field names* property:

   ```
   lang.user.name.required.field.names=last-name
   ```

   Therefore, any fields you enter here are *in addition to* the first name field. Last name is required by default, but you can disable it by deleting its value from the property:

   ```
   lang.user.name.required.field.names=
   ```

   In that case, only first name would be required.

The properties for changing user name settings are those that begin with `lang.user.name` in the language settings section of a locale's language properties file.

For most of the locales enabled by default the user name properties are specifically tailored to that location.

```
locales.enabled=ca_ES,zh_CN,nl_NL,en_US,fi_FI,fr_FR,de_DE,iw_IL,hu_HU,ja_JP,pt_BR,es_ES
```

For example, these are the English (`Language_en.properties`) properties for setting user name fields:

```
lang.user.name.field.names=prefix,first-name,middle-name,last-name,suffix
lang.user.name.prefix.values=Dr,Mr,Ms,Mrs
lang.user.name.required.field.names=last-name
lang.user.name.suffix.values=II,III,IV,Jr,Phd,Sr
```

Compare those to the Spanish (`Language_es.properties`) settings:

```
lang.user.name.field.names=prefix,first-name,last-name
lang.user.name.prefix.values=Sr,Sra,Sta,Dr,Dra
lang.user.name.required.field.names=last-name
```

The biggest difference between the English and Spanish form fields in the images above is that the middle name and suffix fields are omitted in the Spanish configuration. Other differences include the specific prefix values.

¡Muy excelente! Localizing the forms for adding and editing users is accomplished using the same method by which Liferay's UI messages are localized: by overriding one of Liferay's `Lanuguage_xx.properties` files.

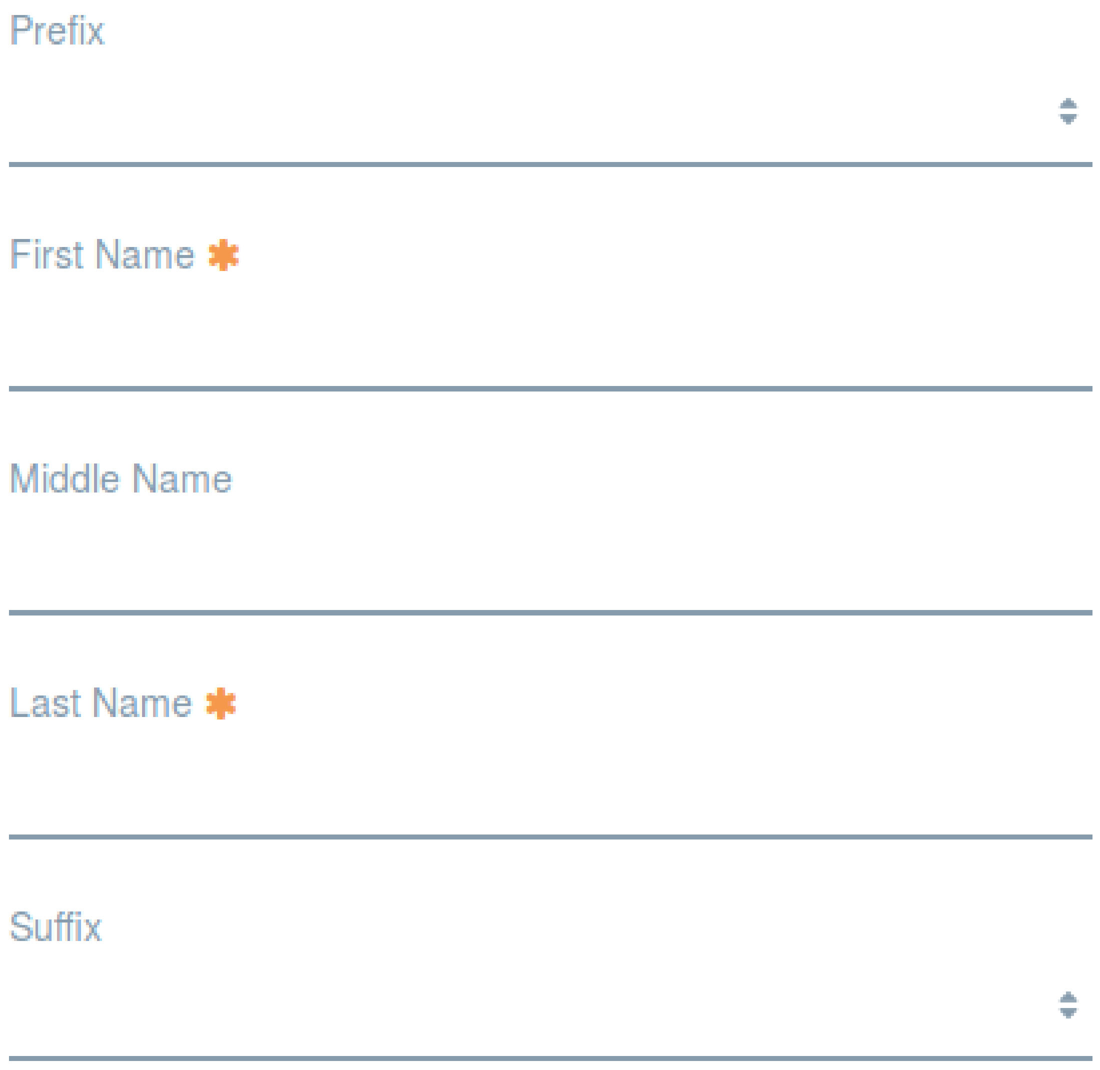


Figure 98.2: The user name settings impact the way user information and forms appear in Liferay.

Right to Left or Left to Right?

The first three properties in the language settings section are used for changing the direction in which the language's characters are displayed. Most languages are read from left to right, but other languages are meant to be read from right to left (Arabic, Hebrew, and Persian, for example). It can also be changed for languages that have been traditionally displayed left to right (like English) as a funny practical joke. Just don't tell anyone that you got the idea here.

Here's what the relevant language properties look like for a language that should be displayed from right to left:

```
lang.dir=rtl
lang.line.begin=right
lang.line.end=left
```

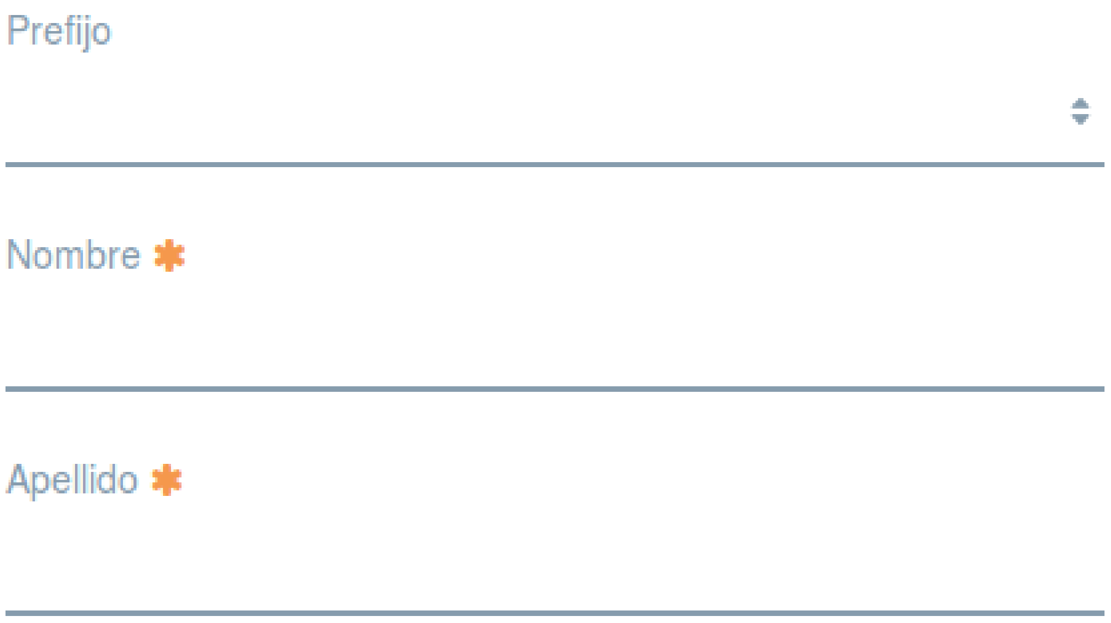


Figure 98.3: The Spanish user name settings omit the suffix and middle name fields entirely.

With these customizations yo can transform Liferay's UI into a user-friendly environment no matter where your users are from.

Chapter 99

WYSIWYG Editors

WYSIWYG editors are an important part of content creation. Liferay's platform supports several different editors, including CKEditor, TinyMCE, and our flagship, AlloyEditor. This section contains tutorials relating to WYSIWYG editors on the Liferay platform.

99.1 Adding a WYSIWYG Editor to a Portlet

It's easy to include WYSIWYG editors in your portlet, thanks to the `<liferay-ui:input-editor />` tag. Below is an example configuration:

```
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>

<div class="alloy-editor-container">
    <liferay-ui:input-editor
        contents="Default Content"
        cssClass="my-alloy-editor"
        editorName="alloyeditor"
        name="myAlloyEditor"
        placeholder="description"
        showSource="true" />
</div>
```

It is also possible to pass JavaScript functions through the `onBlurMethod`, `onChangeMethod`, `onFocusMethod`, and `onInitMethod` attributes. Here is an example configuration that uses the `onInitMethod` attribute to pass a JavaScript function called `OnDescriptionEditorInit`:

```
<%@ taglib prefix="aui" uri="http://liferay.com/tld/aui" %>
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>

<div class="alloy-editor-container">
    <liferay-ui:input-editor
        contents="Default Content"
        cssClass="my-alloy-editor"
        editorName="alloyeditor"
        name="myAlloyEditor"
        onInitMethod="OnDescriptionEditorInit"
        placeholder="description"
        showSource="true" />
</div>

<aui:script>
```

```
    function <portlet:namespace />OnDescriptionEditorInit() {
        <c:if test="<%= !customAbstract %>">
            document.getElementById('<portlet:namespace />myAlloyEditor').setAttribute('contenteditable', false);
        </c:if>
    }
</aui:script>
```

Below is an overview of the main attributes of the `<liferay-ui:input-editor />` tag:

Attribute	Type	Description
autoCreate	`java.lang.String`	Whether to show the HTML edit view of the editor initially
contents	`java.lang.String`	Sets the initial contents of the editor
contentsLanguageId	`java.lang.String`	Sets the language ID for the input editor's text
cssClass	`java.lang.String`	A CSS class for styling the component.
data	`java.util.Map`	Data that can be used as the editorConfig
editorName	`java.lang.String`	The editor you want to use (alloyeditor, ckeditor, tinymce, simple)
name	`java.lang.String`	A name for the input editor. The default value is `editor`.
onBlurMethod	`java.lang.String`	A function to be called when the input editor loses focus.
onChangeMethod	`java.lang.String`	A function to be called on a change in the input editor.
onFocusMethod	`java.lang.String`	A function to be called when the input editor gets focus.
onInitMethod	`java.lang.String`	A function to be called when the input editor initializes.
placeholder	`java.lang.String`	Placeholder text to display in the input editor.
showSource	`java.lang.String`	Whether to enable editing the HTML source code of the content. The default value is `true`.

See the taglibdocs for the complete list of supported attributes.

As you can see, it's easy to include WYSIWYG editors in your portlets!

Related Topics

Adding New Behavior to an Editor

Modifying an Editor's Configuration

Using the Liferay UI Taglib

99.2 Modifying an Editor's Configuration

Liferay DXP supports many different kinds of WYSIWYG editors that can be used in portlets to edit content. Depending on the content you're editing, you may want to modify the editor to provide a better configuration for your needs. In this tutorial, you'll learn how to extend your Liferay supported WYSIWYG editor to add new or modify existing configurations exactly how you'd like.

Extending the Editor's Configuration

To modify the editor's configuration, create a module that has a component that implements the `EditorConfigContributor` interface. When you implement this interface, your module will provide a service that modifies the editors you'd like to change. A simple example of this is provided below.

1. Create an OSGi module.

2. Create a unique package name in the module's `src` directory, and create a new Java class in that package. The class should extend the BaseEditorConfigContributor class.

3. Directly above the class's declaration, insert a component annotation:

```
@Component(
    property = {

    },

    service = EditorConfigContributor.class
)
```

   This annotation declares the implementation class of the Component and specifies the Component's properties. You should implement the `EditorConfigContributor` interface for this scenario. The `property` element is blank in the code snippet above. You need to insert properties that distinguish the editor's name, editor's configuration key, and/or the portlet name where the editor resides. These three properties can be specified independently or in any variation with each other. You can find out more about the available properties and how they should be used by reading the Javadoc provided in the EditorConfigContributor interface.

   The following code is a sample of what the `@Component` annotation could look like when modifying an editor's configuration:

```
@Component(
    property = {
        "editor.config.key=contentEditor", "editor.name=alloyeditor",
        "javax.portlet.name=com_liferay_blogs_web_portlet_BlogsPortlet",
        "javax.portlet.name=com_liferay_blogs_web_portlet_BlogsAdminPortlet",
        "service.ranking:Integer=100"
    },

    service = EditorConfigContributor.class
)
```

   This annotation declares that the following service is applied for the AlloyEditor identified by the `contentEditor` configuration key.

~~~~
**Note:** If you're targeting all editors for a portlet, the
`editor.config.key` is not required. For example, if you just want to target
the Web Content portlet's editors, you can provide the configuration below:

    @Component(
      property = {"editor.name=ckeditor",
      "javax.portlet.name=com_liferay_journal_web_portlet_JournalPortlet",
      "service.ranking:Integer=100"
    }
~~~~

~~~~
Two portlet names are specified (Blogs and Blogs Admin), which means the
service applies to all editors in those portlets. Lastly, the service
ranking is listed, which prioritizes this service over others that are
currently deployed in Liferay DXP.
~~~~

~~~~
**NOTE:** If you want to create a global configuration that applies to an
editor everywhere it's used, you must create two separate configurations:
one configuration that targets just the editor and a second configuration
that targets the Blogs and Blogs Admin portlets. For example, the two
separate configurations below apply the updates to AlloyEditor everywhere
it's used:

Configuration one:

```java
@Component(
    immediate = true,
    property = {
        "editor.name=alloyeditor",
        "service.ranking:Integer=100"
    },

    service = EditorConfigContributor.class
)
```

Configuration two:

```java
@Component(
    immediate = true,
    property = {
        "editor.name=alloyeditor",
        "javax.portlet.name=com_liferay_blogs_web_portlet_BlogsPortlet",
        "javax.portlet.name=com_liferay_blogs_web_portlet_BlogsAdminPortlet",
        "service.ranking:Integer=100"
    },

    service = EditorConfigContributor.class
)
```
~~~~

---

4. Now that you've specified which editor configurations you want to modify, you must specify what about them must change. Add the following method to your new class:

```
@Override
public void populateConfigJSONObject(
    JSONObject jsonObject, Map<String, Object> inputEditorTaglibAttributes,
```

```
        ThemeDisplay themeDisplay,
        RequestBackedPortletURLFactory requestBackedPortletURLFactory) {

    }
```

This method updates the original configuration JSON object with a new configuration. It can even update or delete the original configuration, or any other configuration introduced by another `EditorConfigContributor`. The configuration object contains the configuration to be directly used by the editor. This means that the configuration object used for this editor may differ from other editors used in Liferay DXP.

Currently, this method does nothing. You need to add some logic, which you'll do next.

5. In the `populateConfigJSONObject` method, you need to instantiate a JSONObject that holds the current configuration of the editor. For instance, you could do something like this:

   ```
   JSONObject toolbars = jsonObject.getJSONObject("toolbars");
   ```

   This gets the editor's toolbar.

```
**Note:** This toolbar configuration is only applicable for the AlloyEditor.
If you choose a configuration that is supported by multiple editors, you
could apply it to them all. To do this, you could specify all the editors
(e.g., `"editor.name=alloyeditor"`, `"editor.name=ckeditor"`,
`ckeditor_bbcode` etc.) in the `@Component` annotation  of your
`EditorConfigContributor` implementation, as you did in step 3. Use the
site links provided at the bottom of this tutorial to view each editor's
configuration options and requirements.
```

```
Now that you've retrieved the toolbar, you can modify it. You'll do this
next.
```

6. You'll modify the editor's toolbar by adding a camera button. To complete this, extract the *Add* buttons out of your toolbar configuration object as a JSONArray, and then add the button to that `JSONArray`. The following code adds a *Camera* button to the editor's toolbar:

   ```
   if (toolbars != null) {
       JSONObject toolbarAdd = toolbars.getJSONObject("add");

       if (toolbarAdd != null) {
           JSONArray addButtons = toolbarAdd.getJSONArray("buttons");

           addButtons.put("camera");
       }
   }
   ```

   The configuration JSON object is passed to the editor with the modifications you've implemented in the `populateConfigJSONObject` method.

Your Java class is complete! The only thing left to do is generate the module's JAR file and copy it to your Portal's `deploy` folder. Once the module is installed and activated in your Portal's service registry, your new editor configuration is available for use.

Liferay DXP supports several different types of WYSIWYG editors, which include (among others):

- AlloyEditor
- CKEditor
- TinyMCE

Make sure to visit each editor's configuration API to learn what each editor offers for configuration settings.

Related Topics

Adding New Behavior to an Editor
Embedding Portlets in Themes
Developing Portlets

99.3 Adding New Behavior to an Editor

With the support of several kinds of WYSIWYG editors, Liferay gives you many options to support your users' editing needs. Sometimes, however, you can't get what you want with configuration alone. To help developers in these situations, Liferay provides a way to programmatically access the editor instance to create the editor experience you want.

This can be done by using the `liferay-util:dynamic-include` JavaScript extension point. This allows anyone to inject JavaScript code right after the editor instantiation to configure/change the editor.

Note: By default, the CKEditor strips empty <i> tags, such as those used for Font Awesome icons, from published content, when switching between the Code View and the Source View of the editor. You can disable this behavior by using the `ckeditor#onEditorCreate` or `alloyeditor#onEditorCreate` extension points to add the following code to the editor:

```
CKEDITOR.dtd.$removeEmpty.i = 0
```

In this tutorial, you'll learn how to use the JavaScript extension point in your Liferay supported WYSIWYG editor.

Injecting JavaScript into a WYSIWYG Editor

The `liferay-util:dynamic-include` extension point is available in the JSP files of Liferay DXP's configurable editors. This extension point serves as the gateway for injecting JavaScript into your editor instance. To take advantage of this extension point, you should follow these steps:

1. Create a JS file with the JavaScript code you'd like to execute in your editor. Create the JS file in a folder that makes sense to reference, since you'll need to register the file in your module. Also remember that the extension point is configured to inject the JavaScript code into the editor immediately following editor initialization.

   Some examples of JS files that are injected into the CKEditor are creole_dialog_definition.js, creole_dialog_show.js, and dialog_definition.js. These JS files are used by Liferay DXP to redefine which fields show in different dialogs, depending on what the selected language (HTML, BBCode, Creole) supports. For example, Creole doesn't support background color in table cells, so the table cells are removed from the options displayed to the user when running in Creole mode.

2. Create a module that can register your new JS file and inject it into your editor instance. The module should have a structure similar to this:

    - `bnd.bnd`
    - `build.gradle`
    - `gradle/`
        - `wrapper/`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`
    - `gradlew`
    - `gradlew.bat`
    - `src/main/`
        - `java/com/liferay/editor/myeditormodule/`
            - `constants/`
                - `MyEditorModulePortletKeys.java`
            - `internal/`
                - `CKEditorOnDialogDefinitionCreateDynamicInclude.java`
        - `resources/`
            - `META-INF/resources/ckeditor/extension/`
                - `ckeditor_dialog_definition.js`
            - `content/`
                - `Language.properties`

3. Create a unique package name in the module's src directory, and create a new Java class in that package. To follow naming conventions, your class name should begin with the editor you're modifying, followed by custom attributes, and ending with *DynamicInclude* (e.g., `CKEditorCreoleOnEditorCreateDynamicInclude.java`). Your Java class should implement the DynamicInclude interface.

4. Directly above the class's declaration, insert the following code:

    ```
    @Component(immediate = true, service = DynamicInclude.class)
    ```

    This annotation declares the implementation class of the Component, and specifies to immediately start the module once deployed to Portal.

5. If you have not yet inherited the abstract methods from `DynamicInclude`, do that now. You'll have two implemented methods to edit: `include(...)` and `register(...)`.

6. In the `include(...)` method, retrieve the bundle where your custom JS file resides. Then retrieve the JS file as a URL and inject the contents into the editor. You can view some example code below that does this for the `creole_dialog_definition.js` file:

   ```
   Bundle bundle = _bundleContext.getBundle();

   URL entryURL = bundle.getEntry(
       "/META-INF/resources/html/editors/ckeditor/extension" +
           "/creole_dialog_definition.js");

   StreamUtil.transfer(entryURL.openStream(), response.getOutputStream());
   ```

   In the `include(...)` method, you can also retrieve editor configurations and choose what JS file to inject based on the configuration selected by the user. For example, this would be applicable for the use case that was suggested previously dealing with Creole's deficiency with displaying background colors in table cells. You can look at how this could be done by looking at the `include(...)` method in the CKEditorCreoleOnEditorCreateDynamicInclude class.

7. Make sure you've instantiated your bundle's context so you can successfully retrieve your bundle. As a best practice, do this by creating an activation method and then setting the `BundleContext` as a private field. Here's an example:

   ```
   @Activate
   protected void activate(BundleContext bundleContext) {
       _bundleContext = bundleContext;
   }

   private BundleContext _bundleContext;
   ```

   This method uses the `@Activate` annotation, which specifies that it should be invoked once the service component has satisfied its requirements. For this default example, the `_bundleContext` was used in the `include(...)` method.

8. Now register the editor you're customizing. For example, if you were injecting JS code into the CKEditor's JSP file, the code would look like this:

   ```
   dynamicIncludeRegistry.register(
       "com.liferay.frontend.editor.ckeditor.web#ckeditor#onEditorCreate");
   ```

   This registers the CKEditor into the Dynamic Include registry and specifies that JS code will be injected into the editor once it's created.

   Just as you can configure individual JSP pages to use a specific implementation of the available WYSIWYG editors, you can use those same implementation options for the registration process. Visit the Editors section of `portal.properties` for more details. For example, to configure the Creole implementation of the CKEditor, you could use the following key:

   ```
   "com.liferay.frontend.editor.ckeditor.web#ckeditor_creole#onEditorCreate"
   ```

That's it! The JS code that you created is now injected into the editor instance you've specified. You're now able to use JavaScript to add new behavior to your Liferay supported WYSIWYG editor!

Related Topics

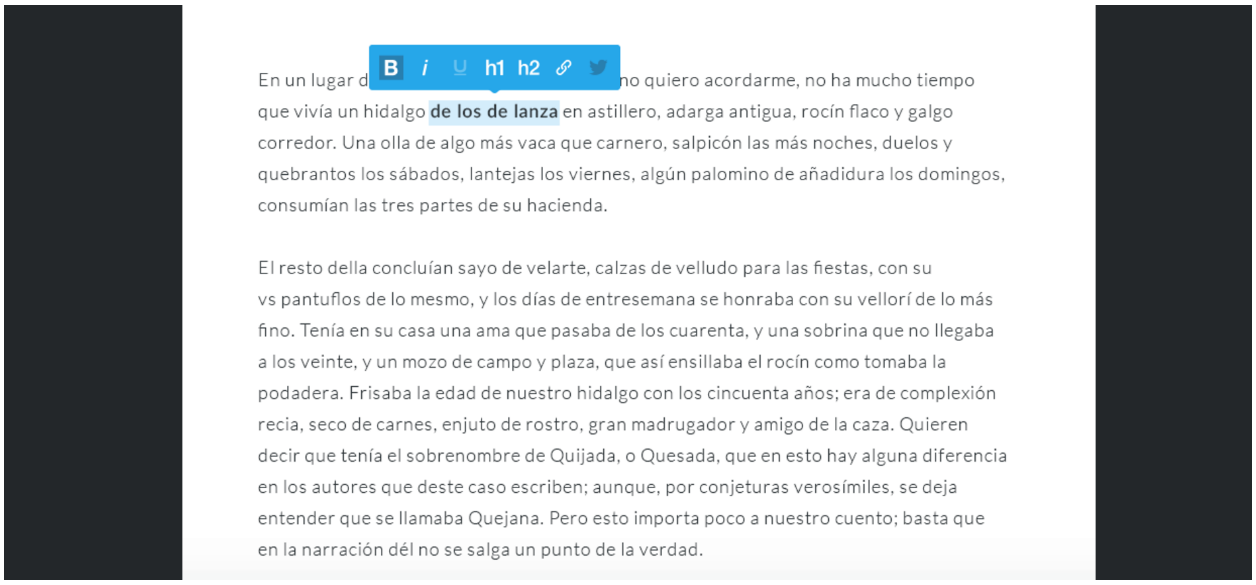
Modifying an Editor's Configuration
Embedding Portlets in Themes
Portlets

CHAPTER 100

AlloyEditor

AlloyEditor is a modern WYSIWYG editor built on top of CKEDITOR, designed to create modern and gorgeous web content.

As of Liferay DXP, AlloyEditor is the default WYSIWYG editor in Liferay DXP. Writing content is now enjoyable, and it has never been so easy!



En un lugar d no quiero acordarme, no ha mucho tiempo que vivía un hidalgo **de los de lanza** en astillero, adarga antigua, rocín flaco y galgo corredor. Una olla de algo más vaca que carnero, salpicón las más noches, duelos y quebrantos los sábados, lantejas los viernes, algún palomino de añadidura los domingos, consumían las tres partes de su hacienda.

El resto della concluían sayo de velarte, calzas de velludo para las fiestas, con su vs pantuflos de lo mesmo, y los días de entresemana se honraba con su vellorí de lo más fino. Tenía en su casa una ama que pasaba de los cuarenta, y una sobrina que no llegaba a los veinte, y un mozo de campo y plaza, que así ensillaba el rocín como tomaba la podadera. Frisaba la edad de nuestro hidalgo con los cincuenta años; era de complexión recia, seco de carnes, enjuto de rostro, gran madrugador y amigo de la caza. Quieren decir que tenía el sobrenombre de Quijada, o Quesada, que en esto hay alguna diferencia en los autores que deste caso escriben; aunque, por conjeturas verosímiles, se deja entender que se llamaba Quejana. Pero esto importa poco a nuestro cuento; basta que en la narración dél no se salga un punto de la verdad.

Figure 100.1: AlloyEditor is the new WYSIWYG editor by default, built on top of CKEditor.

100.1 Creating and Contributing new Buttons to AlloyEditor

It is possible to add additional AlloyEditor functionality through OSGi bundles. This tutorial demonstrates how to add a button to the editor.

Note: To use the syntax covered here, you must have AlloyEditor 2.11.0 or higher, which is included in Liferay DXP 7.0 Fix Pack 90 and Service Pack 13.

In this tutorial, you will learn how to

- Create an OSGi bundle for your own button
- Create a custom button for `AlloyEditor`
- Contribute your button to the list of available buttons
- Use your custom button in a toolbar in `AlloyEditor`

Go ahead and get started by creating the OSGi bundle next.

Creating the OSGi Bundle

AlloyEditor is built on `React.js` and uses `jsx` to render each button in the editor. Below is the folder structure for a module that adds a new button:

- `frontend-editor-my-button-web`
  - `src`
    - `main`
      - `java` - `com/liferay/frontend/editor/my/button/web/`
      - `editor`
      - `configuration`
      - `AlloyEditorMyButtonConfigContributor.java`
    - `servlet`
      - `taglib`
      - `AlloyEditorMyButtonDynamicInclude.java`
    - `resources`
      - `META-INF`
      - `resources`
      - `js`
      - `my_button.jsx`
  - `.babelrc` - needed since `JSX` is being compiled
  - `bnd.bnd`(example configuration shown below)

    Bundle-Name: Liferay Frontend Editor AlloyEditor My Button Web Bundle-SymbolicName: com.liferay.frontend.editor.alloyeditor.my.button.web Bundle-Version: 1.0.0 Liferay-Releng-Module-Group-Description: Liferay-Releng-Module-Group-Title: Rich Text Editors Web-ContextPath: /frontend-editor-alloyeditor-my-button-web

- `build.gradle`(contents shown below)

  configJSModules { enabled = false }

  dependencies { provided group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0" provided group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1" provided group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0" }

  transpileJS { bundleFileName = "js/buttons.js" globalName = "AlloyEditor.Buttons" modules = "globals" srcIncludes = "**/*.jsx" }

- `package.json`(contents shown below)

  { "devDependencies": { "babel-preset-react": "^6.11.1", "metal-cli": "^2.0.0" }, "name": "frontend-editor-alloyeditor-my-button-web", "version": "1.0.0" }

The contents of some of the files have been added as well, since the `build gradle` file requires some customizing.

Now that your OSGi bundle is configured, you can learn how to create buttons for the AlloyEditor next.

Creating the Button

Below is an example configuration for a JSX file that creates a new button:

```
/* global React, ReactDOM AlloyEditor */
(function() {
	'use strict';
	var React = AlloyEditor.React;
	var ButtonMyButton = React.createClass(
		{
			mixins: [AlloyEditor.Compat.ButtonStateClasses],
			displayName: 'ButtonMyButton',
			propTypes: {
				editor: React.PropTypes.object.isRequired
			},
			statics: {
				key: 'myButton'
			},
			/**
			 * Lifecycle. Renders the UI of the button.
			 *
			 * @method render
			 * @return {Object} The content which should be rendered.
			 */
			render: function() {
				var cssClass = 'ae-button ' + this.getStateClasses();
				return (
					<button className={cssClass}
					onClick={this._requestExclusive}
					tabIndex={this.props.tabIndex}>
						<small className="ae-icon small">
						Alt
						</small>
					</button>
				);
			},
			/**
			 * @protected
			 * @method  _doSomething
			 * @param {MouseEvent} event
			 */
			_doSomething: function(event) {
				console.log('do something!');
			}
```

```
                }
        );
        AlloyEditor.Buttons[ButtonMyButton.key] = AlloyEditor.ButtonMyButton
        = ButtonMyButton;
}());
```

The configuration above creates a new button called ButtonMyButton. The key aspects to note here are the lines that reference the global AlloyEditor. You can create your own JavaScript functions to interact with your button.

Now that you've seen how you can use a JSX file to create a new button, you can learn how to use your button in the editor next.

Contributing the Button

The next step is to add your button to the list of already available buttons. This can be achieved thanks to some smartly placed <liferay-util:dynamic-include /> tags in the editor's infrastructure. To make your button available in the AlloyEditor, you must extend the BaseDynamicInclude class. Below is an example configuration that extends this class:

```
package com.liferay.frontend.editor.alloyeditor.my.button.web.servlet.taglib;

import com.liferay.portal.kernel.servlet.taglib.BaseDynamicInclude;
import com.liferay.portal.kernel.servlet.taglib.DynamicInclude;
import com.liferay.portal.kernel.theme.ThemeDisplay;
import com.liferay.portal.kernel.util.PortalUtil;
import com.liferay.portal.kernel.util.StringBundler;
import com.liferay.portal.kernel.util.WebKeys;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

@Component(immediate = true, service = DynamicInclude.class)
public class AlloyEditorMyButtonDynamicInclude extends BaseDynamicInclude {

        @Override
        public void include(
                        HttpServletRequest request, HttpServletResponse response,
                        String key)
                throws IOException {

                ThemeDisplay themeDisplay = (ThemeDisplay)request.getAttribute(
                        WebKeys.THEME_DISPLAY);

                PrintWriter printWriter = response.getWriter();

                StringBundler sb = new StringBundler(7);

                sb.append("<script src=\"");
                sb.append(themeDisplay.getPortalURL());
                sb.append(PortalUtil.getPathProxy());
                sb.append(_servletContext.getContextPath());
                sb.append("/js/buttons.js");
                sb.append("\" ");
                sb.append("type=\"text/javascript\"></script>");

                printWriter.println(sb.toString());
```

```
	}

	@Override
	public void register(DynamicIncludeRegistry dynamicIncludeRegistry) {
		dynamicIncludeRegistry.register(
			"com.liferay.frontend.editor.alloyeditor.web#alloyeditor#" +
				"additionalResources");
	}

	@Reference(
		target = "(osgi.web.symbolicname=com.liferay.frontend.editor.alloyeditor.my.button.web)"
	)
	private ServletContext _servletContext;
}
```

Now that your button is included, you can learn how to make the button available in the editor's toolbar next.

Using the Button in a Toolbar

As explained in the Modifying an Editor's Configuration tutorial, you can configure which buttons show in the AlloyEditor toolbars by adding your own `EditorConfigContributor`. This file allows you to specify where in the toolbar your button should appear. The example configuration below doesn't specify a portlet name, so the button is added to the global AlloyEditor.

```
package com.liferay.frontend.editor.alloyeditor.my.button.web.editor.configuration;

import com.liferay.portal.kernel.editor.configuration.BaseEditorConfigContributor;
import com.liferay.portal.kernel.editor.configuration.EditorConfigContributor;
import com.liferay.portal.kernel.json.JSONArray;
import com.liferay.portal.kernel.json.JSONFactoryUtil;
import com.liferay.portal.kernel.json.JSONObject;
import com.liferay.portal.kernel.portlet.RequestBackedPortletURLFactory;
import com.liferay.portal.kernel.theme.ThemeDisplay;

import java.util.Map;
import java.util.Objects;

import org.osgi.service.component.annotations.Component;

@Component(
	property = {"editor.name=alloyeditor", "service.ranking:Integer=1000"},
	service = EditorConfigContributor.class
)
public class AlloyEditorMyButtonConfigContributor
	extends BaseEditorConfigContributor {

	@Override
	public void populateConfigJSONObject(
		JSONObject jsonObject, Map<String, Object> inputEditorTaglibAttributes,
		ThemeDisplay themeDisplay,
		RequestBackedPortletURLFactory requestBackedPortletURLFactory) {

		JSONObject toolbarsJSONObject = jsonObject.getJSONObject("toolbars");

		if (toolbarsJSONObject == null) {
			toolbarsJSONObject = JSONFactoryUtil.createJSONObject();
		}

		JSONObject stylesJSONObject = toolbarsJSONObject.getJSONObject(
			"styles");

		if (stylesJSONObject == null) {
			stylesJSONObject = JSONFactoryUtil.createJSONObject();
```

```
                }

                JSONArray selectionsJSONArray = stylesJSONObject.getJSONArray(
                        "selections");

                for (int i = 0; i < selectionsJSONArray.length(); i++) {
                        JSONObject selection = selectionsJSONArray.getJSONObject(i);

                        if (Objects.equals(selection.get("name"), "text")) {
                                JSONArray buttons = selection.getJSONArray("buttons");

                                buttons.put("myButton");
                        }
                }

                stylesJSONObject.put("selections", selectionsJSONArray);

                toolbarsJSONObject.put("styles", stylesJSONObject);

                jsonObject.put("toolbars", toolbarsJSONObject);
        }
}
```

There you have it. Now you know how to create and use custom buttons in the AlloyEditor!

Related Topics

Adding a WYSIWYG Editor to a Portlet

Modifying an Editor's Configuration

100.2 Using the Default CKEditor Plugins Bundled with AlloyEditor

You can customize an editor's configuration to include several modifications, such as adding new buttons and adding new behaviors. You can also use existing CKEditor plugins in AlloyEditor. Several CKEditor plugins are packaged with Liferay DXP's AlloyEditor, so you can use them with just a few configuration adjustments. This tutorial shows how to use the CKEditor plugins bundled with Liferay DXP's AlloyEditor. The `com.liferay.docs.myblogseditorconfigcontributor` module is used as an example throughout this tutorial.

Follow these steps:

1. Create a module to modify the AlloyEditor's configuration. The example boilerplate below modifies the AlloyEditor's configuration for the Blogs and Blogs Admin portlets:

```
@Component(
    immediate = true,
    property = {
        "editor.config.key=contentEditor",
        "editor.name=alloyeditor",
        "editor.name=ckeditor",
        "javax.portlet.name=com_liferay_blogs_web_portlet_BlogsPortlet",
        "javax.portlet.name=com_liferay_blogs_web_portlet_BlogsAdminPortlet",
        "service.ranking:Integer=100"
    },
    service = EditorConfigContributor.class
)
public class MyBlogsEditorConfigContributor
  extends BaseEditorConfigContributor {

    @Override
    public void populateConfigJSONObject(
      JSONObject jsonObject, Map<String, Object> inputEditorTaglibAttributes,
      ThemeDisplay themeDisplay,
      RequestBackedPortletURLFactory requestBackedPortletURLFactory) {

    }
  }
```

2. Add additional plugins to the AlloyEditor via the `extraPlugins` JSON object. To add a CKEditor plugin, extract the current list of `extraPlugins` from your editor configuration object as a `String`:

   ```
   String extraPlugins = jsonObject.getString("extraPlugins");
   ```

3. Choose the plugin(s) you want to use from the default CKEditor plugins bundled with Liferay DXP's AlloyEditor.

4. Add the CKEditor plugin(s) you want to use to the `extraPlugins` configuration. Liferay DXP's AlloyEditor also comes with several plugins to bridge the gap between the CKEditor's UI and the AlloyEditor's UI. These are prefixed with `ae_`. We recommend that you include them all to ensure compatibility. The example below checks for existing `extraPlugins` and adds the `font` CKEditor plugin along with its required Rich Combo plugin dependency and the remaining UI bridge plugins:

   ```
   if (Validator.isNotNull(extraPlugins)) {
     extraPlugins = extraPlugins + ",ae_uibridge,ae_autolink,ae_buttonbridge,ae_menubridge,ae_panelmenubuttonbridge,ae_placeholder,ae_richcombobridge,f
   }
   else {
     extraPlugins = "ae_uibridge,ae_autolink,ae_buttonbridge,ae_menubridge,ae_panelmenubuttonbridge,ae_placeholder,ae_richcombobridge,font";
   }

   jsonObject.put("extraPlugins", extraPlugins);
   ```

```
**Note:** Make sure the `ae_uibridge` plugin appears first, followed by
the remaining UI bridge plugins, and finally the CKEditor plugin(s).
```

---

5. If the plugin includes buttons, add them in the appropriate toolbar. The configuration below retrieves the Text Selection Toolbar's buttons and adds the `font` plugin's `Font` and `FontSize` buttons to it:

   ```
   JSONObject toolbarsJSONObject = jsonObject.getJSONObject("toolbars");

   if (toolbarsJSONObject == null) {
    toolbarsJSONObject = JSONFactoryUtil.createJSONObject();
   }

   JSONObject stylesJSONObject = toolbarsJSONObject.getJSONObject(
    "styles");

   if (stylesJSONObject == null) {
    stylesJSONObject = JSONFactoryUtil.createJSONObject();
   }

   JSONArray selectionsJSONArray = stylesJSONObject.getJSONArray(
    "selections");

   for (int i = 0; i < selectionsJSONArray.length(); i++) {
    JSONObject selection = selectionsJSONArray.getJSONObject(i);
   ```

```
            if (Objects.equals(selection.get("name"), "text")) {
             JSONArray buttons = selection.getJSONArray("buttons");

             buttons.put("Font");
             buttons.put("FontSize");
            }

        }

        stylesJSONObject.put("selections", selectionsJSONArray);

        toolbarsJSONObject.put("styles", stylesJSONObject);

        jsonObject.put("toolbars", toolbarsJSONObject);
```

Note: A plugin's buttons may not have the same name as the plugin. You can find the button names for a plugin by searching its `plugin.js` file for `editor.ui.addButton`. Note that button names are case sensitive and may be aliased in the `addButton()` method, such as the `clipboard` plugin's `Cut`, `Copy`, and `Paste` buttons.

Below is the full example `*EditorConfigContributor` class that adds the `font` plugin to the AlloyEditor for the Blogs and Blogs Admin portlets:

```
package com.liferay.docs.myblogseditorconfigcontributor;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Objects;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;


import com.liferay.portal.kernel.editor.configuration.BaseEditorConfigContributor;
import com.liferay.portal.kernel.editor.configuration.EditorConfigContributor;
import com.liferay.portal.kernel.json.JSONArray;
import com.liferay.portal.kernel.json.JSONFactoryUtil;
import com.liferay.portal.kernel.json.JSONObject;
import com.liferay.portal.kernel.portlet.RequestBackedPortletURLFactory;
import com.liferay.portal.kernel.theme.ThemeDisplay;
import com.liferay.portal.kernel.util.Portal;
import com.liferay.portal.kernel.util.Validator;

/**
 * @author liferay
 */
 @Component(
    immediate = true,
    property = {
        "editor.config.key=contentEditor",
        "editor.name=alloyeditor",
        "editor.name=ckeditor",
        "javax.portlet.name=com_liferay_blogs_web_portlet_BlogsPortlet",
        "javax.portlet.name=com_liferay_blogs_web_portlet_BlogsAdminPortlet",
        "service.ranking:Integer=100"
    },
    service = EditorConfigContributor.class
 )
 public class MyBlogsEditorConfigContributor
   extends BaseEditorConfigContributor {

   @Override
   public void populateConfigJSONObject(
     JSONObject jsonObject, Map<String, Object> inputEditorTaglibAttributes,
     ThemeDisplay themeDisplay,
     RequestBackedPortletURLFactory requestBackedPortletURLFactory) {

            String extraPlugins = jsonObject.getString("extraPlugins");

             if (Validator.isNotNull(extraPlugins)) {
                  extraPlugins = extraPlugins + ",ae_uibridge,ae_autolink,ae_buttonbridge,ae_menubridge,ae_panelmenubuttonbridge,ae_placeholder,ae_richcombo
                }
                else {
                  extraPlugins = "ae_uibridge,ae_autolink,ae_buttonbridge,ae_menubridge,ae_panelmenubuttonbridge,ae_placeholder,ae_richcombobridge,font";
                }

                jsonObject.put("extraPlugins", extraPlugins);

                JSONObject toolbarsJSONObject = jsonObject.getJSONObject("toolbars");

                if (toolbarsJSONObject == null) {
                 toolbarsJSONObject = JSONFactoryUtil.createJSONObject();
                }

                JSONObject stylesJSONObject = toolbarsJSONObject.getJSONObject(
                 "styles");

                if (stylesJSONObject == null) {
                 stylesJSONObject = JSONFactoryUtil.createJSONObject();
                }

                JSONArray selectionsJSONArray = stylesJSONObject.getJSONArray(
                 "selections");

                for (int i = 0; i < selectionsJSONArray.length(); i++) {
                 JSONObject selection = selectionsJSONArray.getJSONObject(i);

                        if (Objects.equals(selection.get("name"), "text")) {
                         JSONArray buttons = selection.getJSONArray("buttons");

                         buttons.put("Font");
                         buttons.put("FontSize");
                        }

                 }

                stylesJSONObject.put("selections", selectionsJSONArray);

                toolbarsJSONObject.put("styles", stylesJSONObject);

                jsonObject.put("toolbars", toolbarsJSONObject);
   }
 }
```

Now you know how to use Liferay DXP's bundled CKEditor plugins in its AlloyEditor!

Related Topics

Modifying an Editor's Configuration
Creating and Contributing New Buttons to AlloyEditor
Adding New Behavior to an Editor

Chapter 101

JavaScript Module Loaders

JavaScript modules encapsulate code into useful units that export their functions. Structuring an application this way makes it easier to work with in these ways:

- Other modules can explicitly require this piece of code.
- Structuring an application this way makes it easier to see the broader scope.
- Modular applications keep related functionality close together.
- Modularized code makes it easier to find what you're looking for.

This section contains tutorials relating to the different JavaScript Module Loaders included present on the Liferay platform.

101.1 Configuring Modules for Liferay DXP's Loaders

To load your modules in Liferay DXP, you need to know when they are needed, where they are located at build time, if you want to bundle them together or load them independently, and you must assemble them at runtime. Keeping track of all these tasks can be a hassle. Liferay DXP's module Loaders (YUI Loader and AMD Loader) provide a streamlined process that handles loading for you, which saves you time.

ES2015 `*.es.js` files are automatically transpiled to AMD modules and configured, so no additional work is needed for the Loader to recognize them. Other JavaScript modules, however, require more information to use Liferay DXP's Loaders.

Manual configuration is required for the following use cases:

- Custom AUI and YUI modules
- External libraries with named AMD modules
- External libraries with global exports that you want to load asynchronously or from other modules
- Initialization code

This tutorial covers these concepts:

- How to configure JavaScript modules to use Liferay DXP's Loaders
- How to use your loaded JavaScript modules in a portlet or a JavaScript

See the Preparing your JavaScript Files for ES2015 tutorial to learn how to use ES2015 in your JavaScript modules.

Get started by configuring your module next.

Configuring your Module

To use the loaders you must first define your modules. This metadata, known as the *module definition*, provides details such as dependencies, name and location, when they should be loaded, and more.

The module is defined using a configuration file. Liferay DXP uses `config.js` as a naming convention, but you can use whatever name you prefer.

You must specify your configuration file's location in your bundle's `bnd.bnd` file, so Liferay DXP knows where to access it. You'll learn how to do this next.

Configuring your Bundle's BND File

Follow these steps to configure your BND file:

1. Open your bundle's `bnd.bnd` file and add the `Liferay-JS-Config` header to point to the configuration file that contains the module's definition.

   For example, the header below points to a `config.js` file in the module's bundle:

   ```
   Liferay-JS-Config: /META-INF/resources/config.js
   ```

2. Next, add a web context path to retrieve resources for your module:

   ```
   Web-ContextPath: /my-bundle-name
   ```

Now that Liferay DXP knows how to find the file, you can write it next.

Writing the Configuration File

Follow these steps to define your module:

1. Create a configuration file, for example `config.js`, in the location you specified above.

   For example:

   ```
   src/main/resources/META-INF/resources
   ```

2. Identify the loader your module requires.

   The type of module you are configuring determines the loader you must use in your configuration file.

   YUI and AlloyUI modules: Use the `YUI.applyConfig` mechanism to provide the module information. Note that AUI modules use the AUI mechanism built on top of the existing YUI mechanism: `AUI().applyConfig`. You can also use this mechanism to override Liferay DXP's default YUI/AUI modules.

   AMD or global libraries: Use the `Liferay.Loader.addModule` mechanism to provide the module information.

   Initialization code: requires no loader mechanism. Simply add your code to the module's configuration file.

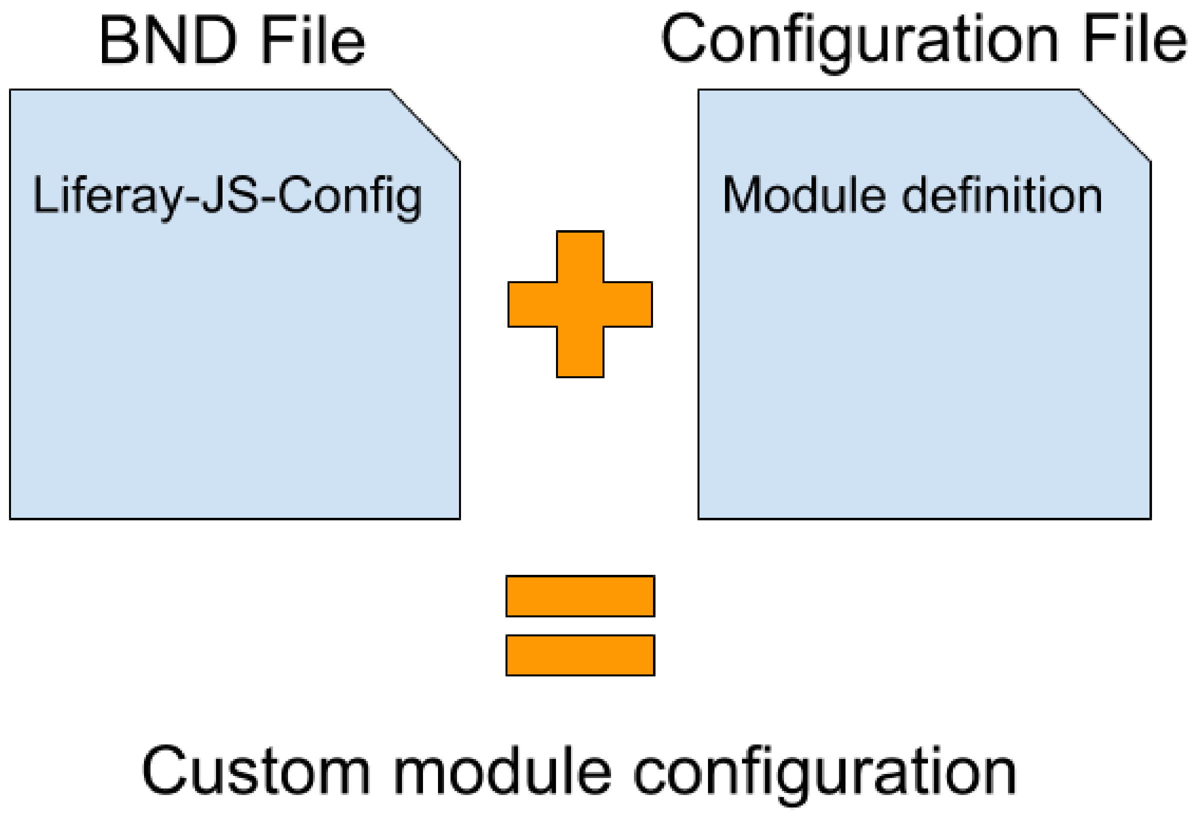


Figure 101.1: Custom JavaScript modules must use the `Liferay-JS-Config` BND header to point to a configuration file with the module definition.

3. Add the module's definition to the configuration file using the loader mechanism you identified in step 2.

   Below are some example configurations for each use case.

   Custom AUI module `config.js` example `com.liferay.map.common` module:

```
;(function() {
        AUI().applyConfig(
                {
                        groups: {
                                mapbase: {
                                        base: MODULE_PATH + '/js/',
                                        combine: Liferay.AUI.getCombine(),
                                        modules: {
                                                'liferay-map-common': {
                                                        path: 'map.js',
                                                        requires: [
                                                                'aui-base'
                                                        ]
                                                }
                                        },
                                        root: MODULE_PATH + '/js/'
                                }
                        }
                }
                );
        })();
```

Note: You can use the MODULE_PATH variable to reference your module's location in relative paths. This mechanism is more robust and reliable than hard coded paths in the event of modified system settings.

The parameters used in the AUI module example are defined below:

groups: A list of group definitions. Each group can contain definitions for base, comboBase, Combine, and a list of modules.

base: The base directory to fetch the module from.

combine: Whether to use a combo service to reduce the number of HTTP connections required to load your dependencies. Best practice is to use Liferay.AUI.getCombine() as the value, as Liferay DXP's own modules do. If js_fast_load in enabled in your theme, Liferay.AUI.getCombine() returns true, otherwise it returns false. Hard coding a value can result in odd or unexpected behavior, and is not recommended.

modules: A list of module definitions.

path: The path to the script from base. This parameter is required.

requires: An array of modules required by this component.

See the Loader.addModule method for a full list of the supported module metadata.

root: The root path to prepend to module names for the combo service. Ex: 2.5.2/build.

See the Loader class for a full list of available methods and properties.

Custom AMD module config.js example com.liferay.frontend.js.polyfill.babel.web module:

```
Liferay.Loader.addModule(
        {
                dependencies: [],
                exports: '_babelPolyfill',
                name: 'polyfill-babel',
                path: MODULE_PATH + 'browser-polyfill.min.js'
        }
);
```

The parameters used in the custom AMD module example are defined below:

dependencies: An array of module dependencies.

exports: The value, as a string, that the module exports to the global namespace. This is used for non-AMD modules. For example if your module exposes the global attribute window.MyLibrary, then you can set exports = 'MyLibrary' to let the loader know when this module is done loading.

name: The name of the module.

path: Sets the path of the module. If omitted, the module name value will be used as the path.

fullpath: Sets the full path to the module. This property should be used instead of the path property when the module isn't located in Portal. For example, you can use the fullpath property to load a library from an external CDN: fullPath: 'https://web/address/external-library.js'.

Library initialization code module config.js example com.liferay.frontend.js.metal.web module:

```
window.__METAL_COMPATIBILITY__ = {
        renderers: ['soy']
};
```

Although the example above contains library configuration code, you could add any initialization code that you require.

Liferay DXP automatically collects all the module definitions in a single request at startup, so you don't need to be concerned about the timing and placement of their configuration.

Now that your module is configured, you can learn how to use it in Liferay DXP next.

Using your Module

Once your module is configured, you have a few ways in which you can use it in Liferay DXP.

This example is configured to use a module in the JSP of a portlet, via the `aui:script`'s `require` attribute:

```
<aui:script require="relative/path/to/module/module-name">
  // variable `relativePathToModuleModuleName` is available here
</aui:script>
```

To adhere to JavaScript standards, references to the module within the script tag are named after the require value, in camel-case and with all invalid characters removed. For more information on using your module in a portlet, see the Using ES2015 Modules in your Portlet tutorial.

You can also use the module in a generic JavaScript:

```
<script>
Liferay.Loader.require('module-name', function (moduleName) {
  // variable `moduleName` is available here
});
</script>
```

Note: Using `Liferay.Loader.require` rather than just `require` is safer if you plan to hide the Loader by disabling the `exposeGlobal` option.

In Liferay DXP 7.1, `exposeGlobal` will be disabled by default.

Now you know how to load your custom JavaScript modules and global libraries in Liferay DXP!

Related Topics

Preparing your JavaScript Files for ES2015

Using ES2015 Modules in your Portlet

Overriding Liferay DXP's Default YUI and AUI Modules

101.2 Using External Libraries

You can use external (i.e., anything but Metal.js, jQuery, or Lodash, which are included in Liferay DXP) JavaScript libraries in your portlets. There are a few methods you can use to make external libraries available. The method you should choose depends on the external libraries you plan to use and how you plan to use them (as modules or as browser globals).

This tutorial covers how to adapt external libraries for Liferay's JavaScript Loaders.

Go ahead and get started.

Configuring Libraries to Support UMD

If you're the owner of the library, you should make sure that it supports UMD (Universal Module Definition). You can configure your code to support UMD with the template shown below:

```
// Assuming your "module" will be exported as "mylibrary"
(function (root, factory) {
    if (typeof Liferay.Loader.define === 'function' && Liferay.Loader.define.amd) {
        // AMD. Register as a "named" module.
        Liferay.Loader.define('mylibrary', [], factory);
    } else if (typeof module === 'object' && module.exports) {
        // Node. Does not work with strict CommonJS, but
        // only CommonJS-like environments that support module.exports,
        // like Node.
        module.exports = factory();
    } else {
        // Browser globals (root is window)
        root.mylibrary = factory();
  }
}(this, function () {

    // Your library code goes here
    return {};
}));
```

Next you can learn how to load external libraries as browser globals.

Loading Libraries as Browser Globals

If you want to use a library that doesn't export itself as a named module (as is the case for many plugins) or load the library as a browser global, follow the steps in this section.

Note: These steps only apply to users on Liferay Portal CE 7.0 GA4, Liferay Digital Enterprise 7.0 SP2 (Fix Pack 8), or lower patch levels. If you're on a higher patch level, follow the steps in the Using Libraries that You Host section.

Follow these steps to load your libraries as browser globals:

1. Add a `<script>` tag with the following content before loading your module:

   ```
   <script>
       Liferay.Loader.define._amd = Liferay.Loader.define.amd;
       Liferay.Loader.define.amd = false;
   </script>
   ```

2. Next, add a `<script>` tag to load the module itself. Below is an example configuration:

   ```
   <script type="text/javascript" src="${javascript_folder}/library.js">
   </script>
   ```

3. Finally, cancel the change made in the previous step, by adding the following `<script>` tag:

   ```
   <script>
       Liferay.Loader.define.amd = Liferay.Loader.define._amd;
   </script>
   ```

This approach lets you load your modules as browser globals. Next, you can learn how to load libraries that you host.

Using Libraries That You Host

If you're hosting the library (and not loading it from a CDN), you must hide the Liferay AMD Loader to use your Library.

If you're running Liferay Portal CE 7.0 GA4, Liferay Digital Enterprise 7.0 SP2 (Fix Pack 8), or a higher patch level, you can hide the Liferay AMD Loader through the control panel. Follow these steps:

1. Open the Control Panel, navigate to *Configuration* → *System Settings*.
2. Click *JavaScript Loader* under the *Foundation* tab.
3. Uncheck the `expose global` option.

Note: Once this option is unchecked, you can no longer use the `Liferay.Loader.define` or `Liferay.Loader.require` functions in your app. Also, if you're using third party libraries that are AMD compatible, they could stop working after unchecking this option because they usually use global functions like `require()` or `define()`.

If you're running a lower patch level than Liferay Portal CE 7.0 GA4 or Liferay Digital Enterprise 7.0 SP2 (Fix Pack 8), follow these steps to hide the Liferay AMD Loader:

1. Name the library in the define function, as covered in the Configuring Libraries to Support UMD section. Below is an example configuration:

   ```
   Liferay.Loader.define('mylibrary', [], factory);
   ```

2. Remove the UMD wrapper `if (typeof Liferay.Loader.define === 'function' && Liferay.Loader.define.amd)` or update the UMD wrapper to match the one below:

   ```
   if (false && typeof Liferay.Loader.define === 'function' && Liferay.Loader.define.amd)
   ```

3. Configure your bundle's build task to run the `configJSModules` task over the library.

   This task names the library and generates the appropriate loader configuration for you.

Now you know how to adapt external libraries for Liferay's JavaScript Loaders.

Related Topics

Configuring Modules for Liferay Portal's Loaders
Liferay AMD Module Loader
Using ES2015 Modules in Your Portlet

101.3 Liferay AMD Module Loader

The Liferay AMD Module Loader is a JavaScript module loader.

What is a JavaScript module?

A JavaScript module encapsulates a piece of code into a useful unit that exports its capability/value. This makes it easy for other modules to explicitly require this piece of code. Structuring an application this way makes it easier to see the broader scope, easier to find what you're looking for, and keeps related pieces close together. This way of coding is a specification for the JavaScript language called Asynchronous Module Definition, or AMD.

Purpose of Liferay AMD Module Loader

A normal web page usually loads JavaScript files via HTML `script` tags. That's fine for small websites, but when developing large scale web applications, a more robust organization and loader is needed. A module loader allows an application to load dependencies easily by specifying a string that identifies the module name.

Now that you know the purpose of the Liferay AMD Module Loader, you can learn how to define modules next.

Defining a Module

The Liferay AMD Module loader works with JavaScript modules that are in the AMD format. Here is a basic example of the definition of an AMD module:

```
Liferay.Loader.define('my-dialog', ['my-node', 'my-plugin-base'], function(myNode, myPluginBase) {
    return {
        log: function(text) {
            console.log('module my-dialog: ' + text);
        }
    };
});
```

You can specify to load the module when another module is triggered or when a given condition is met:

```
Liferay.Loader.define('my-dialog', ['my-node', 'my-plugin-base'], function(myNode, myPluginBase) {
    return {
        log: function(text) {
            console.log('module my-dialog: ' + text);
        }
    };
}, {
    condition: {
        trigger: 'my-test',
        test: function() {
            var el = document.createElement('input');

            return ('placeholder' in el);
        }
    },
    path: 'my-dialog.js'
});
```

The configuration above specifies that this module should be loaded automatically, if the developer requests the `my-test` module under the given condition.

Next you can learn how to load a module.

Loading a Module

Loading a module is as easy as passing the module name to the `Liferay.Loader.require` method. The example below loads a module called `my-dialog`:

```
Liferay.Loader.require('my-dialog', function(myDialog) {
    // your code here
}, function(error) {
    console.error(error);
});
```

Next you can learn how to map module names.

Mapping Module Names

You can map module names to specific versions or other naming conventions. The example below maps the `liferay` and `liferay2` modules to `liferay@1.0.0`:

```
__CONFIG__.maps = {
    'liferay': 'liferay@1.0.0',
    'liferay2': 'liferay@1.0.0'
};
```

Mapping a module changes its name to the value specified in the map. Take this require value for example:

```
Liferay.Loader.require('liferay/html/js/autocomplete'...)
```

Under the hood, this is the same as the value shown below:

```
Liferay.Loader.require('liferay@1.0.0/html/js/autocomplete'...)
```

Using Liferay AMD Module Loader in Liferay DXP

Tools, like the Liferay AMD Module Config Generator, have been integrated into Liferay DXP to make it easy for developers to create and load modules. Here's how it works:

1. The Module Config Generator scans your code and looks for AMD module `Liferay.Loader.define(...)` statements.

2. It then names the module if it is not named already.

3. It uses that information, along with the listed dependencies, as well as any other configurations specified, to create a `config.json` file. Below is an example of a generated `config.json` file:

```
{
    "frontend-js-web@1.0.0/html/js/parser": {
        "dependencies": []
    },
    "frontend-js-web@1.0.0/html/js/list-display": {
        "dependencies": ["exports"]
    },
    "frontend-js-web@1.0.0/html/js/autocomplete": {
        "dependencies": ["exports", "./parser", "./list-display"]
    }
}
```

This configuration object tells the loader which modules are available, where they are, and what dependencies they require.

Now you know all about the Liferay AMD Module Loader!

Related Topics

Configuring Modules for Liferay Portal's Loaders

101.4 Loading Modules with AUI Script in Liferay DXP

The `aui:script` tag is a JSP tag that loads JavaScript in script tags on the page, while ensuring that certain resources are loaded before executing.

Using aui:script

The `aui:script` tag supports the following options:

- `require`: Requires an AMD module that will be loaded with the Liferay AMD Module Loader.
- `use`: Uses an AlloyUI/YUI module that is loaded via the YUI loader.
- `position`: The position the script tag is put on the page. Possible options are `inline` or `auto`.
- `sandbox`: Whether to wrap the script tag in an anonymous function. If set to `true`, in addition to the wrapping, `$` and `_` are defined for jQuery and underscore.

Next you can learn how to load ES2015 and Metal.js modules.

Loading ES2015 and Metal.js Modules

You can use `aui:script` to load your ES2015 and Metal.js modules like this:

```
<aui:script require="metal-clipboard/src/Clipboard">
    new metalClipboardSrcClipboard.default();
</aui:script>
```

This resolves the dependencies of the registered `Clipboard.js` and loads them in order until all of them are satisfied and the requested module can be safely executed.

In the browser, the `aui:script` translates to the full HTML shown below:

```
<script type="text/javascript">
    Liferay.Loader.require("metal-clipboard/src/Clipboard", function(metalClipboardSrcClipboard) {
        (function() {
            new metalClipboardSrcClipboard.default();
        })()
    }, function(error) {
        console.error(error)
    });
</script>
```

Next you can learn how to load AlloyUI modules.

Loading AlloyUI Modules

You can use the `use` attribute to load AlloyUI/YUI modules:

```
<aui:script use="aui-base">
    A.one('#someNodeId').on(
        'click',
        function(event) {
            alert('Thank you for clicking.')
        }
    );
</aui:script>
```

This loads the `aui-base` AlloyUI component and makes it available to the code inside the `aui:script`.

In the browser, the `aui:script` translates to the full HTML shown below:

```
<script type="text/javascript">
    AUI().use("aui-base",
        function(A){
            A.one('#someNodeId').on(
                'click',
                function(event) {
                    alert('Thank you for clicking.')
                }
            );
        }
    );
</script>
```

Next you can learn how to load AlloyUI modules together with ES2015 and Metal.js modules.

Loading AlloyUI Modules and ES2015 and Metal.js Modules Together

You may want to load an AUI module along with an ES2015 module or Metal.js module in an `aui:script`. The `aui:script` tag doesn't support both the `require` and `use` attributes in the same configuration. Not to worry though. You can use the `aui:script`'s `require` attribute to load the ES2015 and Metal.js modules, while loading the AUI module(s) with the `AUI().use()` function within the script. Below is an example configuration:

```
<aui:script require="path-to/metal/module">
 AUI().use(
    'liferay-aui-module',
    function(A) {
        let var = pathToMetalModule.default;
    }
);
</aui:script>
```

Now you know how to load modules with the `aui:script` tag!

Related Topics

Configuring Modules for Liferay Portal Loaders

Chapter 102

Liferay JavaScript APIs

The Liferay JavaScript object is populated with some helpful tools. This section contains a comprehensive list of some of the most useful utilities you can find inside the Liferay object.

102.1 Liferay ThemeDisplay

In Java, developers are used to being able to find lots of context information at runtime. You can learn about what user is browsing your application, what page it's on, what site it's in, and lots more. Wouldn't it be great if you could access that same information in JavaScript? You can! You can use Liferay DXP's ThemeDisplay JavaScript object!

It's a part of the Liferay global object that's automatically available to you in Liferay DXP at runtime. You can refer to the object as Liferay.ThemeDisplay. The ThemeDisplay object provides information on many aspects of a portal. It can identify the portal instance, the current user, the user's language, and the user's navigational context. It can tell you the paths to a portlet's scripts and images, a theme's images and files, and a portal's main folder. And it lets you know if a user is signed in and if the user is being impersonated. You can quickly assess your portal surroundings with ThemeDisplay.

This tutorial describes some of the most commonly used ThemeDisplay methods for getting IDs, paths, and user sign-in details.

Retrieving IDs

Using the ThemeDisplay methods below, you can grab IDs of various portal elements:

getCompanyId: Returns the company ID.

getLanguageId: Returns the user's language ID.

getScopeGroupId: Returns the group ID of the current site.

getUserId: Returns the user's ID.

getUserName: Returns the user's name.

Now that you know how to retrieve IDs of some of Liferay's key elements, you can learn how to get paths to various deployed entities in the portal.

Retrieving File Paths

The ThemeDisplay object has methods for retrieving commonly used file paths. Below are a few of the methods:

getPathImage: Returns the relative path of the portlet's image directory.

getPathJavaScript: Returns the relative path of the directory containing the portlet's JavaScript source files.

getPathMain: Returns the path of the portal instance's main directory.

getPathThemeImages: Returns the path of the current theme's image directory.

getPathThemeRoot: Returns the relative path of the current theme's root directory.

Now that you know how to retrieve paths to Liferay's deployed entities, you can next learn how to get information about the current user.

Retrieving Login Information

Here are a couple methods related to the current user.

isImpersonated: Returns `true` if the current user is being impersonated. Authorized administrative users can impersonate act as another user to test that user's account.

isSignedIn: Returns `true` if the user is logged in to the portal.

Below is JavaScript code that demonstrates using `ThemeDisplay`'s `isSignedIn` method:

```
if(Liferay.ThemeDisplay.isSignedIn()){
    alert('Hello ' + Liferay.ThemeDisplay.getUserName() + '. Welcome Back.')
}
else {
    alert('Hello Guest.')
}
```

The example above alerts a signed in user with a personalized greeting. Otherwise, it defaults to a guest greeting. Although this is a basic example, it shows how you can easily define unique user experiences with the `ThemeDisplay` object.

Liferay ThemeDisplay API

For completeness, you can find every available method inside the `Liferay.ThemeDisplay` object in the table below:

Method	Type	Description
getLayoutId	number	
getLayoutRelativeURL	string	
getLayoutURL	string	
getParentLayoutId	number	
isControlPanel	boolean	
isPrivateLayout	boolean	
isVirtualLayout	boolean	
getBCP47LanguageId	number	
getCDNBaseURL	string	
getCDNDynamicResourcesHost	string	
getCDNHost	string	
getCompanyGroupId	number	
getCompanyId	number	Returns the company ID
getDefaultLanguageId	number	
getDoAsUserIdEncoded	string	

Method	Type	Description
getLanguageId	number	Returns the user's language ID
getParentGroupId	number	
getPathContext	string	
getPathImage	string	Returns the relative path of the portlet's image directory
getPathJavaScript	string	Returns the relative path of the directory containing the

portlet's JavaScript source files |

getPathMain | string | Returns the path of the portal instance's main directory | getPathThemeImages | string | Returns the path of the current theme's image directory | getPathThemeRoot | string | Returns the relative path of the current theme's root

directory |

getPlid | string | | getPortalURL | string | | getScopeGroupId | number | Returns the group ID of the current site | getSiteGroupIdOrLiveGroupId | number | Returns the group ID of the live site. This is relevant for staging. | getSessionId | number | | getSiteGroupId | number | | getURLControlPanel | string | | getURLHome | string | | getUserId | number | Returns the user's ID | getUserName | string | Returns the user's name | isAddSessionIdToURL | boolean | | isFreeformLayout | boolean | | isImpersonated | boolean | Returns true if the current user is being impersonated. Authorized administrative users can impersonate act as another user to test that user's account | isSignedIn | boolean | Returns true if the user is logged in to the portal | isStateExclusive | boolean | | isStateMaximized | boolean | | isStatePopUp | boolean | |

Related Topics

Liferay DXP JavaScript Utilities

102.2 Working with URLs in JavaScript

In Java, developers are able to create and work with URLs using their APIs. The `Liferay` global object offers some features to help you create and work with URLs.

This tutorial covers some of the most commonly used methods inside the `Liferay` global JavaScript object to manipulate URLs.

Liferay PortletURL

The `Liferay.PortletURL` class provides a way to create Liferay PortletURL's such as the `actionURL`, `renderURL`, and `resourceURL` through JavaScript. Below is an example configuration:

```
var portletURL = Liferay.PortletURL.createURL(themeDisplay.getURLControlPanel());

portletURL.setDoAsGroupId('true');
portletURL.setLifecycle(Liferay.PortletURL.ACTION_PHASE);
portletURL.setParameter('cmd', 'add_temp');
portletURL.setParameter('javax.portlet.action', '/document_library/upload_file_entry');
portletURL.setParameter('p_auth', Liferay.authToken);
portletURL.setPortletId(Liferay.PortletKeys.DOCUMENT_LIBRARY);
```

Liferay AuthToken

Below is an example configuration for the `Liferay.authToken`:

```
Liferay.authToken = '<%= AuthTokenUtil.getToken(request) %>';
```

Liferay CurrentURL

The `Liferay.currentURL` variable holds the path of the current URL from the server root.

For example, if checked from `my.domain.com/es/web/guest/home`, the value is `/es/web/guest/home`, as shown below:

```
// Inside my.domain.com/es/web/guest/home
console.log(Liferay.currentURL); // "/es/web/guest/home"
```

Liferay CurrentURLEncoded

The `Liferay.currentURLEncoded` variable holds the path of the current URL from the server root encoded in ASCII for safe transmission over the Internet.

For example, if checked from `my.domain.com/es/web/guest/home`, the value is `%2Fes%2Fweb%2Fguest%2Fhome`, as shown below:

```
// Inside my.domain.com/es/web/guest/home
console.log(Liferay.currentURLEncoded); // "%2Fes%2Fweb%2Fguest%2Fhome"
```

Now you know how to manipulate URLs using methods within the `Liferay` global JavaScript object.

Related Topics

Liferay DXP JavaScript Utilities

Liferay Theme Display

102.3 Liferay DXP JavaScript Utilities

This tutorial explains some of the utility methods and objects inside the `Liferay` global JavaScript object.

Liferay Browser

The `Liferay.Browser` object contains methods that expose the current user agent characteristics without the need of accessing and parsing the global `window.navigator` object.

The available methods for the `Liferay.Browser` object are listed in the table below:

Method	Type	Description
acceptsGzip	boolean	
getMajorVersion	number	
getRevision	number	
getVersion	number	
isAir	boolean	
isChrome	boolean	
isFirefox	boolean	

Method	Type	Description
isGecko	boolean	
isIe	boolean	
isIphone	boolean	
isLinux	boolean	
isMac	boolean	
isMobile	boolean	
isMozilla	boolean	
isOpera	boolean	
isRtf	boolean	
isSafari	boolean	
isSun	boolean	
isWebKit	boolean	
isWindows	boolean	

Related Topics

Liferay Theme Display

102.4 Invoking Liferay Services

Liferay DXP provides many web services out-of-the-box to you. These services include retrieving data and information about various assets, creating new assets, and even editing existing assets.

To see a comprehensive list of the available web services, start up a bundle and navigate to `http://localhost:8080/api/jsonws`. This list includes any custom web services that have been deployed to the bundle. These services are useful for creating single page applications, and can even be used to create custom front-ends, both inside and outside of Liferay DXP.

This tutorial explains how to invoke these web services using JavaScript.

Go ahead and get started.

Invoking Web Services via JavaScript

7.0 contains a global JavaScript object called `Liferay` that has many useful utilities. One method is `Liferay.Service`, which is used for invoking JSON web services.

The `Liferay.Service` method takes four possible arguments:

service {string|object}: Either the service name, or an object with the keys as the service to call, and the value as the service configuration object. (Required)

data {object|node|string}: The data to send to the service. If the object passed is the ID of a form or a form element, the form fields will be serialized and used as the data.

successCallback {function}: A function to execute when the server returns a response. It receives a JSON object as it's first parameter.

exceptionCallback {function}: A function to execute when the response from the server contains a service exception. It receives an exception message as it's first parameter.

One of the major benefits of using the `Liferay.Service` method versus using a standard AJAX request is that it handles the authentication for you.

Below is an example configuration of the `Liferay.Service` method:

```
Liferay.Service(
        '/user/get-user-by-email-address',
        {
                companyId: Liferay.ThemeDisplay.getCompanyId(),
                emailAddress: 'test@example.com'
        },
        function(obj) {
                console.log(obj);
        }
);
```

The example above retrieves information about a user by passing in the `companyId` and `emailAddress` of the user in question. The response data resembles the following JSON object:

```
{
        "agreedToTermsOfUse": true,
        "comments": "",
        "companyId": "20116",
        "contactId": "20157",
        "createDate": 1471990639779,
        "defaultUser": false,
        "emailAddress": "test@example.com",
        "emailAddressVerified": true,
        "facebookId": "0",
        "failedLoginAttempts": 0,
        "firstName": "Test",
        "googleUserId": "",
        "graceLoginCount": 0,
        "greeting": "Welcome Test Test!",
        "jobTitle": "",
        "languageId": "en_US",
        "lastFailedLoginDate": null,
        "lastLoginDate": 1471996720765,
        "lastLoginIP": "127.0.0.1",
        "lastName": "Test",
        "ldapServerId": "-1",
        "lockout": false,
        "lockoutDate": null,
        "loginDate": 1472077523149,
        "loginIP": "127.0.0.1",
        "middleName": "",
        "modifiedDate": 1472077523149,
        "mvccVersion": "7",
        "openId": "",
        "portraitId": "0",
        "reminderQueryAnswer": "test",
        "reminderQueryQuestion": "what-is-your-father's-middle-name",
        "screenName": "test",
        "status": 0,
        "timeZoneId": "UTC",
        "userId": "20156",
        "uuid": "c641a7c9-5acb-aa68-b3ea-5575e1845d2f"
}
```

Now that you know how to send an individual request, you're ready to run batch requests.

Batching Requests

Another format for invoking the `Liferay.Service` method is by passing an object with the keys as the service to call and the value as the service configuration object.

Below is an example configuration for a batch request:

```
Liferay.Service(
        {
                '/user/get-user-by-email-address': {
                        companyId: Liferay.ThemeDisplay.getCompanyId(),
                        emailAddress: 'test@example.com'
                }
        },
        function(obj) {
                console.log(obj);
        }
);
```

You can use this format to invoke multiple services with the same request by passing in an array of service objects. Here's an example:

```
Liferay.Service(
        [
                {
                        '/user/get-user-by-email-address': {
                                companyId: Liferay.ThemeDisplay.getCompanyId(),
                                emailAddress: 'test@example.com'
                        }
                },
                {
                        '/role/get-user-roles': {
                                userId: Liferay.ThemeDisplay.getUserId()
                        }
                }
        ],
        function(obj) {
                // obj is now an array of response objects
                // obj[0] == /user/get-user-by-email-address data
                // obj[1] == /role/get-user-roles data

                console.log(obj);
        }
);
```

Next you can learn how to nest your requests.

Nesting Requests

Nested service calls allow you to bind information from related objects together in a JSON object. You can call other services in the same HTTP request and conveniently nest returned objects.

You can use variables to reference objects returned from service calls. Variable names must start with a dollar sign ($).

The example in this section retrieves user data with `/user/get-user-by-id`, and uses the `contactId` returned from that service to then invoke `/contact/get-contact` in the same request.

Note: You must flag parameters that take values from existing variables. To flag a parameter, insert the `@` prefix before the parameter name.

Below is an example configuration that demonstrates the concepts covered in this section:

```
Liferay.Service(
        {
                "$user = /user/get-user-by-id": {
                        "userId": Liferay.ThemeDisplay.getUserId(),
                        "$contact = /contact/get-contact": {
                                "@contactId": "$user.contactId"
```

```
                    }
                }
        },
        function(obj) {
                console.log(obj);
        }
);
```

Here is what the response data would look like for the request above:

```
{
        "agreedToTermsOfUse": true,
        "comments": "",
        "companyId": "20116",
        "contactId": "20157",
        "createDate": 1471990639779,
        "defaultUser": false,
        "emailAddress": "test@example.com",
        "emailAddressVerified": true,
        "facebookId": "0",
        "failedLoginAttempts": 0,
        "firstName": "Test",
        "googleUserId": "",
        "graceLoginCount": 0,
        "greeting": "Welcome Test Test!",
        "jobTitle": "",
        "languageId": "en_US",
        "lastFailedLoginDate": null,
        "lastLoginDate": 1472231639378,
        "lastLoginIP": "127.0.0.1",
        [...]
        "screenName": "test",
        "status": 0,
        "timeZoneId": "UTC",
        "userId": "20156",
        "uuid": "c641a7c9-5acb-aa68-b3ea-5575e1845d2f",
        "contact": {
                "accountId": "20118",
                "birthday": 0,
                [...]
                "createDate": 1471990639779,
                "emailAddress": "test@example.com",
                "employeeNumber": "",
                "employeeStatusId": "",
                "facebookSn": "",
                "firstName": "Test",
                "lastName": "Test",
                "male": true,
                "middleName": "",
                "modifiedDate": 1471990639779,
                [...]
                "userName": ""
        }
}
```

Now that you know how to process requests, you can learn how to filter the results next.

Filtering Results

If you don't want all the properties returned by a service, you can define a whitelist of properties. This returns only the specific properties you request in the object.

Below is an example of whitelisting properties:

```
Liferay.Service(
        {
                '$user[emailAddress,firstName] = /user/get-user-by-id': {
                        userId: Liferay.ThemeDisplay.getUserId()
                }
        },
        function(obj) {
                console.log(obj);
        }
);
```

To specify whitelist properties, place the properties in square brackets (e.g., `[whiteList]`) immediately following the name of your variable. The example above requests only the `emailAddress` and `firstName` of the user.

Below is the filtered response:

```
{
        "firstName": "Test",
        "emailAddress": "test@example.com"
}
```

Next you can learn how to populate the inner parameters of the request.

Inner Parameters

When you pass in an object parameter, you'll often need to populate its inner parameters (i.e., fields).

Consider a default parameter `serviceContext` of type `ServiceContext`. To make an appropriate call to JSON web services you might need to set `serviceContext` fields such as`scopeGroupId`, as shown below:

```
Liferay.Service(
        '/example/some-web-service',
        {
                serviceContext: {
                        scopeGroupId: 123
                }
        },
        function(obj) {
                console.log(obj);
        }
);
```

Now you know how to invoke Liferay services!

Related Topics

Liferay JavaScript APIs

Chapter 103

Front-End Taglibs

Liferay DXP offers a powerful set of taglibs that are fully maintained and integrated. They provide common implementations for UI components and utilities to ensure that your apps, themes, and web content behave in a very clean and efficient way.

In this section of tutorials, you'll learn how to use Liferay DXP's taglibs to build awesome user interfaces.

103.1 Using the Liferay UI Taglib

You can create a lot of components using the Liferay UI taglibs. Liferay DXP's taglibs provide the following benefits to your markup:

- Consistent
- Responsive
- Accessible across your portlets

The full markup generated by the tags can be found in the JSPs of the tag's folder in the Liferay Github Repo.

Now that you know the benefits of Liferay DXP's tags, you can learn how to use them next.

Using Liferay UI Tags

A list of the available Liferay UI taglibs can be found here. To use the Liferay-UI taglib library in your apps, you must add the following declaration to your JSP:

```
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>
```

The Liferay-UI taglib is also available via a macro for your FreeMarker theme and web content templates. Follow this syntax:

```
<@liferay_ui["tag-name"] attribute="string value" attribute=10 />
```

Now you're good to go!

Each taglib has a list of attributes that can be passed to the tag. Some of these are required and some are optional. See the taglibdocs to view the requirements for each tag.

The example below uses the `<liferay-ui:alert>` taglib to create a success alert that the user can close:

```
<liferay-ui:alert
        closeable="true"
        icon="exclamation-full"
        message="Here is our awesome alert example"
        type="success"
/>
```

Here is an example implementation of a `<liferay-ui:user-display>` taglib:

```
<liferay-ui:user-display
        markupView="lexicon"
        showUserDetails="true"
        showUserName="true"
        userId="<%= themeDisplay.getRealUserId() %>"
        userName="<%= themeDisplay.getRealUser().getFullName() %>"
/>
```

Now you know how to use Liferay-UI taglibs in your JSPs!

Related Topics

JavaScript Module Loaders
Loading Modules with the AUI Script in Liferay
Using the Liferay Util Taglib

103.2 Using the Liferay Util Taglib

The Liferay Util taglib is used to pull in other resources into a portlet or theme, it can be used to dictate which resources need to be inserted at the bottom or top of the HTML source.

Using Liferay Util Tags

A list of the available Liferay Util tags can be found here. To use the Liferay Util taglib library in your apps, you must add the following declaration to your JSP:

```
<%@ taglib prefix="liferay-util" uri="http://liferay.com/tld/util" %>
```

The Liferay Util taglib is also available via a macro for your FreeMarker theme and web content templates. Follow this syntax:

```
<@liferay_util["tag-name"] attribute="string value" attribute=10 />
```

Each taglib has a list of attributes that can be passed to the tag. Some of these are required and some are optional. See the taglibdocs to view the requirements for each tag.

Since each of the Liferay Util taglibs is unique, each tag is covered briefly in the sections that follow.

Using Liferay Util Body Bottom

The `<liferay-util:body-bottom>` tag is not a self-closing tag. The content placed between the opening and closing of this tag is moved to the bottom of the `body` tag. When something is passed using this taglib, the body_bottom.jsp is passed markup and outputs in this JSP. The attribute `outputKey` is the reference key for this content.

Below is an example configuration for the `<liferay-util:body-bottom>` tag:

```
<%@ taglib uri="http://liferay.com/tld/portlet" prefix="liferay-portlet" %>
<%@ taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %>
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<%@ page import="com.liferay.portal.kernel.util.PortalUtil" %>
<%@ page import="com.liferay.product.navigation.product.menu.web.constants.ProductNavigationProductMenuPortletKeys" %>

<liferay-theme:defineObjects />

<%
String portletNamespace = PortalUtil.getPortletNamespace(ProductNavigationProductMenuPortletKeys.PRODUCT_NAVIGATION_PRODUCT_MENU);
%>

<liferay-util:body-bottom outputKey="productMenu">
        <div class="lfr-product-menu-panel sidenav-fixed sidenav-menu-slider"
        id="<%= portletNamespace %>sidenavSliderId">
                <div class="product-menu sidebar sidenav-menu">
                        <liferay-portlet:runtime portletName="<%= ProductNavigationProductMenuPortletKeys.PRODUCT_NAVIGATION_PRODUCT_MENU %>" />
                </div>
        </div>
</liferay-util:body-bottom>
```

Using Liferay Util Body Top

The `<liferay-util:body-top>` tag is not a self-closing tag. The content placed between the opening and closing of this tag is moved to the top of the `body` tag. When something is passed using this taglib the body_top.jsp is passed markup and outputs in this JSP. The attribute `outputKey` is the reference key for this content.

Below is an example configuration for the `<liferay-util:body-top>` tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:body-top outputKey="topContent">
        <div>
                <h1>I'm at the top of the page!</h1>
        </div>
</liferay-util:body-top>
```

Using Liferay Util Buffer

`<liferay-util:buffer>` is not a self-closing tag. The content placed between the opening and closing of this tag is saved to the value of the var attribute. This allows a developer to build a piece of markup that can be reused in a JSP.

Below is an example configuration for the `<liferay-util:buffer>` tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:buffer var="myBuffer">
        <small class="text-capitalize text-muted">
                This is my buffer content
        </small>
</liferay-util:buffer>

<div class="container">
        <h1>Welcome!</h1>

        <%= myBuffer %>
</div>

<div class="container">
        <h1>A Wonderful Title!</h1>
        <%= myBuffer %>
</div>
```

Using Liferay Util Dynamic Include

The `<liferay-util:dynamic-inlude>` tag allows you to register some content with the `DynamicIncludeRegistry`. You can read more about the OSGi Service Registry here. It's easier for modules using the OSGi registry to use the content that you include with this tag.

Below is an example configuration for the `<liferay-util:dynamic-inlude>` tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:dynamic-include key="/path/to/jsp#pre" />

<div>
        <p>And here we have our content</p>
</div>

<liferay-util:dynamic-include key="/path/to/jsp#post" />
```

Using Liferay Util Get URL

The `<liferay-util:get-url>` tag scrapes the URL provided by the `url` attribute. If a value is provided for the var attribute, the content from the screen scrape is scoped to that variable. Otherwise, the scraped content is displayed where the taglib is used.

Below is a basic example configuration for the `<liferay-util:get-url>` tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:get-url url="https://www.google.com/" />
```

Here is an example that uses the var attribute:

```
<liferay-util:get-url url="https://www.google.com/" var="google" />

<div>
        <h2>We stole <a href="https://www.google.com/">Google</a>, here it is.</h2>

        <div class="google">
                <%= google %>
        </div>
</div>
```

Using Liferay Util HTML Bottom

The `<liferay-util:html-bottom>` tag is not a self-closing tag. The content placed between the opening and closing of this tag will be moved to the bottom of the `html` tag. When something is passed using this taglib the bottom.jsp is passed markup and outputs in this JSP. The attribute `outputKey` is the reference key for this content.

Below is an example of using `<liferay-util:html-bottom>`. Commonly, JavaScript is passed to this tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:html-bottom outputKey="taglib_alert_user">
        <aui:script use="liferay-alert">
                new Liferay.Alert(
                        {
                                closeable: true,
                                message: 'Just saying hello from the
                                &lt;liferay-util:html-bottom&gt; taglib!',
                                type: 'success'
                        }
                ).render(#wrapper);
        </aui:script>
</liferay-util:html-bottom>
```

Using Liferay Util HTML Top

The `<liferay-util:html-top>` tag is not a self-closing tag. The content placed between the opening and closing of this tag is moved to the `head` tag. When something is passed using this taglib the top_head.jsp is passed markup and outputs in this JSP. The attribute `outputKey` is the reference key for this content.

Below is an example configuration for the `<liferay-util:html-top>` tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:html-top>
        <link data-senna-track="permanent" href="/path/to/style.css" rel="stylesheet" type="text/css" />
</liferay-util:html-top>
```

Using Liferay Util Include

The `<liferay-util:include>` tag can be used to include other JSP files in a portlet. This can increase readability as well as provide separation of concerns for JSP files.

Below is an explanation of some of the available attributes:

- `page`: This attribute is required. The value of this attribute is the path to the JSP or JSPF to be included.
- `servletContext`: Refers to the request context that the included JSP should use. Passing `<%= application %>` to this attribute allows the included JSP to use the same `request` object and other objects that might be set in the prior JSP.

Below is an example configuration for the `<liferay-util:include>` tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:include page="/path/to/view.jsp" servletContext="<%= application %>" />
```

Using Liferay Util Param

The `<liferay-util:param>` tag can be used to add a parameter value to a URL. This tag is best used when combined with the `<liferay-util:include>` tag for accessing new parameter values in another JSP.

Below is an example configuration for the `<liferay-util:param>` tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:include page="/path/to/answer.jsp" servletContext="<%= application %>">
        <liferay-util:param name="answer" value="42" />
</liferay-util:include>
```

The new parameter can then be used in `answer.jsp` like this:

```
<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>

<div>
        <p>The answer to life the universe and everything is
        <%= ParamUtil.getString(request, "answer") %></p>
</div>
```

Using Liferay Util Whitespace Remover

The `<liferay-util:whitespace-remover>` tag is used for removing all whitespace from whatever is included between the opening and closing of the tag.

Below is an example configuration for the `<liferay-util:whitespace-remover>` tag:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:whitespace-remover>
        <div class="input-container">
                <label for="myInput">
                        Is the &lt;liferay-util:whitespace-remover&gt; taglib
                        fantastic!
                </label>

                <input class="input" id="myInput" name="myInput" type="checkbox">
        </div>
</liferay-util:whitespace-remover>
```

Now you know how to use the Liferay-Util taglib!

Related Topics

Loading Modules with the AUI Script in Liferay

Using the Liferay UI Taglib

CHAPTER 104

HTML FORMS

Forms are a must-have tool in any developer's UI bag. Rather than write your HTML from scratch, Liferay provides you with taglibs that offer everything standard HTML elements have, as well as some additional attributes that are Liferay specific. For a full list of all the available taglibs and attributes, look at Liferay's API docs on docs.liferay.com

Using a combination of Liferay's `AUI` taglibs and Lexicon classes, you can create forms in your app's JSPs. You can learn how to create an MVC portlet project in the using-the-mvc-portlet-template tutorial.

In this section of tutorials, you'll learn how to use the platform features to create forms with ease.

104.1 Forms and Validation

Creating forms is easy and flexible, thanks to Liferay DXP's form, input, and validation tags.

The `<aui:form>` tag sets up the necessary code (HTML, JS), uses Liferay JavaScript APIs, calls the validation framework, and submits the form data to the back-end.

The *input* tags (`<aui:input>`, `<liferay-ui:input-*>`, etc.) generate the necessary code (HTML, CSS, JS) to provide you with a consistent UI throughout your form. These tags also initialize the field's state (e.g. `value`, `disabled`, `readonly`, etc.).

The `<aui:validator>` tag lets you create validation rules for your form's input tags. This ensures that your users enter the proper data in the format you expect before it gets sent to the back-end for final validation and processing.

The following tags support validators natively:

- `<aui:input>`
- `<aui:select>`
- `<liferay-ui:input-date>`
- `<liferay-ui:input-search>`

All other fields can be validated using the method described in the Adding Custom Validators in JavaScript section below.

Get started by creating a basic form next.

Creating Your First Form

Below is an example configuration for a simple form in a JSP:

```
<aui:form action="<%= myActionURL %>" method="post" name="myForm">
        <aui:input label="My Text Input" name="myTextInput" type="text"
        value='<%= "My Text Value" %>' />

        <aui:button type="submit" />
</aui:form>
```

Although this form is simple, it provides all that's needed to pass the data to the back-end. The full HTML generated for this example is shown below:

```
<form action="http://localhost:8080/web/guest/home?p_p_id=my_portlet&amp;p_p_lifecycle=1&amp;p_p_state=normal&amp;p_p_mode=view&amp;p_p_col_id=column-
1&amp;p_p_col_count=3&amp;_my_portlet_javax.portlet.action=%2Fmy%2Faction&amp;_my_portlet_mvcRenderCommandName=%2Fmy%2Faction&amp;p_auth=WAxorpsN" class="fo
fm-namespace="_my_portlet_" id="_my_portlet_myForm" method="post" name="_my_portlet_myForm">
        <input class="field form-control" id="_my_portlet_formDate"
        name="_my_portlet_formDate" type="hidden" value="1472516415545">

        <div class="form-group input-text-wrapper">
                <label class="control-label"
                for="_my_portlet_myTextInput">My Text Input</label>

                <input class="field form-control" id="_my_portlet_myTextInput"
                name="_my_portlet_myTextInput" type="text"
                value="My Text Value">
        </div>

        <button class="btn  btn-primary btn-default" id="_my_portlet_kpyg"
        type="submit">
                <span class="lfr-btn-label">Save</span>
        </button>
</form>
```

As you can see, the tags provide all this to you for very little work! Next you can learn how to add validation to your forms.

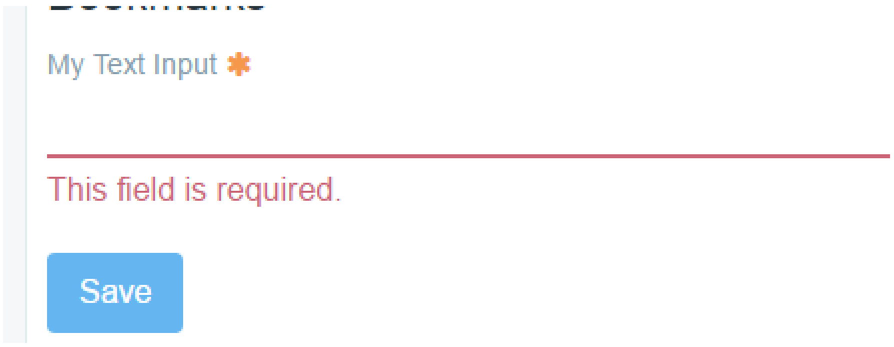
Adding Validation

What if you want to ensure the user enters the required data? Add validators.

An example configuration is shown below:

```
<aui:input label="My Text Input" name="myTextInput" type="text"
value='<%= "My Text Value" %>'>
        <aui:validator name="required" />
</aui:input>
```

This forces the user to enter something before the form is submitted.



This field is required.

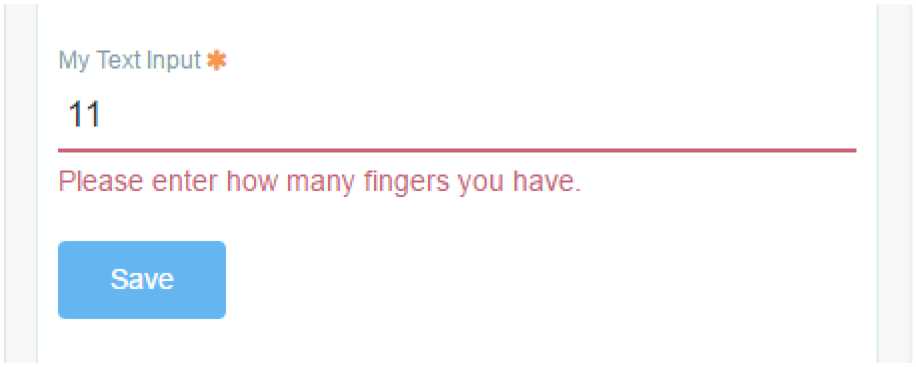
Figure 104.1: A field with a failed required validator.

What if you wanted to restrict the value to a number between `0` and `10`? There's a validator for that. Each of the validators shown below must pass to submit the form:

```
<aui:validator name="required" />
<aui:validator name="number" />
<aui:validator name="range">[0,10]</aui:validator>
```

You can even customize the error message, as shown below:

```
<aui:validator errorMessage="Please enter how many fingers you have."
name="range">[0,10]</aui:validator>
```



Please enter how many fingers you have.

Figure 104.2: A required validator with a custom error message.

Below is a list of all the available validation rules:

acceptFiles: Specifies that the field can contain only the file types given. Each file extension must be separated by a comma. For example `<aui:validator name="acceptFiles">'jpg,png,tif,gif'</aui:validator>`

alpha: Allows only alphabetic characters.

alphanum: Allows only alphanumeric characters.

date: Allows only a date.

digits: Allows only digits.

email: Allows only an email address.

equalTo: Allows only contents equal to some other field that has the specified field ID. The ID is declared in the opening and closing validator tags. For example `<aui:validator name="equalTo">'#<portlet:namespace />password'</aui:validator>`

max: Allows only an integer value less than the specified value. For example, a `max` value of `20` is specified here `<aui:validator name="max">20</aui:validator>`

maxLength: Allows a maximum field length of the specified size. The syntax is the same as `max`.

min: Allows only an integer value greater than the specified value. The syntax is the same as `max`.

minLength: Allows a field length longer than the specified size. The syntax is the same as `max`.

number: Allows only numerical values.

range: Allows only a number between the specified range. For example, a range between 1.23 and 10 is specified here `<aui:validator name="range">[1.23,10]</aui:validator>`

rangeLength: Allows a field length between the specified range. For example, a range between 3 and 8 characters long is specified here `<aui:validator name="rangeLength">[3,8]</aui:validator>`

required: Prevents a blank field.

url: Allows only a URL value.

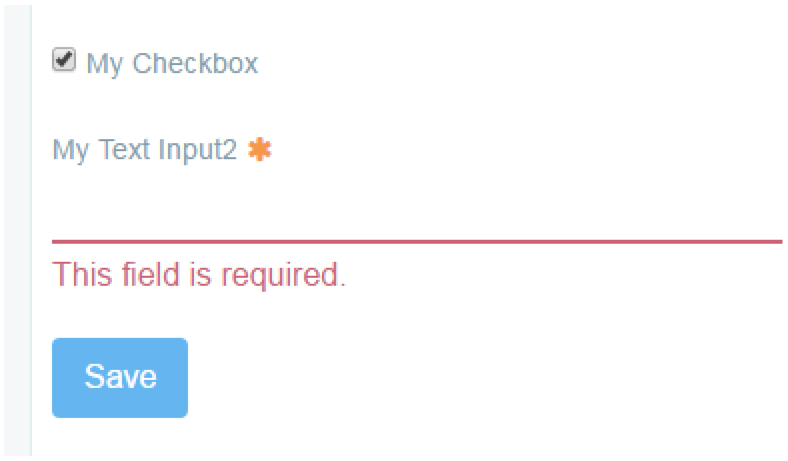
Now that you know how to validate your forms, you can learn how to conditionally require user input next.

Conditionally Requiring A Field

Sometimes you'll want to validate a field based on the value of another field. You can do this by checking for that condition in a JavaScript function within the `required` validator's body.

Below is an example configuration:

```
<aui:input label="My Checkbox" name="myCheckbox" type="checkbox" />

<aui:input label="My Text Input" name="myTextInput" type="text">
        <aui:validator name="required">
                function() {
                        return AUI.$('#<portlet:namespace />myCheckbox').prop('checked');
                }
        </aui:validator>
</aui:input>
```



This field is required.

Figure 104.3: Fields can required based on other conditions.

Next you can learn how to add custom validators.

Adding Custom Validators

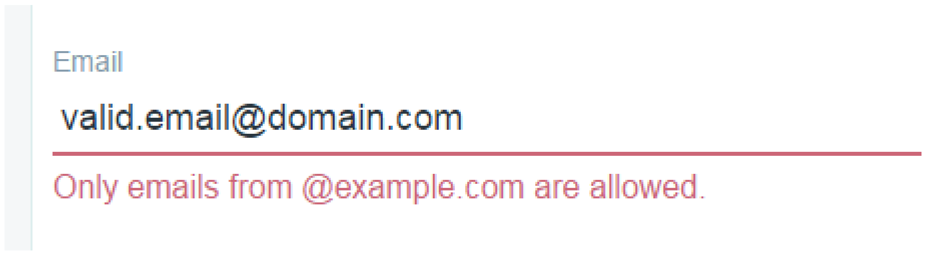
So far, you've only seen the default set of AUI validator rules. What if you need something that the default rules don't provide?

You can write your own validator and optionally supplement it with built-in validators, as shown below:

```
<aui:input label="Email" name="email" type="text">
        <aui:validator name="email" />

        <aui:validator errorMessage="Only emails from @example.com are allowed."
        name="custom">
                function(val, fieldNode, ruleValue) {
                        var regex = new RegExp(/@example\.com$/i);

                        return regex.test(val);
                }
        </aui:validator>
</aui:input>
```

This example runs the regular `email` validator, as well as a `custom` domain validator.

Next you can learn how to add custom validators in JavaScript.


Email

valid.email@domain.com

Only emails from @example.com are allowed.

Figure 104.4: You can write your own custom validations.

Adding Custom Validators in JavaScript

Sometimes you need to add additional validation dynamically after the page has rendered. Perhaps some additional fields were added to the DOM via an AJAX request.

To do this, you must access the `Liferay.Form` object, as demonstrated below:

```
<aui:script use="liferay-form">
        var form = Liferay.Form.get('<portlet:namespace />myForm');
        // ...
```

The `Liferay.Form` object allows you to `get()` and `set()` `fieldRules`.

`fieldRules` are the JavaScript equivalent of all the validators attached to the form.

This example below uses a `custom` validator and the built-in `number` validator.

```
<aui:script use="liferay-form">
        var form = Liferay.Form.get('<portlet:namespace />myForm');

        var oldFieldRules = form.get('fieldRules');

        var newFieldRules = [
                {
                        body: function (val, fieldNode, ruleValue) {
                                return (val !== '2');
                        },
                        custom: true,
                        errorMessage: 'must-not-equal-2',
                        fieldName: '<portlet:namespace />fooInput',
                        validatorName: 'custom_fooInput'
                },
                {
                        fieldName: '<portlet:namespace />fooInput',
                        validatorName: 'number'
                }
        ];

        var fieldRules = oldFieldRules.concat(newFieldRules);

        form.set('fieldRules', fieldRules);

</aui:script>
```

Notice that `newFieldRules` are combined with `oldFieldRules`, using `concat()`. You could leave this part out, and you'd have a whole new set of validators for the whole form.

Next you can learn how to validate your forms manually.

Manual Validation

You may need to execute validation on a field, based on some event not typical of user input. For instance, you may need to validate a related field at the same time.

To do this, you must access the `formValidator` object and call the `validateField()` method, passing the field's `name`. Below is an example configuration:

```
<aui:input label="Old Title" name="oldTitle" type="text">
        <aui:validator errorMessage="The New Title cannot match the Old Title"
        name="custom">
                function(val, fieldNode, ruleValue) {
                        <portlet:namespace />checkOtherTitle('<portlet:namespace />newTitle');

                        // check if old and new titles are a match

                        return !match;
                }
        </aui:validator>
</aui:input>

<aui:input label="New Title" name="newTitle" type="text">
        <!-- same custom validator -->
</aui:input>

<aui:script use="liferay-form">
        function <portlet:namespace />checkOtherTitle(fieldName) {
                var formValidator =
                Liferay.Form.get('<portlet:namespace />myForm').formValidator;

                formValidator.validateField(fieldName);
        }
</aui:script>
```

Now you know how to create and validate forms!

Related Topics

Front-End Taglibs

104.2 Creating Forms with Liferay's Taglibs

This tutorial demonstrates how to:

- Create a form using 7.0's taglibs in your application's JSPs

To begin, take a look at the Portlet Configuration application's add_configuration_template.jsp form for example:

```
<aui:form action="<%= updateArchivedSetupURL %>"
cssClass="container-fluid-1280" method="post" name="fm">
        <aui:input name="redirect" type="hidden" value="<%= redirect %>" />
        <aui:input name="portletResource" type="hidden"
        value="<%= portletResource %>" />

        <aui:fieldset-group markupView="lexicon">
                <aui:fieldset>

                        <%
                        String name = StringPool.BLANK;
                        boolean useCustomTitle =
                        GetterUtil.getBoolean(portletPreferences.getValue
                        ("portletSetupUseCustomTitle", null));
                        if (useCustomTitle) {
                                name = PortletConfigurationUtil.getPortletTitle
                                (portletPreferences, LocaleUtil.toLanguageId
                                (themeDisplay.getSiteDefaultLocale()));
                        }
                        %>

                        <aui:input name="name" placeholder="name"
                        required="<%= true %>" type="text" value="<%= name
                        %>">
                                <aui:validator name="maxLength">75
                                </aui:validator>
                        </aui:input>
                </aui:fieldset>
        </aui:fieldset-group>

        <aui:button-row>
                <aui:button type="submit" />

                <aui:button href="<%= redirect %>" type="cancel" />
        </aui:button-row>
</aui:form>
```

As you can see, a standard form uses quite a few taglibs. If you take a closer look, you'll notice that all of the taglibs are prefixed with aui. aui stands for AlloyUI, a JavaScript framework that uses Bootstrap to allow you to create UI components, easily and effectively. In order to use these AUI tags, you will need to have the AUI taglib declaration imported into your JSP. You'll take care of this next.

Adding the Taglib Declaration

The first thing you'll need to do is make sure you have the `<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui"%>` declaration is in your JSP. This can be added to the main view of your app, `view.jsp` for example, or it can be added to a separate JSP file, such as `init.jsp`, and imported into the JSP in which you want to show the form. For example, the Portlet Configuration application includes the taglib declarations in its `init.jsp`, and then imports the `init.jsp` into its `view.jsp`, using the following line:

```
<%@ include file="/init.jsp"%>
```

Once the AUI taglib declaration is imported, you can move onto creating the form next.

Creating the Form

A form's design is determined by the input needed from the user. To that end, there are multiple design possibilities. The examples covered in the sections that follow are one possible design, and highlight some of the available attributes for the tags. The steps are laid out in a natural order, however you can jump to any section you wish.

For a full list of the available attributes for the form tags covered, checkout the API docs for the AUI Tags.

Go ahead and get started by adding the form tag next.

Adding the Form Wrapper

Start off by adding a `<aui:form>` tag to your jsp. Make sure to add a closing `<aui:form/>` tag, to wrap your form. This acts as a wrapper for your form and offers some additional styling and custom portlet namespacing for you to use. If you are familiar with HTML `<form>` tags, `<aui:form>` tags are configured the same way. For instance, the Bloggs Aggregator app has the following configuration:

```
<aui:form action="<%= configurationActionURL %>" method="post" name="fm">
```

The action attribute specifies where to send the form data when the form is submitted. The method attribute defines the method to use to send the form data(in most cases this will be post). Finally, the name attribute specifies the name for the form, as well as the ID for the component instance for the form. It's important to note that, by default, aui:form places the portlet namespace in front of the name and id attributes. This is also the default behavior for the aui:input tag as well.

You can find a full list of the available attributes for the <aui:form> tag in the AUI Form Taglib Docs.

Now that your form element is created, you can add your fieldsets next.

Adding the Fieldset Groups and Fieldsets

The next main element is the <aui:fieldset-group> tag. This tag creates a <div> to group fieldset elements of the form. Looking at the Portlet Configuration example, you can see the following pattern:

```
<aui:fieldset-group markupView="lexicon">
```

It's important to note the addition of the markupView="lexicon" attribute. This ensures that the lexicon HTML markup and CSS styles are used to render the element, rather than the standard markup.

Add the <aui:fieldset-group markupView="lexicon"> tag, just below the <aui:form> tag you just added. And make sure to add the closing </aui:fieldset-group> tag just before the closing </aui:form> tag. Your form should look something like this at this point:

```
<aui:form action="<%= ActionURL %>" method="post" name="fm">
    <aui:fieldset-group markupView="lexicon">

    </aui:fieldset-group>
</aui:form>
```

Now that your fieldset group is added, you can add your fieldset next. The <aui:fieldset> tag creates a <div> to group related form elements and offers some additional styling attributes as well.

Use the following pattern to add the <aui:fieldset> tag to your form:

```
<aui:fieldset >

</aui:fieldset>
```

Alternatively, if you have multiple fieldsets, you can update your tags to be collapsible, using the following pattern:

```
<aui:fieldset collapsed="<%= true %>" collapsible="<%= true %>" >

</aui:fieldset>
```

You can find a full list of the available attributes for the <aui:fieldset> tag in the AUI Fieldset Taglib Docs.

Next, you can add your input fields.

Adding the Input Fields

Each fieldset is used to group similar form elements together. In this section, you'll take a look at the different kinds of input fields you can use in your form.

AUI input fields use the following pattern:

```
<aui:input label="label" name="name" type="type" />
```

The `label` attribute sets the label for the input field. The `name` attribute sets the name for the field. Finally, the `type` attribute sets the type of input to use for the field.

The `type` attribute supports the following types:

- file: Adds a file browser.
- text: The default value if no `type` is specified. Adds a text input field.
- hidden: Adds a hidden text field.
- assetCategories: Adds a `liferay-ui:asset-categories-selector` component.
- assetTags: Adds a `liferay-ui:asset-tags-selector` component.
- textarea: Adds a textarea box.
- timeZone: Adds a `liferay-ui:input-time-zone` component.
- password: Adds a password input field.
- checkbox: Adds a checkbox.
- radio: Adds a radio button.
- color: Adds a HTML color picker.
- editor: Adds a `liferay-ui:input-editor` component.
- email: Adds an email input field.
- number: Adds a number selector.
- range: Adds a range slider.
- resource: Adds a `liferay-ui:input-resource` component.
- toggle-card: Adds a Toggle Card
- toggle-switch: Adds a Toggle Switch
- url: Adds a URL input field.

For a full list of all the attributes available for the `<aui:input>` tag, check out the AUI Input Taglib Docs.

Once you've added all of your input fields, you can move onto the form's buttons next.

Adding the Button Rows and Buttons

Taking a look at the portlet configuration application example once again, you can see that form buttons follow the pattern below:

```
<aui:button-row>
        <aui:button type="submit" />

        <aui:button href="<%= redirect %>" type="cancel" />
</aui:button-row>
```

The `<aui:button-row>` tags acts as a wrapper for the form's buttons, and offers some additional styling through the `cssClass` attribute.

The `<aui:button>` tag is a standard input button, with some additional attributes. It supports the `button`, `submit`(the default type), `cancel`, and `reset` types for the `type` attribute. Note that you if you wish to emphasize the button as a primary action, you can add the `primary="true"` attribute to your button:

```
<aui:button
    cssClass="btn-lg"
    id="submit"
    label="save"
    primary="<%= true %>"
    type="submit"
/>
```

For a full list of the attributes available for the `<aui:button>` tag, check out the AUI Button Taglib Docs.
Your form is complete!

Related Topics

Portlet Decorators
Basic Forms
Liferay Theme Generator

104.3 Form Navigator Extensions

Some data-centric applications require the creation of large data-entry forms. Examples abound in healthcare, transportation, pharmaceutical, or any other heavily regulated industry. For these applications, you need an easy way to section off your forms into easily navigable groups.

Since 7.0, the Form Navigator framework enables you to add new sections and section categories dynamically to existing form navigation. The framework includes a well-described API and a powerful `liferay-ui` tag called `form-navigator`. It's easy to use and facilitates organizing large forms into sections of input and categories.

Form Navigators can be used in two ways: customizing a Form Navigator that already exists in the portal, and creating your own Form Navigator for your application. This tutorial demonstrates customizing an existing form. It references source code from an example portlet called the Form Nav Extension portlet. On GitHub, you can find its complete project called form-nav-extension-portlet. You can also download the Form Nav Extension portlet's bundle `form-nav-extension-portlet-1.0.jar`. To download it, go to its GitHub page and click the *View Raw* link.

Before extending a Form Navigator, you should understand the parts of the Form Navigator Framework and what they do.

Understanding the Parts of the Form Navigator Framework

Form Navigator implementations contain the following parts:

- **A JSP that contains a form**: The form must specify a `form-navigator` tag. All the form's input sections tie into the Form Navigator.
- **Sections**: A section (or entry) is a JSP that specifies inputs and a Java class that models the section and ties it into the Form Navigator.
- **Categories**: A category aggregates one or more sections. A Java class models it and ties it into the Form Navigator.
- **IDs**: A Form Navigator and its categories should be publicly identified. That is, they should all have IDs that can be looked up in a public API (e.g., Javadoc). A developer extending a Form Navigator must otherwise have access to the Form Navigator's source code in order to find the IDs.

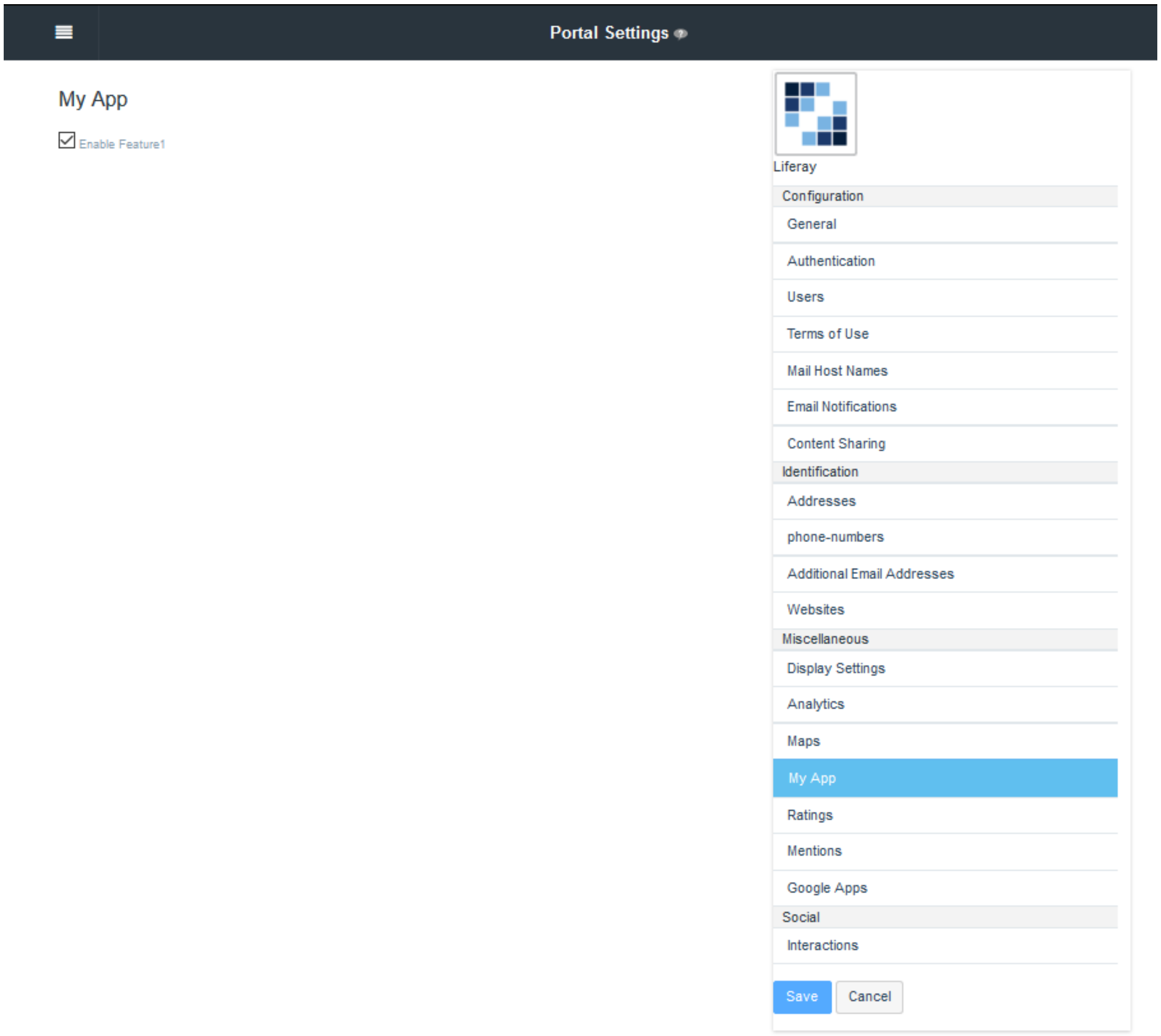


Figure 104.5: The Form Navigator framework lets you add your app's configuration forms to existing form navigators, like the one used in Portal Settings.

Liferay's Form Navigator implementations meet all these requirements. They're implemented similarly and their IDs are published in the Javadoc for the class `FormNavigatorConstants`.

Note: Form Navigator extensions implemented using portal properties and form navigation entry JSPs are deprecated but still supported in 7.0. All new Form Navigator extensions should be implemented as this tutorial describes.

Now that you know the parts of a Form Navigator, it'll be easier for you to extend one.

Extending a Form Navigator

Here's an overview of the steps to extend a Form Navigator:

- **Step 1: Implement a component portlet project to accommodate form navigation.**
- **Step 2: Create a JSP for each new section of inputs.**
- **Step 3: Identify the Form Navigator and category (if any) you're extending.**
- **Step 4: Create new categories.**
- **Step 5: Create new sections.**

Let's set up a component portlet project to support form navigation.

Step 1: Implement a component portlet project to accommodate form navigation

First, your component portlet must be implemented as an OSGi bundle. You can develop it in any environment that supports creating a bundle. Please refer to the 7.0 tutorial section Tooling to learn about development environments. The Form Nav Extension portlet was created with BLADE based on the `blade.portlet.jsp` template. This is also covered in the tutorial section linked above.

A Form Navigator extension bundle's metadata must do the following things:

- Specify the bundle's symbolic name for your servlet context to target
- Include your project's classes, JSPs, and resource bundles (for localization)
- Include the `JspAnalyzerPlugin` to generate generate metadata for the JSPs' dependencies
- Specify a web context path for the Form Navigator classes to associate with the JSPs

You should use a `bnd.bnd` file to specify this metadata. Your Bnd file should include definitions and directives similar to those specified in the Form Nav Extension portlet's `bnd.bnd` file:

```
Bundle-Name: Form Nav Extension Portlet
Bundle-SymbolicName: com.liferay.docs.formnavextensionportlet
Bundle-Version: 1.0.0
Include-Resource:\
        classes,\
        META-INF/resources=src/main/resources/META-INF/resources
-jsp: *.jsp,*.jspf
-plugin.jsp: com.liferay.ant.bnd.jsp.JspAnalyzerPlugin
Web-ContextPath: /formnavextensionportlet
```

The `Bundle-Name` value is arbitrary, but should be recognizable and unique. The `Bundle-SymbolicName` must be unique–the project's package path makes for a good symbolic name. For the `Include-Resource`, make sure to include your project's classes and the root path of its JSPs. The directive below includes all the project's `.jsp` and `.jspf` files:

```
-jsp: *.jsp,*.jspf
```

And the following directive to specify a plugin to include all the JSP dependencies:

```
-plugin.jsp: com.liferay.ant.bnd.jsp.JspAnalyzerPlugin
```

Lastly, the `Web-ContextPath` specifies the root of the portlet's web context.

As you progress through this tutorial, you'll refer to the metadata in your portlet's classes. Before diving into the Java classes, howerver, let's create JSPs for your sections' inputs.

Step 2: Create a JSP for each new section of inputs

The existing Form Navigator has a form, and each of its sections extend the form with sets of input. Your section will add its own inputs. You should create your section's JSP under the `META-INF/resources` property folder you defined in your `bnd.bnd` file's `Include-Resource` directive. The example Bnd file specified the folder like this:

```
META-INF/resources=src/main/resources/META-INF/resources
```

Under the folder, you can add a JSP for each section of inputs you want to add to the Form Navigator. Feel free to organize these with subfolders.

The Form Nav Extension portlet's JSP file `/portal_settings/my_app.jsp` provides a checkbox input to enable/disable My App's feature in the portal:

```
    <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

    <%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
    <%@ taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %>

    <%@ page import="com.liferay.docs.formnavextensionportlet.MyAppWebKeys" %>
    <%@ page import="com.liferay.portal.kernel.util.GetterUtil" %>
    <%@ page import="com.liferay.portal.kernel.util.ResourceBundleUtil" %>

    <%@ page import="java.util.ResourceBundle" %>

    <%
    boolean companyMyAppFeatureEnabled = GetterUtil.getBoolean(request.getAttribute(MyAppWebKeys.COMPANY_MY_APP_FEATURE_ENABLED));

    ResourceBundle resourceBundle = ResourceBundleUtil.getBundle("content.Language", request.getLocale(), getClass());
    %>

    <h3><liferay-ui:message key='<%= resourceBundle.getString("my-app-features") %>' /><h3>

    <aui:input checked="<%= companyMyAppFeatureEnabled %>" label='<%= resourceBundle.getString("enable-my-app-feature") %>' name="settings--
myAppFeatureEnabled--" type="checkbox" value="<%= companyMyAppFeatureEnabled %>" />
```

The input's name is `settings--myAppFeatureEnabled--`. So that the Form Navigator detects the inputs automatically, make sure to start each of your input's names with `settings--` and end them with `--`. Add all the inputs you need in each of your sets of inputs.

After creating section JSPs, you must find out the IDs of the existing Form Navigator and categories you're adding sections to. You refer to these IDs in the category and section classes you'll create to represent your Form Navigator extensions.

Step 3: Identify the form navigator and category you're extending

Liferay's class `FormNavigatorConstants` specifies constant values for all the portlet Form Navigators and categories. The identifiers follow the naming conventions below, where you substitue *FOO* for the navigator's or category's name:

- **Form navigator**: `FORM_NAVIGATOR_ID_FOO`
- **Category**: `CATEGORY_KEY_FOO`

Note the names of the constants that match the Form Navigator and categories you're extending–you'll refer to them in your Java classes in the next steps.

Step 4: Create new categories

To add a new category, create a class that implements the `FormNavigatorCategory` interface. The class needs a `@Component` annotation to register it as a service in Liferay's module framework. Here are the things your category class's component annotation should do:

- Declare that the component is of service type `FormNavigatorCategory.class`
- Request immediate loading
- Optionally, specify the category's entry order relative to the Form Navigator's other categories. The higher the category's order integer relative to the order of the other categories, the higher the category is listed in the form navigation.

Here's an example component annotation for a category:

```
@Component(
    immediate = true,
    property = {"form.navigator.category.order:Integer=20"},
    service = FormNavigatorCategory.class
)
```

Next, you implement the `FormNavigatorCategory` methods:

- `getFormNavigatorId`: Return the Form Navigator's constant you noted previously from `FormNavigatorConstants`
- `getKey`: Return an identifier for your category. You can optionally create a public class like `FormNavigatorConstants`, to publish your project's identifiers.
- `getLabel(Locale)`: Return the localized category label. You can create a `Language.properties` file in your project's `src/main/resources/content` folder and specify a key/value pair for the category label. You can then add the localized value of the property in language properties files for other locales.

For example, here's a Form Navigator category implementation for the *Social* Portal Setting category:

```
package com.liferay.portal.settings.web.internal.servlet.taglib.ui;

import com.liferay.portal.kernel.language.LanguageUtil;
import com.liferay.portal.kernel.servlet.taglib.ui.FormNavigatorCategory;
import com.liferay.portal.kernel.servlet.taglib.ui.FormNavigatorConstants;

import java.util.Locale;

import org.osgi.service.component.annotations.Component;

/**
 * @author Sergio González
 * @author Philip Jones
 */
@Component(
    immediate = true, property = {"form.navigator.category.order:Integer=20"},
    service = FormNavigatorCategory.class
)
public class CompanySettingsSocialFormNavigatorCategory
    implements FormNavigatorCategory {
```

```
    @Override
    public String getFormNavigatorId() {
        return FormNavigatorConstants.FORM_NAVIGATOR_ID_COMPANY_SETTINGS;
    }

    @Override
    public String getKey() {
        return FormNavigatorConstants.CATEGORY_KEY_COMPANY_SETTINGS_SOCIAL;
    }

    @Override
    public String getLabel(Locale locale) {
        return LanguageUtil.get(locale, "social");
    }

}
```

After you've implemented new Form Navigator categories, you can add new Form Navigator sections.

Step 5: Create new sections

To add a new section (entry) that uses a JSP, create a class that extends the abstract base class `BaseJSPFormNavigatorEntry` and implements the `FormNavigatorEntry` interface. The `BaseJSPFormNavigatorEntry` base class integrates the section's JSP with the Form Navigator. In both these parts of your class declaration, you must specify the Form Navigator's model bean class as the generic type on which they operate. For example, if your Form Navigator's model bean class is `User`, your decarlation would be like this:

```
public class MyEntry extends BaseJSPFormNavigatorEntry<User>
        implements FormNavigatorEntry<User>
```

There are a couple different ways to determine the model bean class. If you can access the Form Navigator's JSP source code, inspect the `form-navigator` element's `formModelBean` attribute value. The model bean class is the class type of the object passed in as the `form-navigator`'s `formModelBean` attribute.

You can also deduce the model bean class from the name of the ID's constant in `FormNavigatorConstants`. The word(s) right after `FORM_NAVIGATOR_ID_` in the constant's name hints at the class type. For example, if the navigator's ID is `FORM_NAVIGATOR_ID_USERS_SETTINGS`, then `User` is the model bean class; if the ID is `FORM_NAVIGATOR_ID_ORGANIZATIONS`, then `Organization` is the class; etc. Note: this is only a hint, and there are exceptions. For example, if the ID is is `FORM_NAVIGATOR_ID_SITES`, then `Group` is the class.

For example, here's a class declaration of a `FormNavigatorEntry`, from the Form Nav Extension portlet:

```
    @Component(
        immediate = true,
        property = {
            "form.navigator.entry.order:Integer=71"
        },
        service = FormNavigatorEntry.class
    )
    public class MyAppCompanySettingsFormNavigatorEntry
        extends BaseJSPFormNavigatorEntry<Company>
            implements FormNavigatorEntry<Company> {

            // ...

    }
```

It implements an entry on the `Company` model bean for the Portal Settings Form Navigator: the navigator identifed by the constant `FormNavigatorConstants.FORM_NAVIGATOR_ID_COMPANY_SETTINGS`.

The class also includes the @Component annotation, which is the next thing an entry must specify. An entry's @Component annotation registers the entry as a service in Liferay's module framework.

Here's what a Form Navigation entry's component annotation should do:

- Declare that the component is of service type FormNavigatorEntry.class
- Request immediate loading
- Optionally, specify a form.navigator.entry.order property for the entry relative to the other entries in the category. The higher the entry's order integer relative to the order of the category's other entries, the higher the entry is listed in the category. For example, @Component(property = {"form.navigator.entry.order:Integer=71"}, service = FormNavigatorEntry.class)

Except for your entry's order (optional), your entry's @Component annotation should look similar to the previous example's annotation. Next, you'll implement the entry class's methods.

The FormNavigatorEntry implementation must implement the following methods:

- getFormNavigatorId: Return the Form Navigator's constant you noted previously from FormNavigatorConstants

- getCategoryKey: Return the Form Navigator category constant you noted previously from FormNavigatorConstants

- getKey: Return an identifier for your entry. You can optionally create a public class like FormNavigatorConstants, to publish your project's identifiers.

- getLabel(Locale): Return the entry's localized label. You can create a Language.properties file in your project's src/main/resources/content folder and specify a key/value pair for the entry label.

- getJspPath: Return the path to the entry's JSP, starting from the path you specified previously for your bnd.bnd file's META-INF/resources property.

- include(HttpServletRequest, HttpServletResponse): Sets the request and response attributes for displaying the entry's HTML. You can retrieve the form's current settings and pass them to the request. You can optionally use a template (e.g., FreeMarker or Velocity) to render the form page, as long as you completely override BaseJSPFormNavigatorEntry's include method. The Form Nav Extension portlet's entry class's include method passes to the request the current settings that were saved as portlet preferences. It doesn't use a template language and instead calls BaseJSPFormNavigatorEntry's include method.

```
@Override
public void include(HttpServletRequest request, HttpServletResponse response)
        throws IOException {

    ThemeDisplay themeDisplay =
        (ThemeDisplay)request.getAttribute(WebKeys.THEME_DISPLAY);

    PortletPreferences companyPortletPreferences =
        PrefsPropsUtil.getPreferences(themeDisplay.getCompanyId(), true);

    boolean companyMyAppFeatureEnabled =
        PrefsParamUtil.getBoolean(
            companyPortletPreferences, request, "myAppFeatureEnabled",
            true);

    request.setAttribute(
        MyAppWebKeys.COMPANY_MY_APP_FEATURE_ENABLED,
        companyMyAppFeatureEnabled);
    super.include(request, response);
}
```

- setServletContext(ServletContext): In this method, you set the parent entry class's servlet context. Then, using a @Reference annotation, you unbind the servlet context from its current target and target it to your app's OSGi bundle. First, add the @Reference annotation. Next, unbind the servlet context by specifying unbind = "-". Finally, to target the servlet context to your app's OSGi bundle, specify as the target value the bundle's symbolic name–it's the value you specified for Bundle-SymbolicName in your bnd.bnd file.

  For example, here's the setServletContext(ServletContext) method from the Form Nav Extension portlet's entry class:

  ```
  @Override
  @Reference(
      target = "(osgi.web.symbolicname=com.liferay.docs.formnavextensionportlet)",
      unbind = "-"
  )
  public void setServletContext(ServletContext servletContext) {
      super.setServletContext(servletContext);
  }
  ```

The above method calls its parent's setServletContext(ServletContext) method. But look at what its @Reference annotation does. It unbinds the servlet context from its current binding and instead targets it to its own bundle–it targets the bundle symbolically named com.liferay.docs.formnavextensionportlet. That is the exact symbolic name defined by Bundle-SymbolicName: com.liferay.docs.formnavextensionportlet in the Form Nav Extension portlet's bnd.bnd file.

You've learned what's required to create a section class. You declared your class to extend the BaseJSPFormNavigatorEntry class and implement the FormNavigatorEntry interface, both with respect to your Form Navigator's form model bean class. Using annotations, you registered your entry class as an OSGi service. Then you implemented all the entry methods to relate your entry to a Form Navigator, category, and JSP, populate your entry's request object, and target the servlet context to your bundle.

If you're curious about what a working entry implementation looks like, check out the example entry class next.

Example Form Navigator Entry Class

Inspecting an example implementation can help you work out details in your implementation. Here's the Form Nav Extension portlet's entry class MyAppCompanySettingsFormNavigatorEntry:

```
package com.liferay.docs.formnavextensionportlet;

import java.io.IOException;
import java.util.Locale;
import java.util.ResourceBundle;

import javax.portlet.PortletPreferences;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

import com.liferay.portal.kernel.servlet.taglib.ui.BaseJSPFormNavigatorEntry;
```

```
import com.liferay.portal.kernel.servlet.taglib.ui.FormNavigatorConstants;
import com.liferay.portal.kernel.servlet.taglib.ui.FormNavigatorEntry;
import com.liferay.portal.kernel.util.PrefsParamUtil;
import com.liferay.portal.kernel.util.PrefsPropsUtil;
import com.liferay.portal.kernel.util.ResourceBundleUtil;
import com.liferay.portal.kernel.util.WebKeys;
import com.liferay.portal.model.Company;
import com.liferay.portal.theme.ThemeDisplay;

@Component(
    immediate = true,
    property = {
        "form.navigator.entry.order:Integer=71"
    },
    service = FormNavigatorEntry.class
)
public class MyAppCompanySettingsFormNavigatorEntry
    extends BaseJSPFormNavigatorEntry<Company>
        implements FormNavigatorEntry<Company> {

    @Override
    public String getCategoryKey() {
        return FormNavigatorConstants.CATEGORY_KEY_COMPANY_SETTINGS_MISCELLANEOUS;
    }

    @Override
    public String getFormNavigatorId() {
        return FormNavigatorConstants.FORM_NAVIGATOR_ID_COMPANY_SETTINGS;
    }

    @Override
    protected String getJspPath() {
        return "/portal_settings/my_app.jsp";
    }

    @Override
    public String getKey() {
        return "my-app";
    }

    @Override
    public String getLabel(Locale locale) {
        ResourceBundle resourceBundle = ResourceBundleUtil.getBundle(
            "content.Language", locale, getClass());

        return resourceBundle.getString("my-app");
    }

    @Override
    public void include(HttpServletRequest request, HttpServletResponse response)
            throws IOException {

        ThemeDisplay themeDisplay =
            (ThemeDisplay)request.getAttribute(WebKeys.THEME_DISPLAY);

        PortletPreferences companyPortletPreferences =
            PrefsPropsUtil.getPreferences(themeDisplay.getCompanyId(), true);

        boolean companyMyAppFeatureEnabled =
            PrefsParamUtil.getBoolean(
                companyPortletPreferences, request, "myAppFeatureEnabled",
                true);

        request.setAttribute(
            MyAppWebKeys.COMPANY_MY_APP_FEATURE_ENABLED,
            companyMyAppFeatureEnabled);

        super.include(request, response);
    }

    @Override
    @Reference(
        target = "(osgi.web.symbolicname=com.liferay.docs.formnavextensionportlet)",
        unbind = "-"
    )
    public void setServletContext(ServletContext servletContext) {
        super.setServletContext(servletContext);
    }

}
```

The above class is declared an OSGi component that provides a `FormNavigatorEntry.class` service. Since the entry adds a JSP to Portal Settings, the class extends `BaseJSPFormNavigatorEntry` and implements `FormNavigatorEntry` on the `Company` model bean class. The class specifies that the entry belongs to the *Miscellaneous* portal settings category, by returning navigator key `FormNavigatorConstants.FORM_NAVIGATOR_ID_COMPANY_SETTINGS` from method `getFormNavigatorId` and category key `FormNavigatorConstants.CATEGORY_KEY_COMPANY_SETTINGS_MISCELLANEOUS` from method `getCategoryKey`. The entry's method `getKey` returns *my-app* as its own key and method `getJspPath` maps the class to the entry's JSP by returning its JSP file path `/portal_settings/my_app.jsp`.

The entry's `include` method retrieves a boolean portlet preference variable `myAppFeatureEnabled` that specifies whether My App's feature is enabled for the portal. It then sets the preference's value as an attribute on the request. The Form Nav Portlet's language keys for the entry's name, input screen title, and input label are defined in its `src/main/content/Language.properties` file. The method `getLabel(Locale)` uses language key `my-app` to return the entry's localized label. In summary, the `MyAppCompanySettingsFormNavigatorEntry` class meets all of the Form Navigation framework's section entry requirements.

There you have it! You learned all the parts of the Form Navigator framework and worked through all the steps to implement new categories and sections. To recap, you prepared your project's `bnd.bnd` file to support form navigator extension, created JSPs for your new form sections, identified the targeted form navigator and categories, created new categories, and created new entry implementations. You did it all! You now know what it takes to extend Liferay Form Navigators.

Related Topics

Portlets

Service Builder

104.4 Creating Form Navigator Contexts

Form Navigator System Settings let you specify what categories and sections are visible in your forms. You can learn how to set Form Navigator System Settings in Configuring Form Navigator Forms. Form Navigator configurations let you specify an optional context which defines the circumstances for which the configuration is applied. The following Form Navigator contexts are available by default:

- `add`: Denotes that the form is viewed when new content is being created. For example, you could use the `add` context to specify the visible form sections when someone creates a new site.
- `update`: Denotes that the form is viewed when content is being edited. For example, you could use the `update` context to specify the visible form sections when someone edits a site.

Although the default contexts cover most use cases, you may want to provide additional contexts for users. For example, you may want a custom configuration for a form when it's viewed by an administrator. This tutorial covers how to create additional contexts for Form Navigators.

Your first step is to manage the dependencies.

Adding the Form Navigator Dependency

Open your module's `build.gradle` file and add the following dependency:

```
dependencies {
    compileOnly group: "com.liferay.portal", name:
    "com.liferay.frontend.taglib.form.navigator", version: "1.0.2"
}
```

Now that you have the Form Navigator taglib dependency added, you can create the component class.

Implementing the Context Provider Class

Follow these steps to create a Form Navigator Context:

1. Create a component class that implements the `FormNavigatorContextProvider` service:

   ```
   @Component(
       service = FormNavigatorContextProvider.class
   )
   ```

2. Set the `FormNavigatorContextConstants.FORM_NAVIGATOR_ID` property to the form associated with the context. The example configuration below specifies the Users form, using a constant provided by the `com.liferay.portal.kernel.servlet.taglib.ui.FormNavigatorConstants` class:

   ```
   @Component(
       property = FormNavigatorContextConstants.FORM_NAVIGATOR_ID + "=" +
     FormNavigatorConstants.FORM_NAVIGATOR_ID_USERS,
       service = FormNavigatorContextProvider.class
   )
   ```

3. Implement the `FormNavigatorContextProvider` interface, specifying the Form Navigator's model bean class as a generic type. The example below sets the Users form's `User` model bean class as the generic type:

   ```
   public class UsersFormNavigatorContextProvider
       implements FormNavigatorContextProvider<User> {
       ...
     }
   ```

   You can determine the model bean class from the name of the ID's constant in `FormNavigatorConstants`. The word(s) right after `FORM_NAVIGATOR_ID_` in the constant's name hints at the class type. If you can access the Form Navigator's JSP source code, you can find the model bean from the `form-navigator` tag's `formModelBean` attribute value. You can locate the constant in the Liferay Portal repository using the following pattern: `id="<%= FormNavigatorConstants.[CONSTANT]`.

   For example, if the form navigator's ID is `FORM_NAVIGATOR_ID_USERS`, then you would search for `id="<%= FormNavigatorConstants.FORM_NAVIGATOR_ID_USERS`; Make sure that the `form-navigator` tag's `formModelBean` attribute's value isn't a reference to another class. For example, the web content form navigator's `formModelBean` attribute value is `article`, but upon further inspection, it's clear that `article` is a reference variable to the true model bean class: `JournalArticle`.

4. Override the *ContextProvider's getContext() method. This method also takes the Form Navigator's model bean class as a generic type and defines the logic for the new context. The example configuration below creates a new context called my.account for the Users form:

   ```
   @Override
   public String getContext(User selectedUser) {
       if (PortletKeys.MY_ACCOUNT.equals(_getPortletName())) {
           return "my.account";
       }

       if (selectedUser == null) {
           return FormNavigatorContextConstants.CONTEXT_ADD;
       }

       return FormNavigatorContextConstants.CONTEXT_UPDATE;
   }
   ```

   If you are viewing the Users form from the com_liferay_my_account_web_portlet_MyAccountPortlet portlet, the my.account context is returned. You also need to specify under what circumstances the add and update contexts are returned. In the example above, if the user doesn't exist (i.e. you are creating a new user), the add context is returned. Otherwise it defaults to the update context (edit view).

The new context is ready to use in your Form Navigator configurations.
A full *ContextProvider example class is provided next.

Context Provider Example class

Below is an example configuration for the com.liferay.users.admin.web.servlet.taglib.ui.UsersFormNavigatorContextProvic class:

```
package com.liferay.users.admin.web.servlet.taglib.ui;

import com.liferay.frontend.taglib.form.navigator.constants.FormNavigatorContextConstants;
import com.liferay.frontend.taglib.form.navigator.context.FormNavigatorContextProvider;
import com.liferay.portal.kernel.model.User;
import com.liferay.portal.kernel.service.ServiceContext;
import com.liferay.portal.kernel.service.ServiceContextThreadLocal;
import com.liferay.portal.kernel.servlet.taglib.ui.FormNavigatorConstants;
import com.liferay.portal.kernel.theme.PortletDisplay;
import com.liferay.portal.kernel.theme.ThemeDisplay;
import com.liferay.portal.kernel.util.PortletKeys;

import org.osgi.service.component.annotations.Component;

/**
 * @author Alejandro Tardín
 */
@Component(
    property = FormNavigatorContextConstants.FORM_NAVIGATOR_ID + "=" + FormNavigatorConstants.FORM_NAVIGATOR_ID_USERS,
    service = FormNavigatorContextProvider.class
)
public class UsersFormNavigatorContextProvider
    implements FormNavigatorContextProvider<User> {

    @Override
    public String getContext(User selectedUser) {
        if (PortletKeys.MY_ACCOUNT.equals(_getPortletName())) {
            return "my.account";
        }

        if (selectedUser == null) {
```

```
            return FormNavigatorContextConstants.CONTEXT_ADD;
        }

        return FormNavigatorContextConstants.CONTEXT_UPDATE;
    }

    private String _getPortletName() {
        ServiceContext serviceContext =
            ServiceContextThreadLocal.getServiceContext();

        ThemeDisplay themeDisplay = serviceContext.getThemeDisplay();

        PortletDisplay portletDisplay = themeDisplay.getPortletDisplay();

        return portletDisplay.getPortletName();
    }

}
```

Now you know how to create a Form Navigator context!

Related Topics

Configuring Form Navigator Forms

Form Navigator Extensions

Chapter 105

Themes and Layout Templates

Do you want to transform the look and feel of your Liferay DXP? Create your own user interface with a Liferay Theme! Create a layout template to specify where content can be placed on a page! Define a custom look and feel for your portlets! All these customizations and more are possible with Liferay DXP!

In this section of tutorials, you'll learn how to develop themes and layout templates, customize portlets, and more.

Chapter 106

Themes

A Liferay Theme is the overall look and feel for a site. Themes are a combination of CSS, JavaScript, HTML, and FreeMarker templates. Although the default themes are nice, you may wish to create your own custom look and feel for your site. Liferay DXP provides several tools and environments that you can use to create themes:

- Theme Builder Gradle Plugin
- The Liferay Theme Generator
- @ide@
- Blade CLI's Theme Template

What if you only wish to make a minor change to the overall look and feel? Let's say you just want to change a menu animation. Instead of creating an entire theme for this single modification, you can create a Themelet. Themelets are modular, customizable, reusable, shareable pieces of code that extend a theme. They enable reusable code for themes. Instead of rewriting the code each time, you can use the same themelet in each theme.

Liferay has its own set of base themes, called styled and unstyled that create the default look and feel you see at first start. The *styled* theme inherits from the *unstyled* theme, and simply adds some additional styling on top. These same base themes are used to create a custom theme. See the User Profile Theme, which uses the *styled* theme as its base. Using a base theme as your foundation, you can then make your customizations to the theme files. To modify the theme, mirror the folder structure of the files you wish to change and copy them into your theme. Place the modified files in the `src` folder of your theme if using the Liferay Theme Generator, or copy them into the `webapp` folder of your theme if using Liferay @ide@. Build the theme to apply the changes.

Once your theme is developed it is packaged as a WAR (Web application ARchive) file and can be deployed to the server, either manually or using build tools. Apply your theme to your pages through the Look and Feel menu. The only limitation is your imagination.

106.1 Liferay Theme Generator

The Liferay Theme Generator is an easy-to-use command-line wizard that streamlines the theme creation process. It is independent of the Liferay Plugins SDK, and generates themes for Liferay Portal 6.2, 7.0, and up. It is just one of Liferay JS Theme Toolkit. This tutorial focuses on using the Liferay Theme Generator to create themes. In just a few steps, you'll have a working Liferay theme.

Note: The Liferay Theme Generator is unsupported. The tool is still in development and is not guaranteed to work on all platforms and environments.

This tutorial demonstrates how to:

- Install the Liferay Theme Generator
- Generate a theme

The first step is to install the Liferay Theme Generator.

Installing the Theme Generator

The Liferay Theme Generator has several dependencies. Follow these steps to install them:

1. Install Node.js, if it's not already installed. We recommend installing v6.6.0, which is the version Liferay Portal 7.0 supports.

   To test your Node.js installation, execute the following command:

   ```
   node -v
   ```

   The resulting output should look similar to this:

   ```
   v4.2.2
   ```

   Note: To avoid any potential compatibility issues, we recommend installing the Long Term Support (LTS) version of Node.js–at the time of this writing, the LTS version is v4.2.2.

   Node Package Manager (npm) is installed along with Node.js.

2. Set up your npm environment.

   First, create an `.npmrc` file in your user's home directory. This helps you bypass npm permission-related issues.

   In the `.npmrc` file, specify a `prefix` property like this one:

   ```
   prefix=/Users/[username]/.npm-packages
   ```

   Set the `prefix` value based on your user's home directory. The location you specify is where global npm packages are to be installed.

   Next, set the `NPM_PACKAGES` system environment variable to the `prefix` value you just specified:

   ```
   NPM_PACKAGES=/Users/[username]/.npm-packages (same as prefix value)
   ```

   As a last npm configuration, since npm installs Yeoman and gulp executables to `${NPM_PACKAGES}/bin` on UNIX and to `%NPM_PACKAGES%` on Windows, make sure to add the appropriate directory to your system path. For example, on UNIX you'd set this:

```
PATH=${PATH}:${NPM_PACKAGES}/bin
```

3. Install Yeoman and gulp globally by executing the following command:

   ```
   npm install -g yo gulp
   ```

   You've completed installing the Liferay Theme Generator's dependencies.

4. Install the Liferay Theme Generator. 7.x.x versions of the Theme Generator create themes for 6.2 and 7.0. 8.x.x versions create themes for 7.0 and 7.1. 9.x.x versions create themes for 7.2 and up.

   To create themes for **6.2** or **7.0**, run the following command:

   ```
   npm install -g generator-liferay-theme@7.x.x
   ```

   To create themes for **7.0** or **7.1** versions of Liferay DXP, run this command:

   ```
   npm install -g generator-liferay-theme@8.x.x
   ```

If you are on Windows, you must do additional setup for generated themes to use Sass.

Installing Sass on Windows

To use Sass on Windows, you must use either Sass from node-sass or Sass from Ruby. By default, the generator creates theme projects to use node-sass; but you can reconfigure them to use Ruby based Sass and Compass. Since node-sass indirectly requires Visual Studio, developers who are not already using Visual Studio may opt to use Ruby based Sass and Compass instead of node-sass.

Note: If your theme was built with an older version of the Liferay Theme Generator and specifies the `"liferay-theme-deps-7.0": "1.0.0"` dependency in its `package.json`, `npm install` may fail in Liferay DXP due to its dependency on node-sass `v3.13.1`. To fix this issue, change the dev dependency in your `package.json` to `"liferay-theme-deps-7.0": "7.0.0"` and rebuild your theme. We recommend that you use the latest 7.x.x version of the Theme Generator and dependencies in your `package.json` to ensure full compatibility.

This section explains both Sass installations.

Installing Sass from node-sass By default, the generator uses Sass from node-sass. node-sass requires node-gyp, which in turn requires Python and Visual Studio. The node-gyp installation instructions explain how to set up node-gyp, Python, and Visual Studio. Since Visual Studio is a particularly large dependency, if you aren't already using Visual Studio you might want to consider using Ruby Sass instead of node-sass.

Installing Ruby Sass and Compass As an alternative to using Sass from node-sass, you can use Sass from Ruby. Liferay themes require using Compass along with Ruby based Sass. In order to install and use Sass and Compass, you must install Ruby via the Ruby installer.

Note: Sass version 3.5 has compatibility issues with Compass. We recommend that you install Sass version 3.4.0 as shown in the command below.

The following commands install the Sass and Compass gems:

```
gem install sass -v "=3.4.0"

gem install compass
```

After creating a theme project in the next section, you'll learn how to configure a project to use Ruby based Sass and Compass.

Now that you've installed the Liferay Theme Generator and theme dependencies, you can generate a theme.

Running the Liferay Theme Generator

When you installed the Liferay Theme Generator, you also installed three sub-generators with it: a layout template creator, a themelet creator, and a theme importer. For the purposes of this tutorial, the focus will be on the default Liferay Theme Generator.

Note: If you run into permissions issues during theme generation, make sure you have read/write access to all folders on your system.

From a directory in which you want to create a theme, run the following command:

```
yo liferay-theme
```

Note: Some theme options are deprecated for 7.0 (such as Velocity theme templates). To view the deprecated options, run the generator with the `--deprecated` flag:

```
 yo liferay-theme --deprecated
```

The Liferay Theme Generator prompts you for the following things:

1. Enter a name for your theme.

2. Enter a theme ID, or press *enter* to accept the default.

3. Select the version of your Liferay instance (e.g., 7.0).

4. Choose a template language. Note that Freemarker is used by default; you can only select Velocity if you run the generator with the `--deprecated` flag.

   Based on the inputs up to this point, the default generator starts installing additional required software and creating your theme project.

5. When prompted, enter your app server's path. The information you provide is added to the `liferay-theme.json` in your theme's root folder. you can change the path in that file if you change app servers.

6. Finally, enter your Liferay instance's URL, or press *enter* to accept the default `localhost:8080`.

The generator creates a new theme project in your current directory. The theme inherits styles from the liferay-theme-styled theme. Note that you can switch to using a different base theme by executing the `gulp extend` command.

Note: The liferay-theme-styled and liferay-theme-unstyled themes are base themes. They're analogous to Java APIs. Liferay's Classic theme and other themes that use liferay-theme-styled or liferay-theme-unstyled as a base theme are analogous to API implementations–they're not meant to be extended. Extending Liferay's Classic theme is strongly discouraged.

Important: By default, your theme is based off of the styled theme and uses lib-sass/bourbon, instead of Compass. If, however, you are on Windows and are using the Ruby version of Sass, you must configure the theme to support Compass. To do so, follow these steps:

1. Open the `package.json` file found in the root folder of your theme, and locate the `rubySass` property and change it from `false` to `true`.

   Now that your theme is set to support Compass, you must install the Ruby Sass middleware and save it as a dependency for your theme.

2. Run the following command to install the Ruby Sass middleware:

   ```
   npm i --save gulp-ruby-sass
   ```

   The `--save` flag adds Ruby Sass to the list of dependencies in your theme's `package.json` file. Your theme is ready to use.

3. Run the `gulp build` task to generate the base files for your theme. Open the `build` folder of your theme to view the base files.

There you have it! You now have a working theme. At the moment, the theme is a bit bare bones, but you have everything you need to develop it.

The generated theme's structure differs slightly from a theme created in the Liferay Plugins SDK. In a theme generated in the Liferay Plugins SDK, you put changes in a `_diffs` folder. In a theme generated by the Liferay Theme Generator, you put changes in the `src` folder.

To develop your theme, copy the build files into your `src` folder. For instance to make a change to the `portal_normal.ftl` theme template, you would create a `templates` folder in your `src` folder, and copy the `portal_normal.ftl` file from the `build/templates` folder into the `src/templates` folder. This gives you the base template to build on. It is important that you mirror the folder structure in order for the changes to be applied.

Another noticeable difference in the generator created theme is that all CSS files have been converted to Sass SCSS files. Sassy CSS (SCSS) is the new main syntax which allows you to use the latest CSS3 styles and leverage Sass syntax advantages, such as nesting and variables.

To deploy your theme to your configured Liferay DXP instance, execute this command:

```
gulp deploy
```

You can apply your theme by following the instructions found in the Creating and Managing Pages User Guide.

Note: By default theme images are cached by the browser. If you need to update images in the theme, it is best practice to use versioning in the image URL. For example, `background-image:url("../images/image.jpg?v=1")` . You can then just update the version each time you update the image, which will remove the potential for any caching issues.

Now that you've created a theme and deployed it, you can use the theme project's gulp tasks to further develop and manage your theme. These offer basic functions, such as `build` and `deploy`, as well as more complex interactions, such as auto deploying when a change is made and setting the base theme.

There you have it! You're ready to design a terrific theme!

Related Topics

Theme Gulp Tasks

Themelets

Importing Resources with a Theme

106.2 Themelets

Themelets are small, extendable, and reusable pieces of code. Whereas themes require multiple components, a themelet only requires the files you wish to extend. This creates a more modular approach to theme design, that lends itself well to collaboration, and reduces the need for duplicated code in your theme.

Themelets let developers easily share code snippets across their themes with other developers. A themelet can consist of CSS and JavaScript. Themelets **do not support** theme templates.

Themelets are very flexible, and therefore they have a number of possible uses. You can make a themelet to modify the appearance of 7.0 admin tools, or a themelet that uses a custom JavaScript component for responsive embedded videos. For example, take a look at the Liferay Product Menu Animation Themelet. This themelet simply alters the animation for Liferay's Product Menu.

If there is something you find yourself coding over and over again for themes, it's a good candidate for a themelet.

This tutorial demonstrates how to:

- Create a themelet to extend your theme
- Install a Themelet

To create a themelet, you need a theme to extend and the Liferay Theme Generator and dependencies installed, as explained in the Liferay Theme Generator tutorial.

Creating a Themelet

Follow these steps to create a themelet:

1. Open the Command Line and navigate to the directory you want to create your themelet in.
2. Run `yo liferay-theme:themelet` and follow the prompts to generate the themelet.
3. The generated themelet contains a `package.json` file with configuration information and a `src/css` folder that contains a `_custom.scss` file. Just like a theme, add your CSS changes to the `src/css` folder, and add your JavaScript changes to the `src/js` folder.
4. To use your themelet, you must install it globally first. This makes the themelet visible to the generator. To install your themelet globally, navigate into its root folder and run `npm link`. Note, you may need to run the command using `sudo npm link`. This creates a globally-installed symbolic link for the themelet in your npm packages folder. Now your themelet is available to install in your themes.

Now that your themelet is developed, you can install it in your theme.

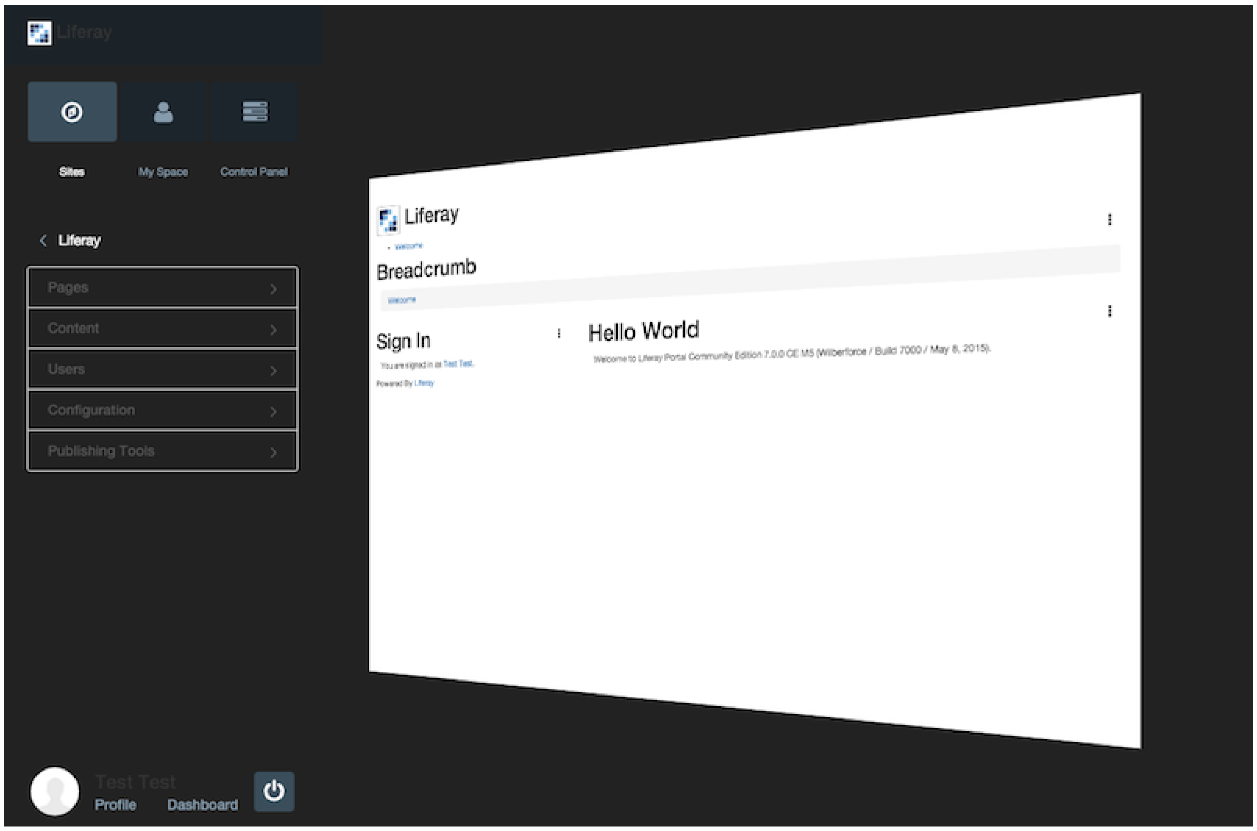
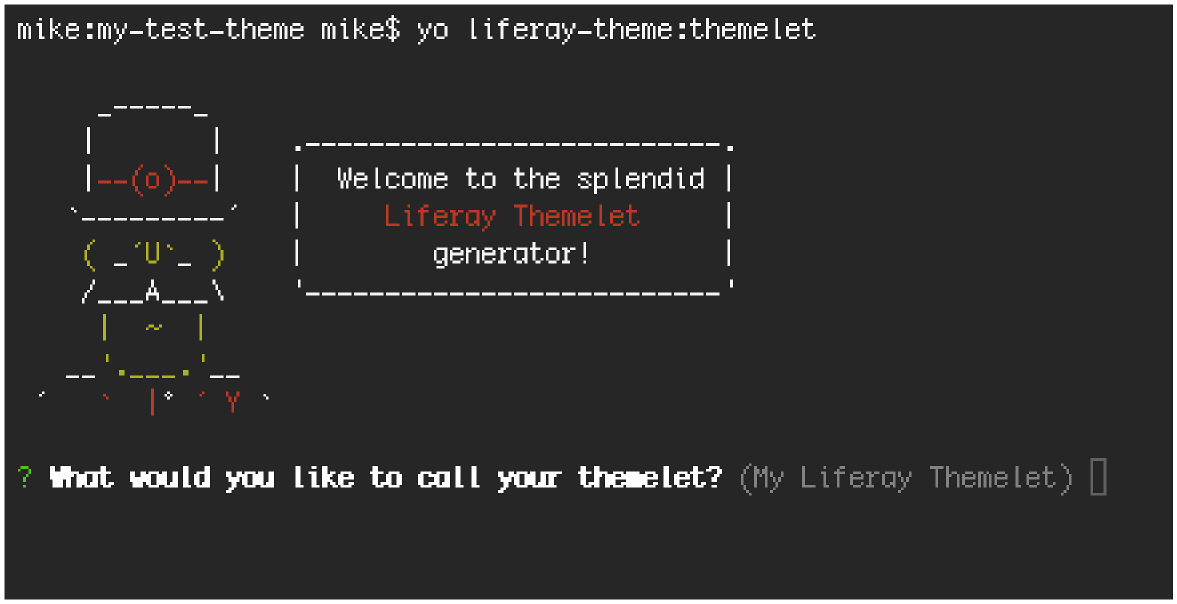


Figure 106.1: Themelets can be used to modify one aspect of the UI, that you can then reuse in your other themes.



```
mike:my-test-theme mike$ yo liferay-theme:themelet

      _-----_
     |       |    .--------------------------.
     |--(o)--|    |  Welcome to the splendid |
    `---------´   |     Liferay Themelet     |
     ( _´U`_ )    |        generator!        |
     /___A___\    '--------------------------'
      |  ~  |
    __'.___.'__
  ´   `  |° ´ Y `

? What would you like to call your themelet? (My Liferay Themelet)
```

Figure 106.2: The Themelet sub-generator automates the themelet creation process, making it quick and easy.

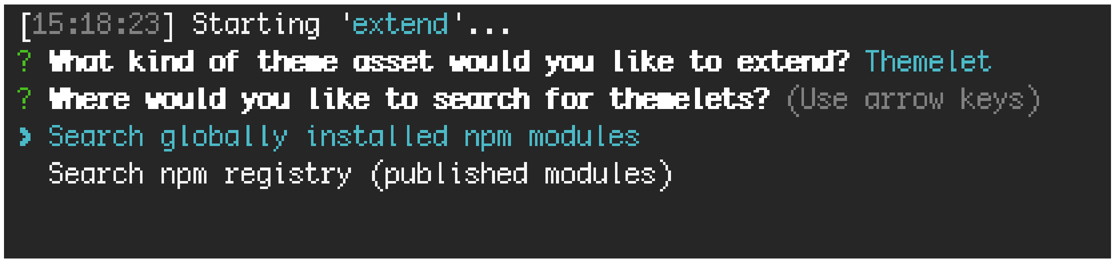
Installing a Themelet

After you've developed your themelet, follow the steps below to install it into your theme.

1. Navigate to your theme's root directory and run the following command:

   ```
   gulp extend
   ```

2. Choose *Themelet* as the theme asset to extend.
3. Select *Search globally installed npm modules*.



```
[15:18:23] Starting 'extend'...
? What kind of theme asset would you like to extend? Themelet
? Where would you like to search for themelets? (Use arrow keys)
❯ Search globally installed npm modules
  Search npm registry (published modules)
```

Figure 106.3: You can extend your theme using globally installed npm modules or published npm modules.

4. Highlight your themelet, press spacebar to activate it, and press *Enter* to install it.
5. Run `gulp deploy` to build and deploy your theme with the new themelet updates.

Your themelet is installed! As you can see, themelets are a handy tool to add to your theme development bag o' tricks.

Related Topics

Importing Resources with Your Themes

Liferay Theme Generator

106.3 Importing Resources with a Theme

A theme without content is like an empty house. If you're trying to sell an empty house, it may be difficult for prospective buyers to see its full beauty. However, staging the house with some furniture and decorations helps prospective buyers imagine what the house might look like with their belongings. Liferay's resources importer module is a tool that allows a theme developer to have files and web content automatically imported into Liferay DXP when a theme is deployed. Usually, the resources are imported into a site template but they can also be imported directly into a site. Liferay Administrators can use the site or site template created by the resources importer to showcase the theme. This is a great way for theme developers to provide a sample context that optimizes the design of their theme. In fact, all standalone themes that are uploaded to Liferay Marketplace must use the resources importer. This ensures a uniform experience for Marketplace users: a user can download a theme from Marketplace, install it on Liferay DXP, go to Sites or Site Templates in the Control Panel and immediately see their new theme in action. In this tutorial, we explain how to include resources with your theme.

Note: The resources importer has undergone some changes that affect the properties, class names, and structures that were referred to in versions prior to 7.0. Please read through the steps below to see the updates. In previous versions of Liferay, you had to deploy the resources importer if you declared it as a dependency in your theme's `liferay-plugin-package.properties` file. In 7.0 and up, this is no longer a requirement. The resources importer is now an OSGi module, and is deployed to your instance by default.

When you create a new theme using the Liferay Theme Generator, check your theme's `src/WEB-INF/liferay-plugin-package.properties` file for the developer mode entry:

```
resources-importer-developer-mode-enabled=true
```

This is a convenience feature for theme developers. With this setting enabled, importing resources to a site or site template that already exists, recreates the site or site template. Importing resources into a site template reapplies the site template and its resources to the sites that are based on the site template. Without `resources-importer-developer-mode-enabled=true`, you have to manually delete the sites or site templates built by the resources importer, each time you want to apply changes from your theme's `src/WEB-INF/src/resources-importer` folder.

Warning: the `resources-importer-developer-mode-enabled=true` setting can be dangerous since it involves *deleting* (and re-creating) the affected site or site template. It's only intended to be used during development. Never use it in production.

If you'd like to import your theme's resources directly into a site, instead of into a site template, you can specify the following in your `liferay-plugin-package.properties` file:

```
resources-importer-target-class-name=com.liferay.portal.kernel.model.Group

resources-importer-target-value=[site-name]
```

If you're using the `resources-importer-target-value=[site-name]` property, double check the site name that you're specifying. If you specify the wrong value, you could end up deleting (and re-creating) the wrong site!

Warning: It's safer to import theme resources into a site template than into an actual site. The `resources-importer-target-class-name=com.liferay.portal.kernel.model.Group` setting can be handy for development and testing but should be used cautiously. Don't use this setting in a theme that will be deployed to a production Liferay instance or a theme that will be submitted to Liferay Marketplace. To prepare a theme for deployment to a production Liferay instance, use the default setting so that the resources are imported into a site template. You can do this explicitly by setting `resources-importer-target-class-name=com.liferay.portal.kernel.model.LayoutSetPrototype` or implicitly by commenting out or removing the `resources-importer-target-class-name` property.

All of the resources a theme uses with the resources importer go in the `[theme-name]/src/WEB-INF/src/resources-importer` folder. The assets to be imported by your theme should be placed in the following directory structure:

- `[theme-name]/src/WEB-INF/src/resources-importer/`

    - `sitemap.json` - defines the pages, layout templates, and portlets
    - `assets.json` - (optional) specifies details on the assets

- document_library/
    * documents/ - contains documents and media files
- journal/
    * articles/ - contains web content (HTML) and folders grouping web content articles (XML) by template. Each folder name must match the file name of the corresponding template. For example, create folder Template 1/ to hold an article based on template file Template 1.ftl.
    * structures/ - contains structures (JSON) and folders of child structures. Each folder name must match the file name of the corresponding parent structure. For example, create folder Structure 1/ to hold a child of structure file Structure 1.json.
    * templates/ - groups templates (VM or FTL) into folders by structure. Each folder name must match the file name of the corresponding structure. For example, create folder Structure 1/ to hold a template for structure file Structure 1.json.

The following is the XML file for a basic web content article:

```
<?xml version="1.0"?>

<root available-locales="en_US" default-locale="en_US">
    <dynamic-element name="content" type="text_area" index-type="keyword" index="0">
        <dynamic-content language-id="en_US">
            <![CDATA[
                <center>
                <p><img alt="" src="[$FILE=space-program-history.jpg$]" /></p>
                </center>

                <p>In the mid-20th century, after two of the
                most violent wars in history, mankind turned
                its gaze upwards to the stars. Instead of
                continuing to strive against one another,
                man choose instead to strive against the
                limits that we had bound ourselves to. And
                so the Great Space Race began.</p>

                <p>At first the race was to reach space--get
                outside the earth's atmosphere, and when
                that had been reached, we shot for the moon.
                After sending men to the moon, robots to
                Mars, and probes beyond the reaches of our
                solar system, it seemed that there was
                nowhere left to go.</p>

                <p>The Space Program aims to change that.
                Beyond national boundaries, beyond what
                anyone can imagine that we can do. The sky
                is not the limit.</p>
            ]]>
        </dynamic-content>
    </dynamic-element>
</root>
```

You can view an article's XML by going to its source.

When you create a new theme using the Liferay Theme Generator, a default sitemap.json file is created and a default liferay-plugin-package.properties file is created in the WEB-INF folder.

You have two options for specifying resources to be imported with your theme. The recommended approach is to add resource files to the folders outlined above and to specify the contents of the site or site template in a `sitemap.json` file (described below). Alternatively, you can use an `archive.lar` file to package the resources you'd like your theme to deploy. To create such an `archive.lar`, just export the contents of a site from Liferay Portal using the site scope. Then place the `archive.lar` file in your theme's `[theme-name]/src/WEB-INF/src/resources-importer` folder. If you choose to use an archive file to package all of your resources, you won't need a `sitemap.json` file or any other files in your `[theme-name]/src/WEB-INF/src/resources-importer` folder. Note, however, a LAR file is version-specific; it won't work on any version of Liferay other than the one from which it was exported. For this reason, using a `sitemap.json` file to specify resources is the most flexible approach. If you're developing themes for Liferay Marketplace, you should use the `sitemap.json` to specify resources to be imported with your theme.

The `sitemap.json` in the `[theme-name]/src/WEB-INF/src/resources-importer` folder specifies the site pages, layout templates, web content, assets, and portlet configurations provided with the theme. This file describes the contents and hierarchy of the site for Liferay to import as a site or site template.

Note: Site templates only support the importing of either public page sets or private page sets.

If you want to import both public and private page sets, as shown in the example `sitemap.json` below, you must import your resources into a site.

Even if you're not familiar with JSON, the `sitemap.json` file is easy to understand. Let's examine a sample `sitemap.json` file:

```
{
"layoutTemplateId": "2_columns_ii",
"privatePages": [
    {
        "friendlyURL": "/private-page",
    "name": "Private Page",
    "title": "Private Page"
    }
],
"publicPages": [
        {
            "columns": [
                [
                    {
                        "portletId": "com_liferay_login_web_portlet_LoginPortlet"
                    },
                    {
                        "portletId": "com_liferay_site_navigation_menu_web_portlet_SiteNavigationMenuPortlet"
                    },
                    {
                        "portletId": "com_liferay_journal_content_web_portlet_JournalContentPortlet",
                        "portletPreferences": {
                            "articleId": "Without Border.html",
                            "groupId": "${groupId}",
                            "portletSetupPortletDecoratorId": "borderless"
                        }
                    },
                    {
                        "portletId": "com_liferay_journal_content_web_portlet_JournalContentPortlet",
                        "portletPreferences": {
                            "articleId": "Custom Title.html",
                            "groupId": "${groupId}",
                            "portletSetupPortletDecoratorId": "decorate",
                            "portletSetupTitle_en_US": "Web Content Display with Custom Title",
                            "portletSetupUseCustomTitle": "true"
                        }
                    }
                ],
                [
                    {
```

```
                "portletId": "com_liferay_hello_world_web_portlet_HelloWorldPortlet"
            },
            {
                "portletId": "com_liferay_site_navigation_menu_web_portlet_SiteNavigationMenuPortlet_INSTANCE_${groupId}",
                "portletPreferences": {
                    "displayStyle": "[custom]",
                    "headerType": "root-layout",
                    "includedLayouts": "all",
                    "nestedChildren": "1",
                    "rootLayoutLevel": "3",
                    "rootLayoutType": "relative"
                }
            },
            "Web Content with Image.html",
            {
                "portletId": "com_liferay_nested_portlets_web_portlet_NestedPortletsPortlet",
                "portletPreferences": {
                    "columns": [
                        [
                            {
                                "portletId": "com_liferay_journal_content_web_portlet_JournalContentPortlet",
                                "portletPreferences": {
                                "articleId": "Child Web Content 1.xml",
                                "groupId": "${groupId}",
                                "portletSetupPortletDecoratorId": "decorate",
                                "portletSetupTitle_en_US": "Web Content Display with Child Structure 1",
                                    "portletSetupUseCustomTitle": "true"
                                }
                            }
                        ],
                        [
                            {
                                "portletId": "com_liferay_journal_content_web_portlet_JournalContentPortlet",
                                "portletPreferences": {
                                "articleId": "Child Web Content 2.xml",
                                "groupId": "${groupId}",
                                "portletSetupPortletDecoratorId": "decorate",
                                "portletSetupTitle_en_US": "Web Content Display with Child Structure 2",
                                    "portletSetupUseCustomTitle": "true"
                                }
                            }
                        ]
                    ],
                    "layoutTemplateId": "2_columns_i"
                }
            }
        ]
    ],
    "friendlyURL": "/home",
    "nameMap": {
        "en_US": "Welcome",
        "fr_FR": "Bienvenue"
    },
    "title": "Welcome"
},
{
    "columns": [
        [
            {
                "portletId": "com_liferay_login_web_portlet_LoginPortlet"
            }
        ],
        [
            {
                "portletId": "com_liferay_hello_world_web_portlet_HelloWorldPortlet"
            }
        ]
    ],
            "friendlyURL": "/layout-prototypes-parent-page", "layouts": [
                {
                    "friendlyURL": "/layout-prototypes-page-1",
                    "layoutPrototypeLinkEnabled": "true",
                    "layoutPrototypeUuid": "371647ba-3649-4039-bfe6-ae32cf404737",
                    "name": "Layout Prototypes Page 1",
                    "title": "Layout Prototypes Page 1"
                },
                {
                    "friendlyURL": "/layout-prototypes-page-2",
                    "layoutPrototypeUuid": "c98067d0-fc10-9556-7364-238d39693bc4",
                    "name": "Layout Prototypes Page 2",
                    "title": "Layout Prototypes Page 2"
                }
            ],
            "name": "Layout Prototypes",
            "title": "Layout Prototypes"
        },
        {
            "columns": [
                [
                    {
                        "portletId": "com_liferay_login_web_portlet_LoginPortlet"
                    }
                ],
                [
                    {
                        "portletId": "com_liferay_hello_world_web_portlet_HelloWorldPortlet"
                    }
                ]
            ],
            "friendlyURL": "/parent-page",
            "layouts": [
                {
                    "friendlyURL": "/child-page-1",
                    "name": "Child Page 1",
                    "title": "Child Page 1"
                },
                {
                    "friendlyURL": "/child-page-2",
                    "name": "Child Page 2",
                    "title": "Child Page 2"
                }
            ],
            "name": "Parent Page",
            "title": "Parent Page"
        },
        {
            "friendlyURL": "/url-page",
            "name": "URL Page",
            "title": "URL Page",
            "type": "url"
        },
        {
            "friendlyURL": "/link-page",
            "name": "Link to another Page",
            "title": "Link to another Page",
            "type": "link_to_layout",
            "typeSettings": "linkToLayoutId=1"
        },
        {
            "friendlyURL": "/hidden-page",
            "name": "Hidden Page",
            "title": "Hidden Page",
            "hidden": "true"
        }
    ]
}
```

The first thing you should declare in your `sitemap.json` file is a default layout template ID so the target site or site template can reference the layout template to use for its pages. You can also specify different layout templates to use for individual pages. You can find layout templates in your Liferay installation's `/layouttpl` folder. Next, you have to declare the layouts, or pages, that your site template should use. Note that pages are called *layouts* in Liferay DXP's code. You can specify a name, title, and friendly URL for a page, and you can set a page to be hidden. To declare that web content should be displayed on a page, simply specify an XML file. You can declare portlets by specifying their portlet IDs, which can be found in the App Manager of the Control Panel. Select the suite that the App is located in, click the App, click the App web link, and open the *Portlets* tab that appears. The portlet ID is displayed below the name of the App. You can find a full list of the default portlet IDs for Liferay in the Portlet ID quick reference. You can also specify portlet preferences for each portlet.

The following properties are available in the `sitemap.json`:

colorSchemeId: Specifies a different color scheme (by ID) than the default color scheme to use for the layout.

columns: Specifies the column contents for the layout.

friendlyURL: Sets the layout's friendly URL.

hidden: Sets whether the layout is hidden.

layoutCss: Sets custom CSS for the layout to load after the theme.

layoutPrototypeLinkEnabled: Sets whether the layout inherits changes made to the page template (if the layout has one).

layoutPrototypeName: Specifies the page template (by name) to use for the layout. If this is defined, the page template's UUID is retrieved using the name, and `layoutPrototypeUuid` is not required.

layoutPrototypeUuid: Specifies the page template (by UUID) to use for the layout. If `layoutPrototypeName` is defined, this is not required.

layoutTemplateId: When defined outside the scope of a portlet, sets the default layout template for the theme's layouts. When placed inside a layout, sets the layout template for the layout.

layouts: Specifies child pages for a layout set (`publicPages` || `privatePages`).

name: The layout's name.

nameMap: Passes a name object with multiple name key/value pairs. As shown in the example sitemap above, you can use this to pass language keys for layout names.

portletPreferences: Specifies the portlet's preferences. See note below for more information.

portletSetupPortletDecoratorId: Specifies the portlet decorator to use for the portlet (`borderless` || `barebone` || `decorate`). See the Portlet Decorators tutorial for more info.

portlets: specifies the portlets to display in the layout's column. To nest portlets, recursively use columns as shown in the example `sitemap.json` above for the `com_liferay_nested_portlets_web_portlet_NestedPortletsPortlet` portlet.

privatePages: Specifies private layouts.

publicPages: Specifies public layouts.

themeId: Specifies a different theme (by ID) than the default theme bundled with the `sitemap.json` to use for the layout.

title: The layout's title.

type: Sets the layout type. The default value is `portlet` (empty page). Possible values are `copy` (copy of a page of this site), `embedded`, `full_page_application`, `link_to_layout`, `node` (page set), `panel`, `portlet`, and `url` (link to URL).

typeSettings: Specifies settings (using key/value pairs) for the layout `type`.

Note: Portlet preferences set in `sitemap.json` are saved in the database to the column: `portletPreferences.preferences`. To determine the proper key:value pair for a portlet preference, there are a couple approaches you can take.

You can manually set the portlet preference in Liferay DXP, and then check the values in this column of the database as a hint for what to configure in your `sitemap.json`. For example, you can configure the Asset Publisher to display assets that match the specified asset tags, using the following configuration:

```
"queryName0": "assetTags",
"queryValues0": "MyAssetTagName"
```

Alternatively, you can search each app in your Liferay DXP bundle for the keyword `preferences--`. This returns some of the app's JSPs that have the portlet preferences defined for the portlet.

You can use a combination of both of these approaches to determine the key/value pairs for the portlet preferences.

Note that portlet preferences that require an existing configuration, such as a tag or category, may require you to create the configuration on the Global site first, so that the Resources Importer finds a match when deployed with the theme.

Tip: You can specify an application display template (ADT) for a portlet in the `sitemap.json` file by setting the `displayStyle` and `displayStyleGroupId` portlet preferences. For example:

```
"portletId": "com_liferay_asset_publisher_web_portlet_AssetPublisherPortlet",
    "portletPreferences": {
        "displayStyleGroupId": "10197",
        "displayStyle": "ddmTemplate_6fe4851b-53bc-4ca7-868a-c836982836f4",
}
```

To learn more about ADTs, visit the Styling Apps with Application Display Templates chapter.

Optionally, you can create an `assets.json` file in your `[theme-name]/src/WEB-INF/src/resources-importer` folder. While the `sitemap.json` file defines the pages of the site or site template to be imported, along with the layout templates, portlets, and portlet preferences of these pages, the `assets.json` file specifies details about the assets to be imported. Tags can be applied to any asset. Abstract summaries and small images can be applied to web content articles. For example, the following `assets.json` file specifies two tags for the `company_logo.png` image, one tag for the `Custom Title.xml` web content article, and an abstract summary and small image for the `Child Web Content 1.json` article structure:

```
{
    "assets": [
        {
            "name": "company_logo.png",
            "tags": [
                "logo",
                "company"
            ]
        },
        {
            "name": "Custom Title.xml",
            "tags": [
                "web content"
            ]
        },
        {
            "abstractSummary": "This is an abstract summary.",
            "name": "Child Web Content 1.xml",
            "smallImage": "company_logo.png"
        }
    ]
}
```

Now that you've learned about the directory structure for your resources, the `sitemap.json` file for referencing your resources, and the `assets.json` file for describing the assets of your resources, it's time to put resources into your theme. You can create resources from scratch and/or bring in resources that you've already created in Liferay. Let's go over how to leverage your HTML (basic web content), JSON (structures), or VM or FTL (templates) files from Liferay:

Note: In previous versions of Liferay, basic web content could be added without the need of a structure or template. In 7.0 and above, all web content articles require a structure and template.

- **web content:** Edit the article, and copy the content from the *Source* view. Create a folder for the article under `resources-importer/journal/articles/`, copy the contents into an XML file, named as desired, and place it into the folder for the article. The web content article's XML fills in the data required by the structure.
- **structure:** Edit the structure by clicking the link under *Structure and Template*, and copy and paste its contents into a new JSON file for the structure in the `resources-importer/journal/structures/` folder. The structure JSON sets a wireframe, or blueprint, for an article's data.
- **template:** Create a folder for the template under `resources-importer/journal/templates/`. Edit the template by clicking the link under *Structure and Template*, and copy and paste its contents into a new FTL file for the template, and place it into the folder for the template. The template defines how the data should be displayed.

Here is an outline of steps you can use in developing your theme and its resources:

1. Create your theme.
2. Add your resources under the `[theme-name]/src/WEB-INF/src/resources-importer` folder and its sub-folders.
3. Create a `sitemap.json` file in your `resources-importer/` folder. In this file, define the pages of the site or site template to be imported, along with the layout templates, portlets, and portlet preferences of these pages.
4. Create an `assets.json` file in your `resources-importer/` folder. In this file, specify details of your resource assets.
5. In your `liferay-plugin-package.properties` file, set `resources-importer-developer-mode-enabled=true`:

   ```
   resources-importer-developer-mode-enabled=true
   ```

6. Set the `resources-importer-target-value` property to the name of the site or site template into which you are importing, or comment it out to use the theme's name.

   For example, the following configuration sets the target value to the name of an existing site or site template:

   ```
   resources-importer-target-value = site/site template name
   ```

   Alternatively, this configuration uses the theme's name as the target value:

```
#resources-importer-target-value
```

```
**Note:** By default resources are imported into a new site template named
after the theme. If you want your resources to be imported into an existing
site or site template, you must specify a value for the
`resources-importer-target-value` property.
```

---

7. Comment out the `resources-importer-target-class-name` property to import into a site template or set it to `com.liferay.portal.kernel.model.Group` to import directly into a site.

   As mentioned above the example `sitemap.json`, **you must import your resources into a site, if you define both public and private page sets in your `sitemap.json`.**

If you don't specify a value for the `resources-importer-target-class-name` property, your resources will be imported into a site template.

8. Deploy your theme into your Liferay instance.

9. View your theme, and its resources, from within Liferay. Log in to your portal as an administrator and check the Sites or Site Templates section of the Control Panel to make sure that your resources were deployed correctly. From the Control Panel you can easily view your theme and its resources:

   - If you imported into a site template, select its *Actions* → *View Pages* to see it.
   - If you imported directly into a site, select its *Actions* → *Go to Public Pages* to see it.

You can go back to any of the beginning steps in this outline to make refinements. It's just that easy to develop a theme with resources intact!

Related Topics

Liferay Theme Generator

Styling Apps with Application Display Templates

106.4 Using Developer Mode with Themes

Do you want to develop Liferay DXP themes without having to redeploy to see your modifications? Use Liferay DXP's Developer Mode! In Developer Mode, all caches are removed, so any changes you make are visible right away. Also, you don't have to reboot the server as often in Developer Mode.

How does Developer Mode let you see your changes more quickly? By default, Liferay DXP is optimized for performance. Developer mode optimizes your configuration for development instead. Here is a list of Developer Mode's key behavior changes and the Portal Property override settings that trigger them (if applicable):

- CSS files are loaded individually rather than being combined and loaded as a single CSS file (`theme.css.fast.load=false`).
- Layout template caching is disabled (`layout.template.cache.enabled=false`).
- The server does not launch a browser when starting (`browser.launcher.url=`).

- FreeMarker Templates for themes and web content are not cached, so changes are applied immediately (via the system setting in your Liferay DXP instance).
- Minification of CSS and JavaScript resources is disabled (`minifier.enabled=false`).

Note: There are two known issues LPS-71350 and LPS-70364 that prevent CSS changes from being applied to the page and the Control Panel and Product Menu to break when using Developer Mode. Both these issues are fixed in Liferay Portal CE 7.0.3 GA4 and Liferay DXP 7.0 Fixpack DE 13.

Individual file loading of your styling and behaviors, combined with disabled caching for layout and FreeMarker templates, lets you see your changes more quickly.

These developer settings are defined in the `portal-developer.properties` file. To use these settings, you can include them in your `portal-ext.properties` file or copy them over to your `portal-ext.properties` file and override specific properties as needed. These configurations are covered in this tutorial.

First, you can explore how it's done in @ide@.

Setting Developer Mode for Your Server in @ide@

To enable Developer Mode for your server in @ide@, follow these steps:

1. Double-click on your server in the *Servers* window and open the *Liferay Launch* section.

2. Select *Custom Launch Settings* and check the *Use developer mode* option.

3. Save the changes and start your server.

Warning: Only change the Server settings from the runtime environment's Liferay Launch section.

For Liferay Portal servers below version 6.2 (e.g., Liferay v6.1 CE Server, Liferay v6.0 CE Server), @ide@ enables Developer Mode by default. When starting your Liferay DXP server for the first time, it creates a `portal-ext.properties` file in your server's directory. This properties file contains the property setting `include-and-override=portal-developer.properties`, which enables Developer Mode.

Most of the configuration is provided by the `portal-developer.properties` file, but you still have to configure the FreeMarker template setting. Follow the steps in the Configuring FreeMarker System Settings section to configure the FreeMarker template cache.

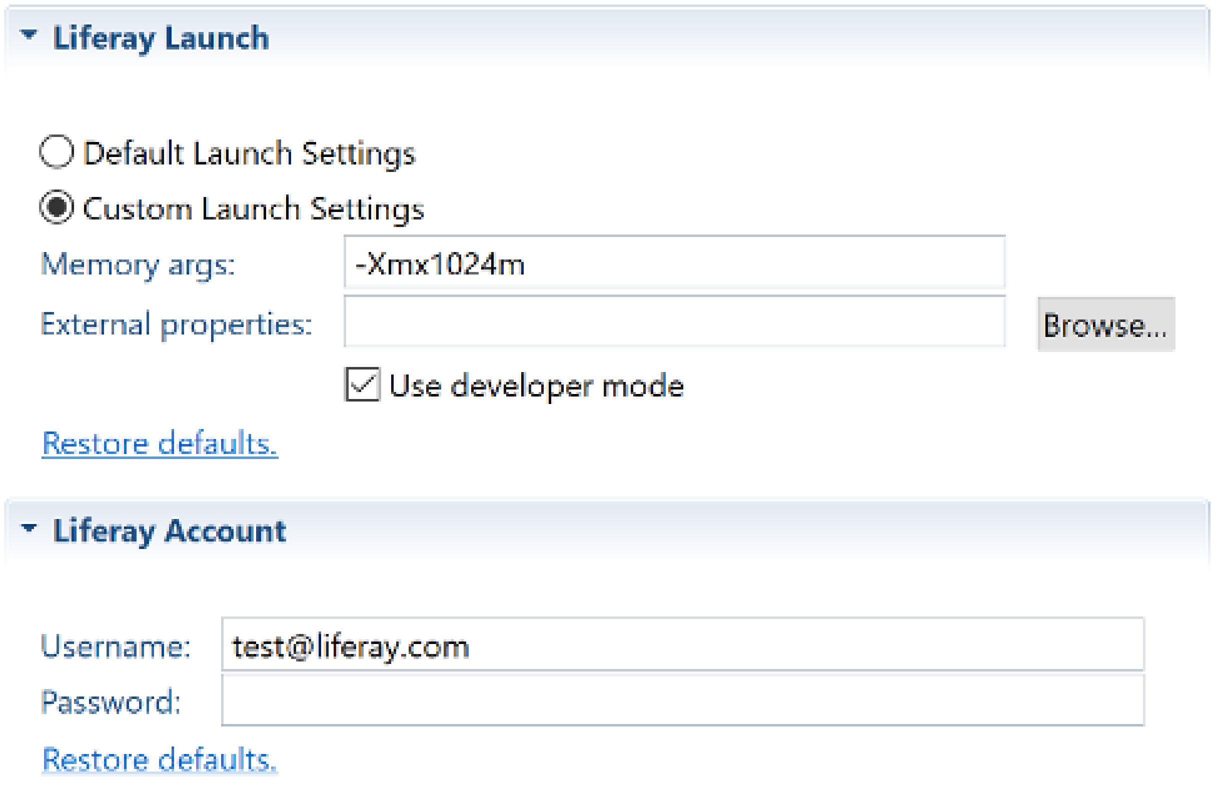
If you're not using @ide@, manual configuration for Developer Mode is covered next.

Setting Developer Mode for Your Server Using portal-developer.properties

To set Developer Mode manually, you must point to `portal-developer.properties` as shown in the last section. Add the `portal-ext.properties` file to the root folder of your app server's bundle and add the following line:

```
include-and-override=portal-developer.properties
```

Developer Mode is enabled upon starting your app server. `portal-developer.properties` provides the majority of the settings you'll need for smooth development. To disable the cache for FreeMarker templates, you must update the System Setting covered in the next section.


Liferay Launch

( ) Default Launch Settings
(●) Custom Launch Settings

Memory args: -Xmx1024m
External properties:
[✓] Use developer mode

Restore defaults.

Liferay Account

Username: test@liferay.com
Password:

Restore defaults.

Figure 106.4: The *Use developer mode* option lets you enable Developer Mode for your server in @ide@.

Configuring FreeMarker System Settings

FreeMarker Templates for themes and web content are cached by default. Therefore, any changes you make to your FreeMarker theme templates aren't immediately displayed. You can change this behavior through System Settings. Follow these steps:

1. Open the Control Panel and go to *Configuration* → *System Settings*.
2. Click the *Foundation* tab and select *FreeMarker Engine*.
3. By default, the *Resource modification check* (the time in milliseconds that the template is cached) is set to `60`. Set this value to `0` to disable caching.

Your FreeMarker templates are ready for development. Next you can learn how you can improve JavaScript file loading for development.

JavaScript Fast Loading

By default, JavaScript fast loading is enabled in Developer Mode (`javascript.fast.load=true`). This loads the packed version of files listed in the Portal Properties `javascript.barebone.files` or `javascript.everything.files`. You can, however, disable JavaScript fast loading for easier debugging for development. Just set `javascript.fast.load` to `false` in your `portal.properties`, or you can disable fast loading by setting the URL parameter `js_fast_load` to `0`.

Note: JavaScript fast loading is retrieved from one of three places: the request (determined by the current URL: `http://localhost:8080/web/guest/home?js_fast_load=1`(on) or `...?js_fast_load=0`(off)), the Session, or the Portal Property (`javascript.fast.load=true`). Liferay DXP gives preference in the order of request, session, and then Portal Properties. This lets you change `js_fast_load`'s value from the default in `portal.properties` without having to manually re-enter `js_fast_load` into the URL upon every new page load.

Great! You've set up your Liferay DXP server for Developer Mode. Now, when you modify your theme's file directly in your bundle, you can see your changes applied immediately on redeploying your theme!

Related Topics

Creating Layout Templates Manually

Creating Themes with @ide@

106.5 Theme Contributors

If you want to package UI resources independent of a specific theme and include them on a Liferay DXP page, Theme Contributors are your best option. If, instead, you'd like to include separate UI resources on a Liferay DXP page that are attached to a theme, you should look into themelets.

A Theme Contributor is a module that contains UI resources to use in Liferay DXP. Once a Theme Contributor is deployed to Liferay DXP, it's scanned for all valid CSS and JS files, and then its resources are included on the page. You can, therefore, style these UI components as you like, and the styles are applied for the current theme.

This tutorial demonstrates how to

- Identify a Theme Contributor module.
- Create a Theme Contributor module.

Next, you'll learn how to create a Theme Contributor.

Creating Theme Contributors

In Liferay versions prior to 7.0, the standard UI for User menus and navigation (the Dockbar) was included in the theme template. Starting in Liferay DXP 7.0, these standard UI components are packaged as Theme Contributors.

For example, the Control Menu, Product Menu, and Simulation Panel are packaged as Theme Contributor modules in Liferay, separating them from the theme. This means that these UI components must be handled outside the theme.

If you want to edit or style these standard UI components, you'll need to create your own Theme Contributor and add your modifications on top. You can also add new UI components to Liferay DXP by creating a Theme Contributor.

To create a Theme Contributor module, follow these steps:

1. Create a generic OSGi module using your favorite third party tool, or use Blade CLI. You can also use the Blade Template to create your module, in which case you can skip step 2.

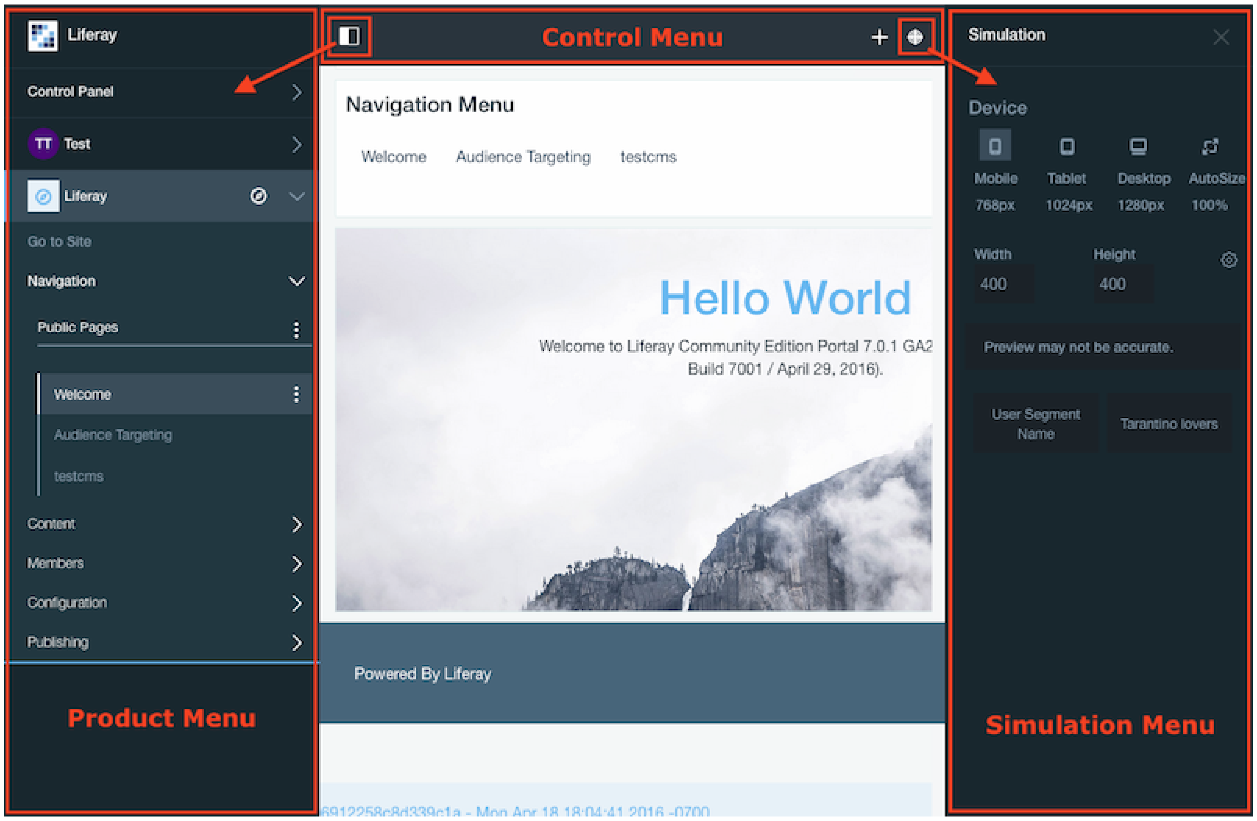


Figure 106.5: The Control Menu, Product Menu, and Simulation Panel are packaged as Theme Contributor modules.

2. To identify your module as a Theme Contributor, add the `Liferay-Theme-Contributor-Type` and `Web-ContextPath` headers to your module's `bnd.bnd` file. For example, see the Control Menu Theme Contributor module's `bnd.bnd`:

```
Bundle-Name: Liferay Product Navigation Control Menu Theme Contributor
Bundle-SymbolicName: com.liferay.product.navigation.control.menu.theme.contributor
Bundle-Version: 1.0.0
Liferay-Releng-Module-Group-Description:
Liferay-Releng-Module-Group-Title: Product Navigation
Liferay-Theme-Contributor-Type: product-navigation-control-menu
Web-ContextPath: /product-navigation-control-menu-theme-contributor
-include: ../../../../../marketplace/web-content-management/bnd.bnd
```

   The Theme Contributor type helps Liferay DXP better identify your module. For example, if you're creating a Theme Contributor to override an existing Theme Contributor, you should try to use the same type to maximize compatibility with future developments. The `Web-ContextPath` header sets the context from which the Theme Contributor's resources are hosted.

3. Because you'll often be overriding CSS of another Theme Contributor, you should load your CSS after theirs. You can do this by setting a weight for your Theme Contributor. In your `bnd.bnd` file, add the following header:

```
Liferay-Theme-Contributor-Weight: [value]
```

The higher the value, the higher the priority. If your Theme Contributor has a weight of 100, it will be loaded after one with a weight of 99, allowing your CSS to override theirs.

4. Create a `src/main/resources/META-INF/resources` folder in your module and place your resources (CSS and JS) in that folder.

5. Build and deploy your module to see your modifications applied to Liferay DXP pages and themes.

That's all you need to do to create a Theme Contributor for your site. Remember, with great power comes great responsibility, so use Theme Contributors wisely. The UI contributions affect every page and aren't affected by theme deployments.

Related Topics

Liferay Theme Generator
Themelets
Importing Resources with Your Themes
Theme Contributor Template

106.6 Context Contributors

JSP templates are the predominant templating framework in Liferay DXP. Themes, application display templates (ADTs), DDM templates, and more make use of JSPs as a templating engine. JSPs, however, are not the only templating language supported by Liferay DXP. Since many developers prefer other templating frameworks (e.g., FreeMarker and Velocity), Liferay enables you to use other template formats by offering the Context Contributors framework.

Because JSPs are "native" to Java EE, they have access to all the contextual objects inherent to the platform, like the request and the session. Through these objects, developers can normally obtain Liferay-specific context information by accessing container objects like `themeDisplay` or `serviceContext`.

Template formats like FreeMarker aren't native to Java EE, so they don't have access to these objects. If your template needs contextual information such as the current user, the page, or anything else, Java EE won't make it available to the template like it does for JSPs: you must inject it yourself into the template. Liferay, however, gives you a head start by injecting a `contextObjects` map of common variables (e.g., `themeDisplay`, `locale`, `user`, etc.) by default into FreeMarker templates (e.g., themes). This map is usually referred to as the *context* of a template. If you need to access some other context object that Liferay doesn't provide by default, you must modify or add to a template's context. To do that, you create a context contributor.

Context contributors modify a template's context by injecting variables and functionality usable by the template framework. This lets you use non-JSP templating languages for themes, ADTs, and any other templates used in Liferay DXP. For example, suppose you want your theme to change color based on the user's organization. You could create a context contributor to inject the user's organization to your theme's context, and then determine the theme's color based on that information.

Context contributors are already used in Liferay DXP by default. Liferay's Product Menu display is determined by a variable injected by a context contributor. You'll learn more about this later.

First, you'll learn how to create your own context contributor, and then you'll examine one example of how Liferay DXP uses context contributors.

1. Create a generic OSGi module using your favorite third party tool, or use Blade CLI.

2. Create a unique package name in the module's src directory and create a new Java class in that package. To follow naming conventions, begin the class name with the entity you want to inject context-specific variables for, followed by *TemplateContextContributor* (e.g., `ProductMenuTemplateContextContributor`).

3. Directly above the class's declaration, insert the following annotation:

   ```
   @Component(
       immediate = true,
       property = {"type=" + TemplateContextContributor.[Type of Contributor]},
       service = TemplateContextContributor.class
   )
   ```

   The `immediate` element instructs the module to start immediately once deployed to Liferay DXP. The `type` property should be set to one of the two fields defined in the TemplateContextContributor interface: `TYPE_GLOBAL` or `TYPE_THEME`. The `TYPE_THEME` field should be set if you only wish to inject context-specific variables for your theme; otherwise, setting the `TYPE_GLOBAL` field affects every context execution in Liferay DXP, like themes, ADTs, DDM templates, etc. Finally, your `service` element should be set to `TemplateContextContributor.class`.

   The `ProductMenuTemplateContextContributor` class's `@Component` annotation follows a similar layout:

   ```
   @Component(
       immediate = true,
       property = {"type=" + TemplateContextContributor.TYPE_THEME},
       service = TemplateContextContributor.class
   )
   ```

4. Implement the TemplateContextContributor interface in your `-TemplateContextContributor` class. This only requires implementing the `prepare(Map<String,Object>, HttpServletRequest)` method.

   Notice that the `prepare` method receives the `contextObjects` map as a parameter. This is your template's context that was described earlier. This method should be used to edit the context by injecting new or modified variables into the `contextObjects` map.

For a quick example of how you can implement the `TemplateContextContributor` interface to inject variables into a template's context, you'll examine the `ProductMenuTemplateContextContributor` class used by Liferay DXP by default. This class injects variables into Liferay's FreeMarker theme and determines whether the Product Menu is displayed in the current theme.

The `ProductMenuTemplateContextContributor` class implements the `prepare(...)` method, which injects a modified variable (`bodyCssClass`) and a new variable (`liferay_product_menu_state`) into the theme context:

```
@Override
public void prepare(
    Map<String, Object> contextObjects, HttpServletRequest request) {

    if (!isShowProductMenu(request)) {
        return;
    }

    String cssClass = GetterUtil.getString(
        contextObjects.get("bodyCssClass"));
    String productMenuState = SessionClicks.get(
        request,
        ProductNavigationProductMenuWebKeys.
            PRODUCT_NAVIGATION_PRODUCT_MENU_STATE,
        "closed");
```

```
    contextObjects.put(
        "bodyCssClass", cssClass + StringPool.SPACE + productMenuState);

        contextObjects.put("liferay_product_menu_state", productMenuState);
}
```

This method prepares the context contributor to inject variables into the theme to be used by the Product Menu. For this example, the `cssClass` and `productMenuState` variables are defined and then placed in the `contextObjects` map. By doing this, these variables have been injected into the theme context, making them accessible to the theme. Specifically, the `cssClass` variable provides styling for the Product Menu and the `productMenuState` variable determines whether the visible Product Menu should be open or closed.

Note: In previous versions of Liferay, if you needed to inject variables into themes, you were forced to create those variables in the `init.ftl` file of every theme. This forced theme developers to keep that logic updated in every theme version of every theme they developed. With context contributors, you can inject variables into existing frameworks without forcing theme developers to update their `init.ftl` files.

The `prepare` method above also determines whether to show the Product Menu or not with the following if statement:

```
if (!isShowProductMenu(request)) {
    return;
}
```

The `isShowProductMenu(...)` method injects functionality into the theme's context by providing an option to show/hide the Product Menu. This method is also included in the `ProductMenuTemplateContextContributor` class:

```
protected boolean isShowProductMenu(HttpServletRequest request) {
    ThemeDisplay themeDisplay = (ThemeDisplay)request.getAttribute(
        WebKeys.THEME_DISPLAY);

    if (themeDisplay.isImpersonated()) {
        return true;
    }

    if (!themeDisplay.isSignedIn()) {
        return false;
    }

    User user = themeDisplay.getUser();

    if (!user.isSetupComplete()) {
        return false;
    }

    return true;
}
```

The `ProductMenuTemplateContextContributor` provides an easy way to inject variables into Liferay DXP's theme directly related to the Product Menu. You can do the same with your custom context contributor. With the power to inject additional variables to any context in Liferay, you're free to fully harness the power of your chosen templating language.

Related Topics

Customizing the Product Menu
Liferay Theme Generator
Theme Contributors

106.7 Macros

Macros let you assign theme template fragments to a variable. This keeps your theme templates from becoming cluttered and makes them easier to read. Liferay DXP defines several FreeMarker macros in `FTL_liferay.ftl` template that you can use in your FreeMarker theme templates to include theme resources, standard portlets, and more. Likewise, Velocity macros are available in `VM_liferay.vm` template Liferay DXP also exposes its taglibs as FreeMarker macros. See the corresponding taglib tutorial for more information on using the taglib in your FreeMarker templates. This tutorial shows how to use Liferay DXP's macros in your FreeMarker and Velocity theme templates.

Note that **Velocity templates are supported, but deprecated as of Liferay Portal CE 7.0 and Liferay DXP 7.0**. We recommend that you convert your Velocity theme templates to FreeMarker at your earliest convenience to avoid future issues. This tutorial covers both FreeMarker and Velocity template macros to help with your conversion process.

The syntax for a macro is straightforward. A macro directive is defined containing the template fragment. For example, below is the macro directive for Liferay DXP's Control Menu:

FreeMarker:

```
<#macro control_menu>
	<#if themeDisplay.isImpersonated() || (is_setup_complete && is_signed_in)>
		<@liferay_product_navigation["control-menu"] />
	</#if>
</#macro>
```

Velocity:

```
#macro (control_menu)
	#if ($themeDisplay.isImpersonated() || ($is_setup_complete && $is_signed_in))
	    $theme.runtime(
	        "com.liferay.admin.kernel.util.PortalProductNavigationControlMenuApplicationType$ProductNavigationControlMenu",
	        $portletProviderAction.VIEW
	    )
	#end
#end
```

Note: Liferay DXP's default FreeMarker macro calls are namespaced with `liferay` (for example, `<@liferay.macro_variable_name />`). If you create custom macros, they can be called with the explicit variable name.

To include the template fragment in your theme templates, call the macro using the variable name:

FreeMarker:

```
<@liferay.control_menu />
```

Velocity:

```
#control_menu()
```

The macro is replaced with the template fragment when the page is rendered. That's all there is to a basic macro.

Macros can also be passed arguments. For example Liferay DXP's `language` macro has a language key parameter:

FreeMarker:

```
<#macro language
       key
>
${languageUtil.get(locale, key)}
</#macro>
```

Velocity:

```
#macro (language $lang_key)
$languageUtil.get($locale, $lang_key)
#end
```

You can pass an argument in the macro call like this:

FreeMarker:

```
<@liferay.language key="powered-by" />
```

Velocity:

```
#language ("powered-by")
```

You can read more about FreeMarker macros and Velocity macros at freemarker.org and velocity.apache.org.

Liferay DXP provides several macros that you can use in your theme templates. These are covered next.

Liferay DXP Macros

There are several default macros defined in the `FTL_Liferay.ftl` template that you can use in your FreeMarker theme templates. Likewise, VM_liferay.vm defines the default macros for Velocity. The table below lists the available macros and parameters:

Macro	Parameters	Description
breadcrumbs	default preferences	Adds the Breadcrumbs portlet with optional preferences
control_menu	N/A	Adds the Control Menu portlet
css	filename	Adds an external stylesheet with the specified file name location
date	format	Prints the date in the current locale with the given format
js	filename	Adds an external JavaScript file with the specified file name source
language	key	Prints the specified language key in the current locale

Macro	Parameters	Description
language_format	argumentskey	Formats the given language key with the specified arguments. For example, passing `go-to-x` as the key and `Mars` as the arguments prints *Go to Mars*.
languages	default preferences	Adds the Languages portlet with optional preferences
navigation_menu	default preferencesinstance ID	Adds the Navigation Menu portlet with optional preferences and instance ID. `${freeMarkerPortletPreferences}` or `$velocityPortletPreferences.toString()` is a common value for default preferences.
search	default preferences	Adds the Search portlet with optional preferences
user_personal_bar	N/A	Adds the User Personal Bar portlet

Now you know how to use Liferay DXP's macros in your theme templates!

Related Topics

Liferay Theme Generator

Themelets

Theme Reference Guide

106.8 Theme Builder

Liferay's Theme Builder gives developers who aren't using Liferay's Theme Generator (e.g., Gradle or Maven) a way to compile and build a theme WAR file. To use the Theme Builder, you must apply it to your project. If you're unsure how to structure themes for Liferay DXP, see the Introduction to Themes tutorial.

Follow the instructions below to apply the Theme Builder plugin and build your theme WAR.

Step 1: Apply the Theme Builder Plugin to Your Theme Project

Liferay provides two Theme Builder plugins depending on your build tool:

- com.liferay.portal.tools.theme.builder (Ant, Maven, etc.)
- com.liferay.gradle.plugins.theme.builder (Gradle)

If you want to apply the Theme Builder plugin to an Ant project, examine the `build.xml` file as an example below:

```
<?xml version="1.0"?>
<!DOCTYPE project>

<project>
    <path id="theme.builder.classpath">
        <fileset dir="[PATH_TO_THEME_BUILDER_JAR]" includes="com.liferay.portal.tools.theme.builder-*.jar" />
    </path>

    <taskdef classpathref="theme.builder.classpath" resource="com/liferay/portal/tools/theme/builder/ant/taskdefs.properties" />

    <target name="build-theme">
        <build-theme
            diffsDir="diffs"
            outputDir="../dist"
            parentDir="[PATH_TO_STYLED_THEME]/classes/META-INF/resources/_styled"
            parentName="_styled"
            unstyledDir="[PATH_TO_UNSTYLED_THEME]/classes/META-INF/resources/_unstyled"
        />
    </target>
</project>
```

You should first supply the path to the Theme Builder JAR. The above code configures the literal path to the JAR on your local machine. As an alternative, you could configure Ivy to do this for you behind the scenes. Then create an Ant target (e.g., `build-theme`) that configures the required parameters to build your theme.

For assistance applying the Theme Builder plugin for a Gradle or Maven project, see the Theme Builder Gradle Plugin or Building Themes in a Maven Project articles, respectively.

Step 2: Build Your Theme

Execute the appropriate command based on your build tool:

- *Ant:* `ant build-theme`
- *Gradle:* `gradlew buildTheme`
- *Maven:* `mvn verify`

The WAR is generated in the following folder, depending on the build tool you used:

- *Ant:* `/dist`
- *Gradle:* `/build`
- *Maven:* `/target`

That's it! You've successfully configured and leveraged the Theme Builder in your project. You can also use the Theme Builder to migrate a Plugins SDK theme to Liferay Workspace. See the Migrating a Theme from the Plugins SDK to Workspace tutorial for details.

106.9 Creating a Theme Thumbnail

Theme thumbnails help users quickly identify your theme. It's especially important to provide thumbnails when your theme offers color schemes.

Here's how to create a proper thumbnail image for your theme:

1. Create a thumbnail image. Make sure it's 150 pixels wide by 120 pixels high. You may want to take a snapshot of your theme and re-size it to these dimensions. **Your thumbnail must be these exact dimensions** or the image won't display properly.

2. Save the image as a .png file named thumbnail.png and place it in the theme's images folder (create this folder if it doesn't already exist). On redeployment, the thumbnail.png file automatically becomes the theme's thumbnail.

Note: The Theme Builder Gradle plugin doesn't recognize a thumbnail.png file. If you're using this plugin to build your theme instead, you must create a screenshot.png file in your theme's images folder that is 1080 pixels high by 864 pixels wide. The thumbnail is automatically generated from the screenshot for you when the theme is built.

Now, when you apply the theme, its thumbnail displays along with the other themes that are available to your site.

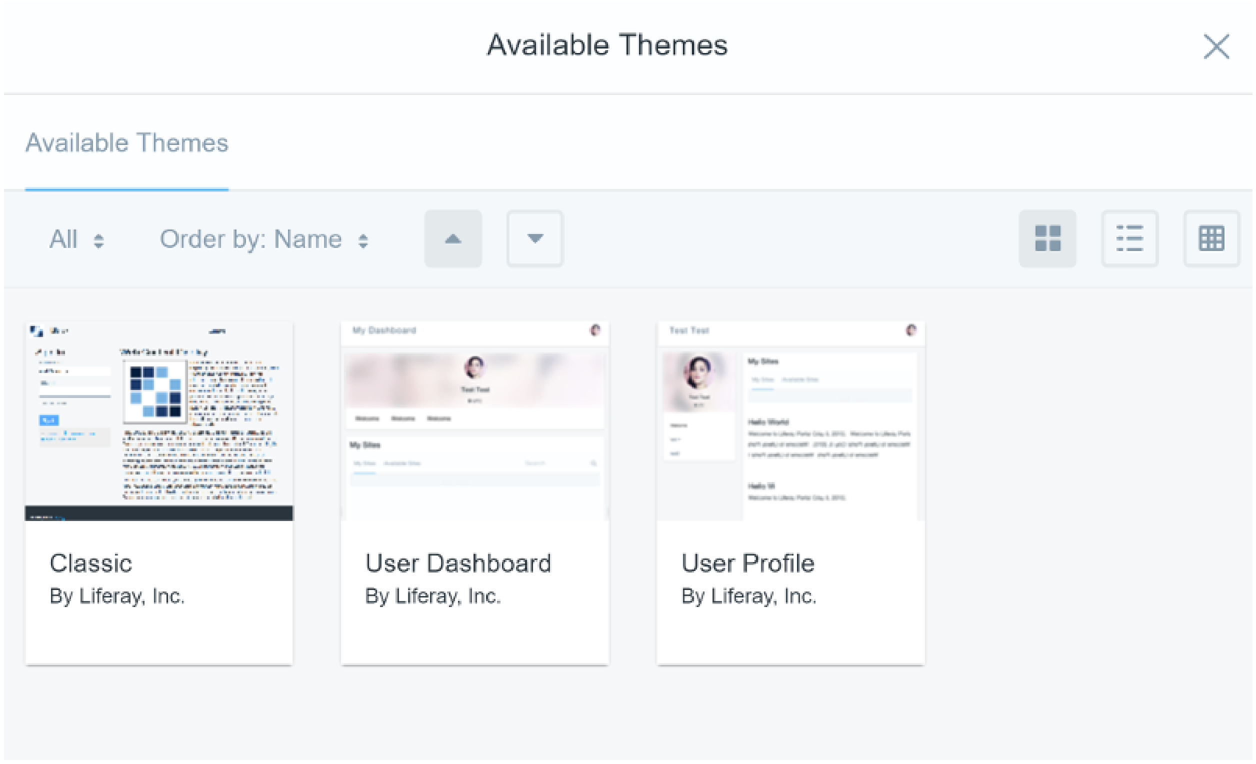


Figure 106.6: Your theme thumbnail is displayed with the rest of the available themes.

Congrats! Now you know how to create a thumbnail image for your theme!

Related Topics

Liferay Theme Generator Specifying Color Schemes in Your Theme

106.10 Specifying Color Schemes in your Theme

You can provide various "flavors" of your theme by creating color schemes. Color schemes let you keep the styles and overall design for your theme, while giving a new look for your users to enjoy. You specify color schemes with a CSS class name, which also lets you choose different background images, different border colors, and more.

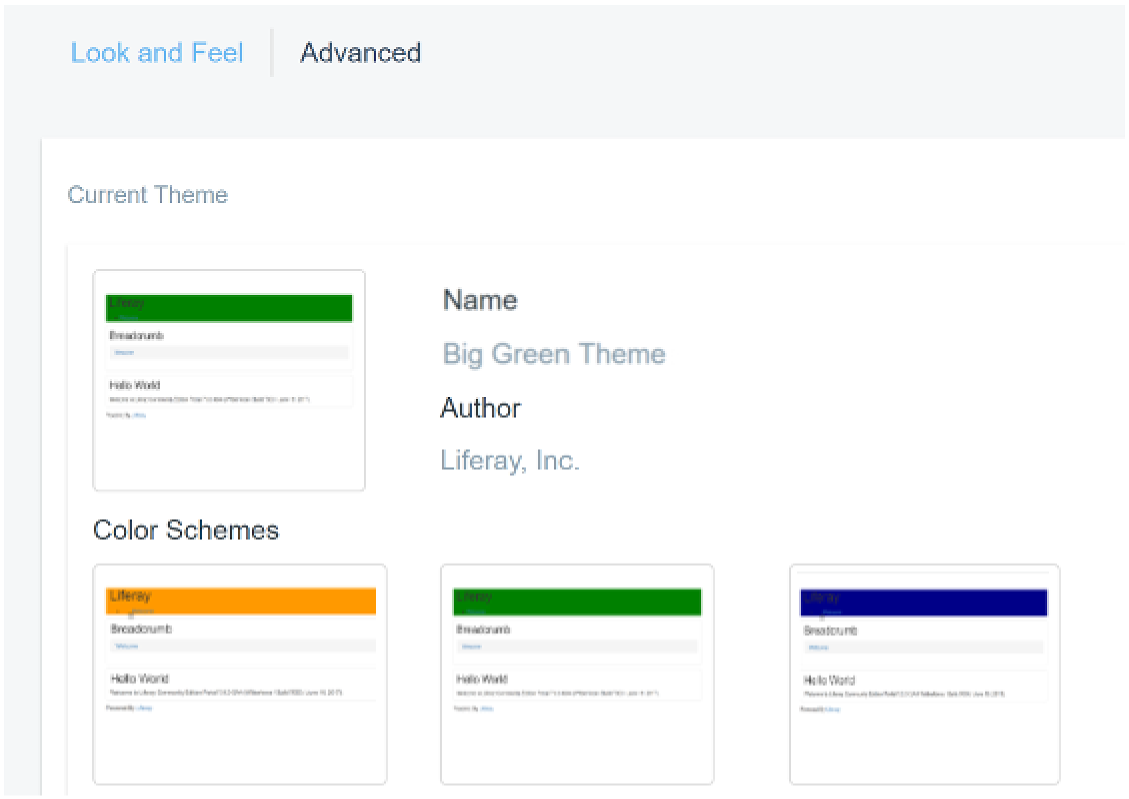


Figure 106.7: You can offer eye-pleasing color schemes for your themes.

Follow these steps to create color schemes for your theme:

1. Create a folder to hold color schemes (`color_schemes` for example) in the theme's css folder.

2. Create an `.scss` file in the color schemes folder for each color scheme your theme supports. If you don't specify a `.scss` file for a color scheme, the theme's default color scheme is used.

3. Prefix all CSS styles with the name of your color scheme. The color scheme CSS class is placed on the page's `<body>` element, so you can use it to identify your styling. For example, you'd prefix all the styles with the word *day* in a color scheme CSS file named `_day.scss`:

```
body.day { background-color: #DDF; }
.day a { color: #66A; }
```

```
**Note:** The default color scheme does not require a prefix, as it uses
the theme's `_custom.scss` for styling.
```

---

4. Import the color scheme `.scss` files into the `_custom.scss` file. The example below imports `_day.scss` and `_night.scss` files:

```
@import "color_schemes/day";
@import "color_schemes/night";
```

5. Open the theme's `liferay-look-and-feel.xml` file and add the default color scheme for the theme. Pass the default color scheme's CSS class name (the name of the CSS file) in the `<css-class>` element. If the default color scheme styling is in the theme's `_custom.scss` file, use `default` for the `<css-class>`:

   ```
   <theme id="my-theme-id" name="My Theme Name">
      <color-scheme id="01" name="My Default Color Scheme Name">
          <default-cs>true</default-cs>
          <css-class>default</css-class>

          <color-scheme-images-path>
              ${images-path}/my_color_schemes_folder_name/${css-class}
          </color-scheme-images-path>
      </color-scheme
      ...
   </theme>
   ```

   A default color scheme lets users return to the theme's default look and feel. Note that the color scheme's `name` is arbitrary. Only the color scheme's `css-class` element must match the name of the color scheme's CSS class.

   Note that color schemes are sorted alphabetically by `name` rather than `id`. For example, a color scheme named `Day` and `id` `02` would be selected by default over a color scheme named `Clouds` with `id` `01`. The `<default-cs>` element overrides the alphabetical sorting and sets the color scheme that is selected by default, when the theme is chosen. Adding this element to the default color scheme ensures that it is selected when the theme is chosen.

   The `<color-scheme-images-path>` element specifies theme thumbnail image location. Place this element in the first color scheme to affect them all. For example you could use the folders `/images/color_schemes/default`, `/images/color_schemes/day`, and `/images/color_schemes/night`.

6. Add the remaining color schemes to the `liferay-look-and-feel.xml` using the pattern below:

   ```
   <color-scheme id="02" name="my-color-scheme-name">
      <css-class>my-color-scheme-css-class-name</css-class>
   </color-scheme>
   ```

   The example below defines a *Day* color scheme and a *Night* color scheme for a theme named *Big Green*. Here's the code as specified in the `liferay-look-and-feel.xml` file:

   ```
   <theme id="big-green" name="Big Green">
       <color-scheme id="01" name="Default">
           <default-cs>true</default-cs>
           <css-class>default</css-class>
           <color-scheme-images-path>
               ${images-path}/color_schemes/${css-class}
           </color-scheme-images-path>
       </color-scheme>
       <color-scheme id="02" name="Day">
           <css-class>day</css-class>
       </color-scheme>
       <color-scheme id="03" name="Night">
           <css-class>night</css-class>
       </color-scheme>
   </theme>
   ```

7. Place a `thumbnail.png` and `screenshot.png` file in each of the color scheme's folders. Make sure thumbnail images follow the specifications defined in the Creating a Theme Thumbnail tutorial.

There you have it. Now you can go color scheme crazy with your themes!

Related Topics

Layout Templates with the Theme Generator

Creating a Theme Thumbnail

106.11 Making Themes Configurable with Settings

Theme settings let site administrators control the look and feel of certain aspects of a theme. For example, you can create a theme setting to control the visibility of theme elements, such as only showing a site banner when the user is logged in. You can also create a theme setting to configure an element, such as a title or a background image. The Settings API is built flexibly to meet your needs. The sky's the limit.

This tutorial covers how to create theme settings for a theme.

Making configurable theme settings involves a multi-step process:

- Add the settings to `liferay-look-and-feel.xml`
- Assign the settings to variables in `init_custom.ftl`
- Use the settings variables in your theme templates

Follow these steps to create theme settings:

1. Open `liferay-look-and-feel.xml` from the theme's `WEB-INF` folder. Settings placed here appear in the *Look and Feel* menu of Liferay DXP's *Site Administration*. If your project doesn't have this file, create it in the `WEB-INF` folder and add the following XML content (make sure to replace the theme `id` and `name` with your theme's):

```
<?xml version="1.0"?>
<!DOCTYPE look-and-feel PUBLIC "-//Liferay//DTD Look and Feel 7.0.0//EN"
"http://www.liferay.com/dtd/liferay-look-and-feel_7_0_0.dtd">

<look-and-feel>
    <compatibility>
        <version>7.0.0+</version>
    </compatibility>
    <theme id="your-theme-name" name="Your Theme Name">
        <template-extension>ftl</template-extension>
    <portlet-decorator id="barebone" name="Barebone">
            <portlet-decorator-css-class>portlet-barebone</portlet-decorator-css-class>
        </portlet-decorator>
        <portlet-decorator id="borderless" name="Borderless">
            <portlet-decorator-css-class>portlet-borderless</portlet-decorator-css-class>
        </portlet-decorator>
        <portlet-decorator id="decorate" name="Decorate">
            <default-portlet-decorator>true</default-portlet-decorator>
            <portlet-decorator-css-class>portlet-decorate</portlet-decorator-css-class>
        </portlet-decorator>
    </theme>
</look-and-feel>
```

2. Add `<settings></settings>` tags before the opening `<portlet-decorator>` tag.

3. Add a `<setting/>` element between the `<settings></settings>` tags for each setting the theme requires. Below is an example pattern along with the available attributes:

   ```
   <setting configurable="true" key="my-setting-language-key"
   options="true,false" type="select" value="false" />
   ```

   The following attributes are available for theme settings:

   configurable: whether the setting is configurable or static. **key:** the language key that identifies the theme setting. **options:** sets the options for the select menu if the `type` is `select`. **type:** the type of UI to render to control the theme setting. Possible values are `checkbox`, `select`, `text`, or `textarea`. **value:** sets the default value for the theme setting.

   You can find more information about setting attributes and all other configurations for the `liferay-look-and-feel.xml` in its DTD docs.

4. Open `init_custom.ftl` and assign the settings to variables using the patterns below.

   Use the following pattern for settings that return a `Boolean` (for example a select box with the options `true` and `false` or a toggle-switch checkbox with values `yes` and `no`):

   ```
   <#assign my_variable_name =
   getterUtil.getBoolean(theme_settings["theme-setting-key"])>
   ```

   Use the following pattern for settings that return a `String` (for example, a text input or textarea input):

   ```
   <#assign footer_text =
   getterUtil.getString(theme_settings["theme-setting-key"])/>
   ```

5. Use the theme setting variable in the theme template. For example, you can use the variable to check a condition, print a value, or even display a theme template. Examples of each case are shown below.

   This configuration is used in `portal_normal.ftl` to show the navigation breadcrumbs element if the `show_breadcrumbs` theme setting is `true`:

   `liferay-look-and-feel.xml`:

   ```
   <setting configurable="true" key="show-breadcrumbs" type="checkbox"
   value="false" />
   ```

   `init_custom.ftl`:

   ```
   <#assign show_breadcrumbs =
   getterUtil.getBoolean(theme_settings["show-breadcrumbs"])>
   ```

   `portal_normal.ftl`:

   ```
   <#if show_breadcrumbs>
     <nav id="breadcrumbs">
       <@liferay.breadcrumbs />
     </nav>
   </#if>
   ```

```
This example configuration prints a text input's value in a `<p>` element,
or defaults to *Hello Text* if no value is given:

`liferay-look-and-feel.xml`:
```

```
    <setting configurable="true" key="custom-text" type="text"
    value="Hello Text"/>

`init_custom.ftl`:

    <#assign custom_text =
    getterUtil.getString(theme_settings["custom-text"])/>

`portal_normal.ftl`:

    <p>${custom_text}</p>

This example renders the brief header template or detailed header
template based on the theme setting:

`liferay-look-and-feel.xml`:

    <setting configurable="true" key="header-type" type="select"
    options="brief,detailed" value="brief"/>

`init_custom.ftl`:

    <#assign header_type =
    getterUtil.getString(theme_settings["header-type"])/>

`header_brief.ftl`: brief header template

`header_detailed.ftl`: detailed header template

`portal_normal.ftl`:

    <#if header_type == "brief">
      <#include "${full_templates_path}/header_brief.ftl" />
    <#elseif header_type == "detailed">
      <#include "${full_templates_path}/header_detailed.ftl" />
    </#if>
```

6. Make sure the theme is installed. Open the *Control Menu* → *Site Administration* → *Navigation* → *Public Pages* and select the *Configure* option. Configure the theme settings from the *Look and Feel* section to see your changes. You can set the theme settings for an individual page by selecting the *Configure Page* option from the page's Actions menu and selecting the *Define a Specific look and feel for this page* option under the *Look and Feel* section.

Now you know how to make configurable theme settings for your themes!

Related Topics

Macros
Themelets
Theme Contributors

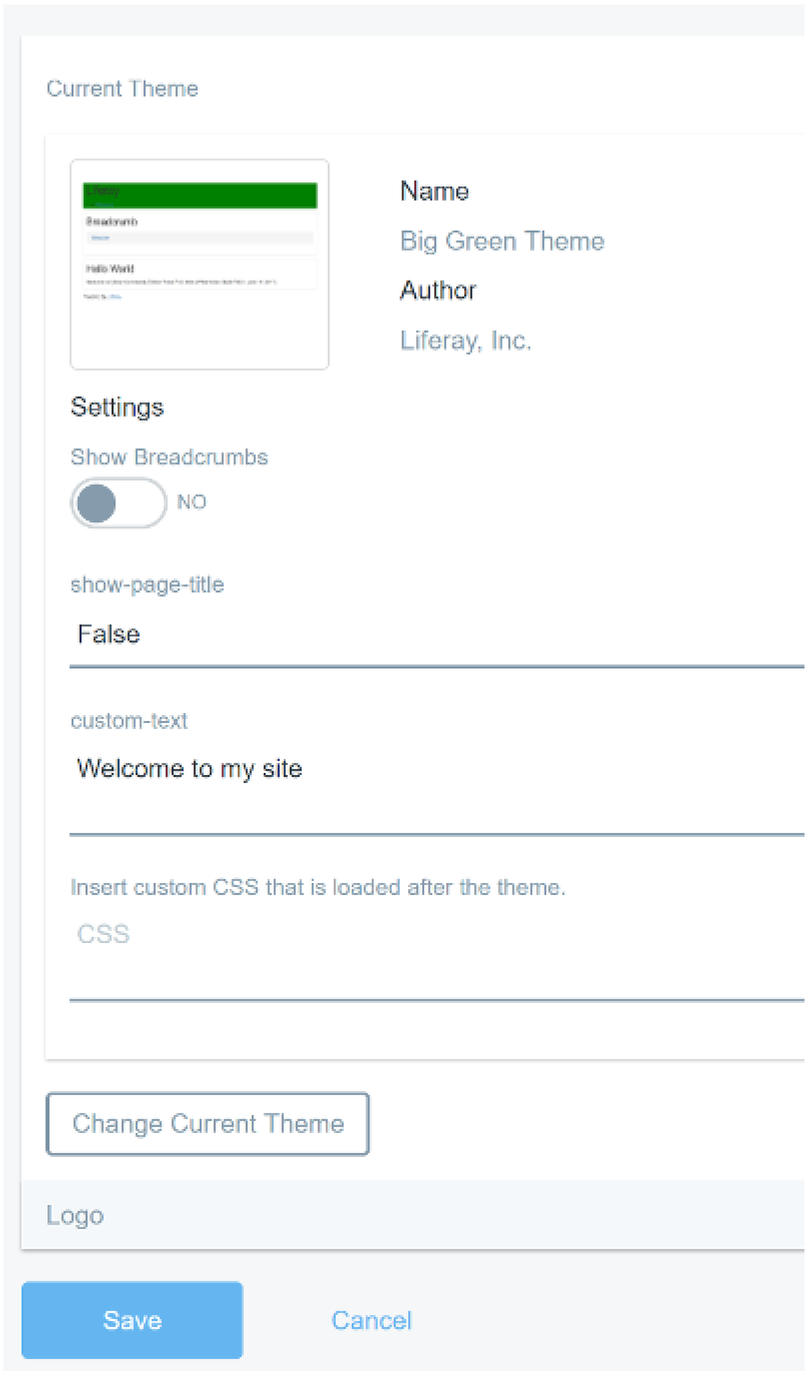


Figure 106.8: Here are examples of configurable settings for the site Admin.

CHAPTER 107

LAYOUT TEMPLATES

Are you craving more than Liferay DXP's default page layouts? Do you have a special use case that a default layout template doesn't meet? Well, look no further. Create your own Layout Template! Layout templates allow you to set the rows and columns of a page and determine where content can be placed.

In this section of tutorials, you'll learn how to develop layout templates for Liferay DXP.

107.1 Layout Templates with the Liferay Theme Generator

Layout Templates specify how content is arranged on your site pages in Liferay DXP. For example, take a look at the *1-2-1 Columns Layout CE* layout shown below:

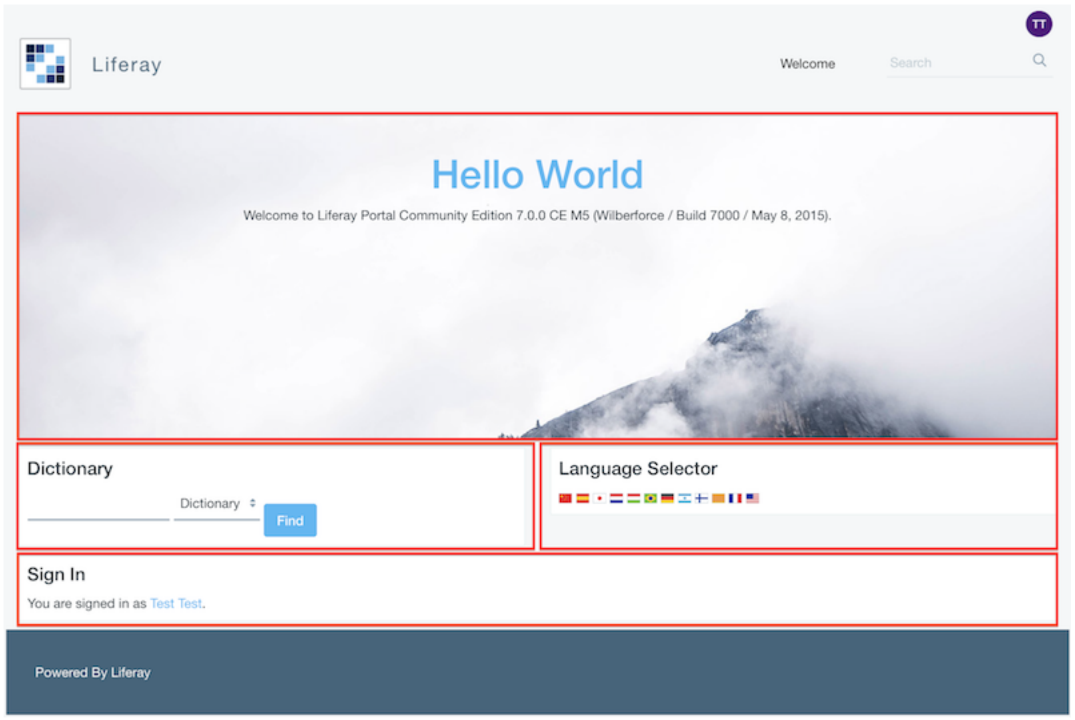

TT
Liferay
Welcome
Search
Hello World
Welcome to Liferay Portal Community Edition 7.0.0 CE M5 (Wilberforce / Build 7000 / May 8, 2015).
Dictionary
Dictionary
Find
Language Selector
Sign In
You are signed in as Test Test.
Powered By Liferay

Figure 107.1: The *1-2-1 Columns Layout CE* page layout creates a nice flow for your content.

7.0 and DXP provide several layout templates out-of-the-box for you to choose from. You can change the layout for your page, and view the installed layout templates, by opening the `Edit` menu for your page, under the `Navigation` heading of the `Product Menu` and scrolling down to the *Layouts* heading.

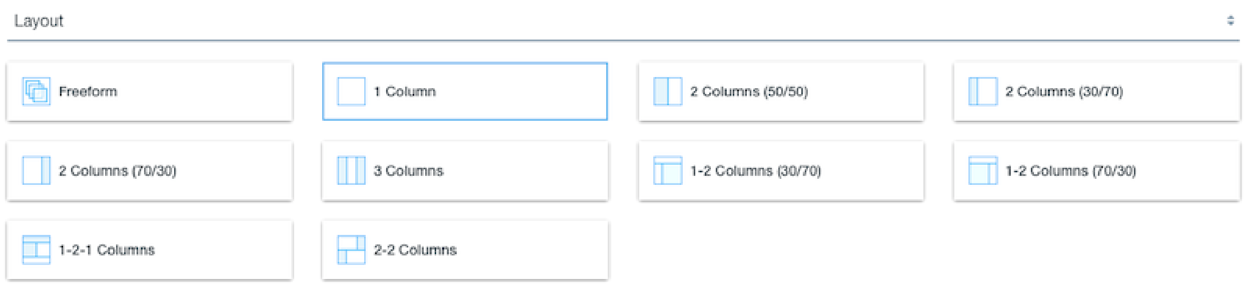


Figure 107.2: Liferay provides several layout templates out-of-the-box for you to use.

If you'd like to create your own custom layout templates, you've come to the right place.
This tutorial demonstrates how to:

- Create a Layout Template with the Layouts Sub-generator
- Create a Thumbnail for a Layout Template

In order to create a layout template with the Layouts Sub-generator, you will need the Node.js build tools installed. The Liferay Theme Generator tutorial explains how to install these tools and how to create a theme.
Once you have the Liferay Theme Generator installed you can go ahead and get started.

Creating a Layout Template with the Layouts Sub-generator

Follow these steps to create a layout template:

1. Open the Command Line and navigate to the directory you want to create your layout template in.

2. Run `yo liferay-theme:layout` to start the layouts sub-generator.

   Note: If you run the layout sub-generator from the root directory of a theme created with the themes generator, it will add the layout template as a part of the theme in the `src/layouttpl` directory.

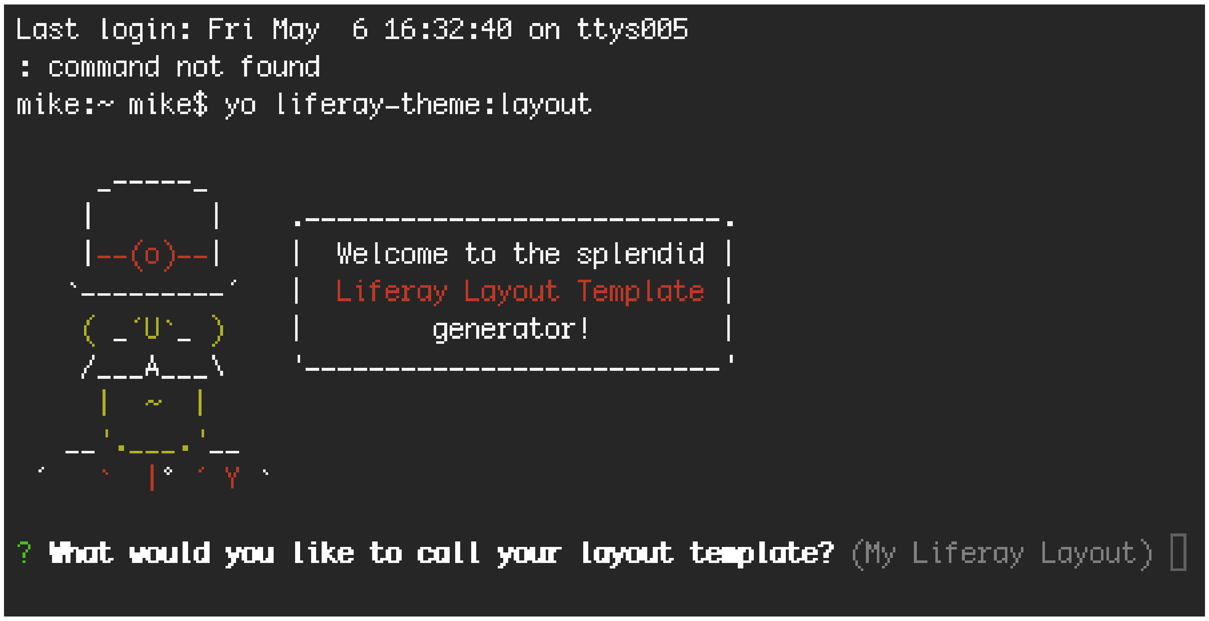
3. Enter a name and ID for your layout template, or press Enter to accept the default values.

4. Choose your Liferay version and press Enter to continue.

   At this point the layout template design process begins. As the generator states, Layout templates implement Bootstrap's grid system. Every row consists of 12 sections, so columns range in size from 1 to 12. The sub-generator is user-friendly, allowing you to add and remove rows and columns as you design.

5. Enter the number of columns you would like for `row 1`.

   Once you've entered a value, the generator asks how wide you want your row and column to be, and presents you with the available width(s).

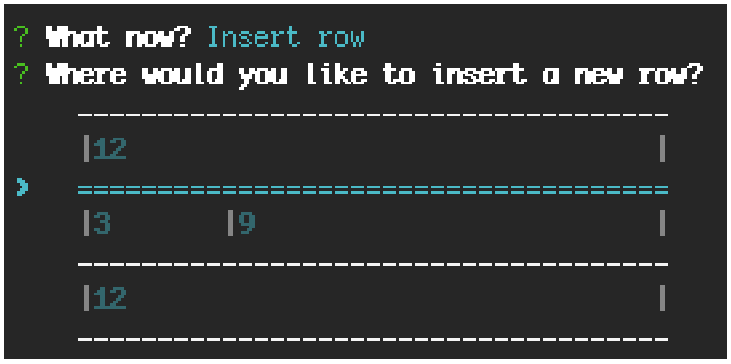


```
Last login: Fri May  6 16:32:40 on ttys005
: command not found
mike:~ mike$ yo liferay-theme:layout

     _-----_
    |       |    .--------------------------.
    |--(o)--|    |  Welcome to the splendid |
   `---------´   |  Liferay Layout Template |
    ( _´U`_ )    |        generator!        |
    /___A___\    '--------------------------'
     |  ~  |
   __'.___.'__
 ´   `  |° ´ Y `

? What would you like to call your layout template? (My Liferay Layout)
```

Figure 107.3: The Layout Template sub-generator automates the layout creation process.

6. Choose from the available option(s) with your arrow keys and press Enter to make your selection. If you have remaining space, the generator will repeat this step for the remaining columns. Once you're done configuring your row, you are presented with a few options:

   - Add a row: Adds a row below the last row.
   - Insert row: Displays a vi to insert your row. Use your arrow keys to choose where to insert your row, highlighted in blue, then press Enter to insert the row.

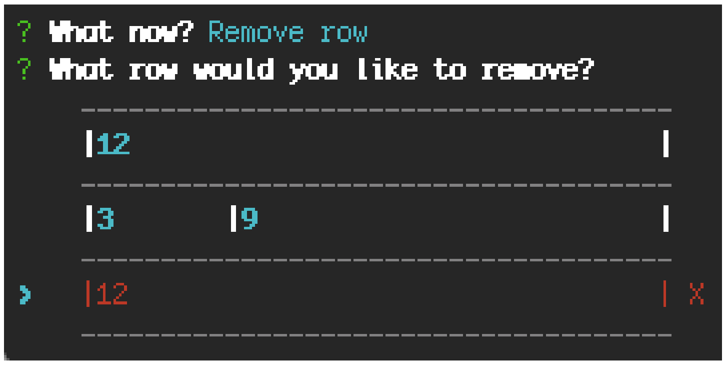


```
? What now? Insert row
? Where would you like to insert a new row?
    -------------------------------------
    |12                                 |
  > =====================================
    |3        |9                        |
    -------------------------------------
    |12                                 |
    -------------------------------------
```

Figure 107.4: Rows can be inserted using the layout vi.

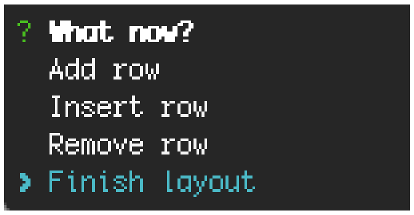
   - Remove row: Displays a vi to remove your row. Use your arrow keys to select the row you want to remove, highlighted in red, then press Enter to remove the row.

   Once you are done designing your layout you can move onto the next step.



```
? What now? Remove row
? What row would you like to remove?
    ----------------------------------------
    |12                                    |
    ----------------------------------------
    |3        |9                           |
    ----------------------------------------
>   |12                                    | X
    ----------------------------------------
```

Figure 107.5: Rows are removed using the layout vi.



```
? What now?
  Add row
  Insert row
  Remove row
> Finish layout
```

Figure 107.6: Select the *Finish layout* option to complete your design.

7. Select *Finish layout* to complete your layout's design.

   Your layout template files are generated for you in the current directory.

8. Enter the path to your app server directory, or press Enter to accept the default.

9. Enter the URL to your server or press Enter to accept the default `http://localhost:8080` development site.

   Your layout template is generated, but you still need to include it in your `liferay-look-and-feel.xml` file.

```
**Note:** Currently the Liferay Theme Generator does not add the layout
template configuration to your `liferay-look-and-feel.xml`. This feature
will be added in a future release. For now you must add this manually next.
```

---

10. Add your custom layout template to your `liferay-look-and-feel.xml` using the `<layout-templates>` tag. Below is an example configuration for the Porygon theme's' layout templates:

    ... /layouttpl/custom/porygon_70_30_width_limited.tpl /layouttpl/custom/porygon_70_30_width_limited.png /layouttpl/custom/porygon_50_50_width_limited.tpl /layouttpl/custom/porygon_50_50_width_limited.png

```
The `<layout-template>` tag's `id` attribute must match the ID you gave in
step 3 (the TPL file name).
```

When your layout template was generated, a default thumbnail was created. You can learn how to create a custom thumbnail in the next section.

Creating a Custom Thumbnail for Your Layout Template

To create your own thumbnail follow the steps below:

1. Navigate to the `docroot` directory of the layout template you just created.

   Note: if you created the layout template in your existing themes generator theme, the thumbnail is located in your theme's `src/layouttpl/custom` directory.

2. Replace the `layout-name.png` file with your own custom thumbnail PNG.

3. navigate back to the layout's root directory and run `gulp deploy` to re-build and deploy the template to your app server.

   Note: If your layout template was added as part of your themes generator theme, the layout template will deploy when the theme is deployed.

```
**Note:** Currently the Liferay Theme Generator does not add the thumbnail
configuration to your `liferay-look-and-feel.xml`. This feature will be
added in a future release. For now you must add this manually next.
```

---

4. Specify the thumbnail's location in your `liferay-look-and-feel.xml` using the `<thumbnail-path>` tag. Below is an example configuration for the Porygon theme:

```
<layout-template id="porygon_50_50_width_limited"
name="Porygon 2 Columns (50/50) width limited">
    <template-path>
        /layoutttpl/custom/porygon_50_50_width_limited.tpl
    </template-path>
    <thumbnail-path>
        /layoutttpl/custom/porygon_50_50_width_limited.png
    </thumbnail-path>
</layout-template>
```

Your layout template is complete! As you can see, the layouts sub-generator makes creating a layout template a piece of cake.

Edit a page on your site and select your new layout template to use it.

Related Topics

Importing Resources with Your Themes

Liferay Theme Generator

107.2 Creating Layout Templates Manually

You can use the Liferay Theme Generator to generate Layout Templates automatically. This is covered in the Layout Templates with the Liferay Theme Generator tutorial. You may, however, want to create or modify your layout templates manually.

In this tutorial you'll learn how to create or modify a Layout Template manually.

You can see the HTML markup for a basic layout template next.

Basic Layout Template

Below is an example of a basic Layout Template `.tpl` file:

```
<div class="my-liferay-layout" id="main-content" role="main">
        <div class="portlet-layout row">
                <div class="col-md-4 portlet-column portlet-column-first"
                id="column-1">
                        $processor.processColumn("column-1",
                        "portlet-column-content portlet-column-content-first")
                </div>
                <div class="col-md-8 portlet-column portlet-column-last" id="column-2">
                        $processor.processColumn("column-2",
                        "portlet-column-content portlet-column-content-last")
                </div>
        </div>
        <div class="portlet-layout row">
                <div class="col-md-12 portlet-column portlet-column-only" id="column-3">
                        $processor.processColumn("column-3",
                        "portlet-column-content portlet-column-content-only")
                </div>
        </div>
        <div class="portlet-layout row">
                <div class="col-md-4 portlet-column portlet-column-first" id="column-4">
                        $processor.processColumn("column-4",
                        "portlet-column-content portlet-column-content-first")
                </div>
                <div class="col-md-4 portlet-column" id="column-5">
                        $processor.processColumn("column-5", "portlet-column-content")
                </div>
                <div class="col-md-4 portlet-column portlet-column-last" id="column-6">
                        $processor.processColumn("column-6",
                        "portlet-column-content portlet-column-content-last")
                </div>
        </div>
</div>
```

The column elements and classes are described in more detail next.

Column Elements and Classes

To understand how the layout template works, you must look closely at how the HTML is structured. This section uses the first column of the example above to demonstrate the key elements and classes of a layout template:

```
<div class="col-md-4 portlet-column portlet-column-first" id="column-1">
        $processor.processColumn("column-1", "portlet-column-content
        portlet-column-content-first")
</div>
```

You can learn more about the column container next.

Column Container

Below is a description of each of the column container classes:

`column-1`: A unique identifier for the column that matches the ID passed to `$processor.processColumn`.

`col-md-4`: This class comes from Bootstrap's grid system and determines two things: the percentage based width of the element, and the media query breakpoint for when this element expands to 100% width. 12 is the maximum amount, so `col-md-4` indicates `4/12` width, or `33.33%`.

portlet-column portlet-column-first: All column containers must use the portlet-column class. For rows with more than one column, the first column must have portlet-column-first and the last must have portlet-column-last. For rows with only one column, use the portlet-column-only class.

Next you can learn more about the $processor.processColumn.

Processor ProcessColumn

$processor.processColumn takes these arguments:

column-1: A unique identifier. This should match the ID of the parent div.

portlet-column-content portlet-column-content-first: Additional classes added to the content element. These classes must match the parent div's classes with -content appended.

Next you can learn how to modify template breakpoints.

Modifying Template Breakpoints

When looking at the example template, you'll notice this Bootstrap grid class is used on every column:

```
col-md-{size}
```

The different sizes available are xs, sm, md, and lg. The medium size is used by default, but the others can be used in layout templates as well.

For example, setting the column classes to col-lg-{size} means the columns would expand to 100% width at a larger screen width than col-md-{size}.

These classes can also be mixed to achieve more advanced layouts, as shown below:

```
<div class="portlet-layout row">
        <div class="col-md-4 col-sm-6 portlet-column portlet-column-first"
        id="column-1">
                $processor.processColumn("column-1",
                "portlet-column-content portlet-column-content-first")
        </div>
        <div class="col-md-8 col-sm-6 portlet-column portlet-column-last"
        id="column-2">
                $processor.processColumn("column-2",
                "portlet-column-content portlet-column-content-last")
        </div>
</div>
```

In the example row above, on medium sized view, ports column-1 are 33.33% width and column-2 are 66.66% width, but on small sized view ports both column-1 and column-2 are 50% width.

Place the completed layout in your theme's src/layouttpl folder if you created your theme with Theme's Generator, or place it in your theme's docroot/layouttpl/custom folder if using the Plugins SDK.

Next you can learn how to include layout templates in a theme.

Including Layout Templates with a Theme

You can deploy a layout template with a theme by specifying it in the theme's liferay-look-and-feel.xml file.

Add your custom layout template to your liferay-look-and-feel.xml using the <layout-templates> tag. Below is an example configuration using the basic layout template example shown above:

```
<theme id="my-theme-name" name="My Theme Name">
    ...
      <layout-templates>
        <custom>
          <layout-template id="my_liferay_layout_template"
          name="My Liferay Layout Template">
            <template-path>
                /layoutttpl/custom/my_liferay_layout_template.tpl
            </template-path>
            <thumbnail-path>
                /layoutttpl/custom/my_liferay_layout_template.png
            </thumbnail-path>
          </layout-template>
        </custom>
      </layout-templates>
    ...
</theme>
```

The `<layout-template>` tag's `id` attribute must match the layout template's filename. The template and thumbnail paths shown above are for files in the `layouttpl/custom` folder. If you created your layout template with the Liferay Theme Generator, your file paths may differ.

There you have it. Now you know how to create and modify layout templates manually and how to include them with a theme!

Related Topics

Layout Templates with the Liferay Theme Generator

Liferay Theme Generator

Chapter 108

Portlets and Themes

Liferay DXP gives you complete control over the look and feel of your portlets. You can provide custom styles for portlets, create style configuration options via portlet decorators, and embed portlets in themes and layout templates.

All these topics are covered in this section of tutorials.

Chapter 109

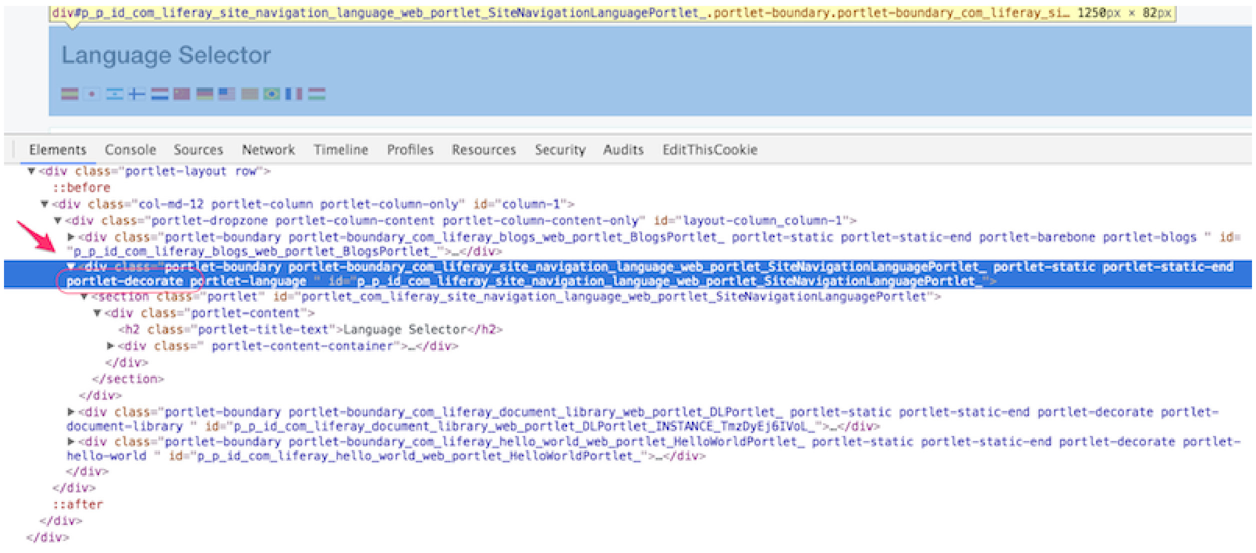
Portlet Decorators

In previous versions of Liferay DXP, administrators could display or hide the application borders through the *Show Borders* option of the look and feel configuration menu. In 7.0 this option has been replaced with Portlet Decorators, a more powerful mechanism to customize the style of the application wrapper.

If you inspect the markup of your Liferay application when it's on a page you'll observe that it is wrapped by two layers. Among other things, these layers provide some common basic features like the drag and drop and the application border style. In order to protect these features, you can't modify the markup of these layers directly with a theme.

Portlet Decorators provide a mechanism to add a CSS class to one of these wrapping layers via a user's setting. By defining styles for this class in your theme, you can change the look and feel of the application instances where the Portlet Decorator is applied, including its wrapper.

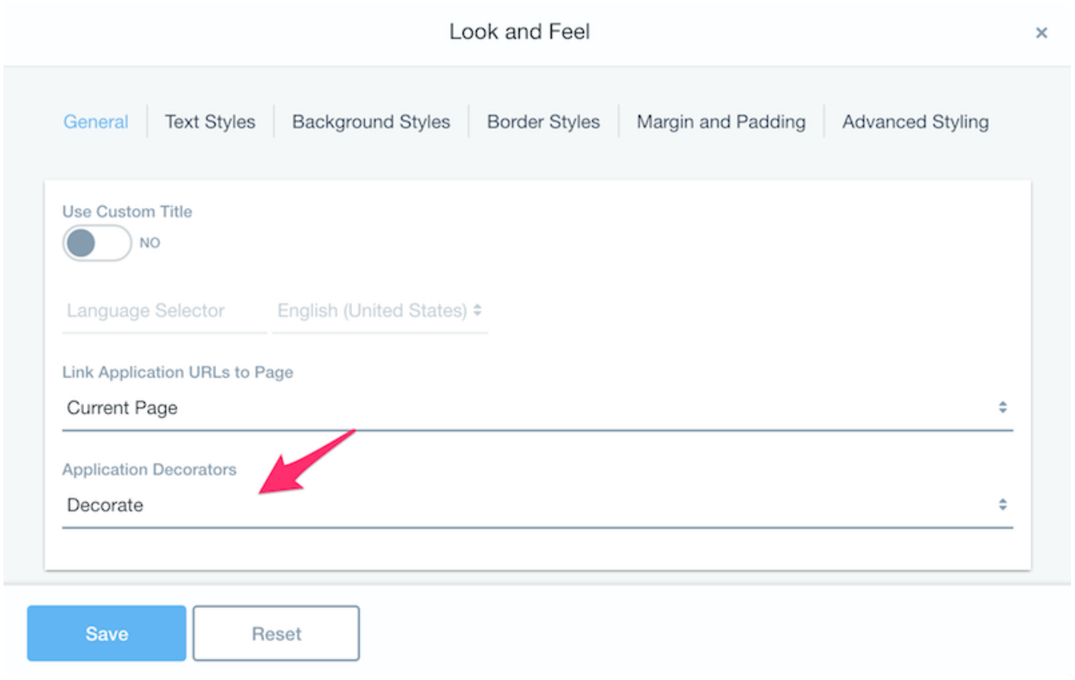
The figure below shows the markup of the layers wrapping a Liferay application when the *Decorate* Portlet Decorator is applied:


div#p_p_id_com_liferay_site_navigation_language_web_portlet_SiteNavigationLanguagePortlet_.portlet-boundary.portlet-boundary_com_liferay_si… 1250px × 82px
Language Selector
Elements
Console
Sources
Network
Timeline
Profiles
Resources
Security
Audits
EditThisCookie

```
▼<div class="portlet-layout row">
   ::before
  ▼<div class="col-md-12 portlet-column portlet-column-only" id="column-1">
    ▼<div class="portlet-dropzone portlet-column-content portlet-column-content-only" id="layout-column_column-1">
      ▶<div class="portlet-boundary portlet-boundary_com_liferay_blogs_web_portlet_BlogsPortlet_ portlet-static portlet-static-end portlet-barebone portlet-blogs " id=
      "p_p_id_com_liferay_blogs_web_portlet_BlogsPortlet_">…</div>
      ▼<div class="portlet-boundary portlet-boundary_com_liferay_site_navigation_language_web_portlet_SiteNavigationLanguagePortlet_ portlet-static portlet-static-end
      portlet-decorate portlet-language " id="p_p_id_com_liferay_site_navigation_language_web_portlet_SiteNavigationLanguagePortlet_">
        ▼<section class="portlet" id="portlet_com_liferay_site_navigation_language_web_portlet_SiteNavigationLanguagePortlet">
          ▼<div class="portlet-content">
              <h2 class="portlet-title-text">Language Selector</h2>
            ▶<div class=" portlet-content-container">…</div>
            </div>
          </section>
        </div>
      ▶<div class="portlet-boundary portlet-boundary_com_liferay_document_library_web_portlet_DLPortlet_ portlet-static portlet-static-end portlet-decorate portlet-
      document-library " id="p_p_id_com_liferay_document_library_web_portlet_DLPortlet_INSTANCE_TmzDyEj6IVoL_">…</div>
      ▶<div class="portlet-boundary portlet-boundary_com_liferay_hello_world_web_portlet_HelloWorldPortlet_ portlet-static portlet-static-end portlet-decorate portlet-
      hello-world " id="p_p_id_com_liferay_hello_world_web_portlet_HelloWorldPortlet_">…</div>
      </div>
    </div>
   ::after
  </div>
</div>
```

Figure 109.1: Portlet Decorators add the decorator's CSS class to the application's wrapper

Once your Portlet Decorator is complete, apply it to your applications through the Look and Feel Configuration menu.



Use Custom Title
NO

Language Selector English (United States)

Link Application URLs to Page
Current Page

Application Decorators
Decorate

Figure 109.2: Portlet Decorators can be applied through the Look and Feel Configuration menu

Your portlet's decor is now in your hands.

109.1 Adding Portlet Decorators to a Theme

Portlet Decorators are associated with a particular theme. If your theme does not define any portlet decorators, none are available. It is recommended that you provide a few decorators for your portlets, to cover the basic use cases.

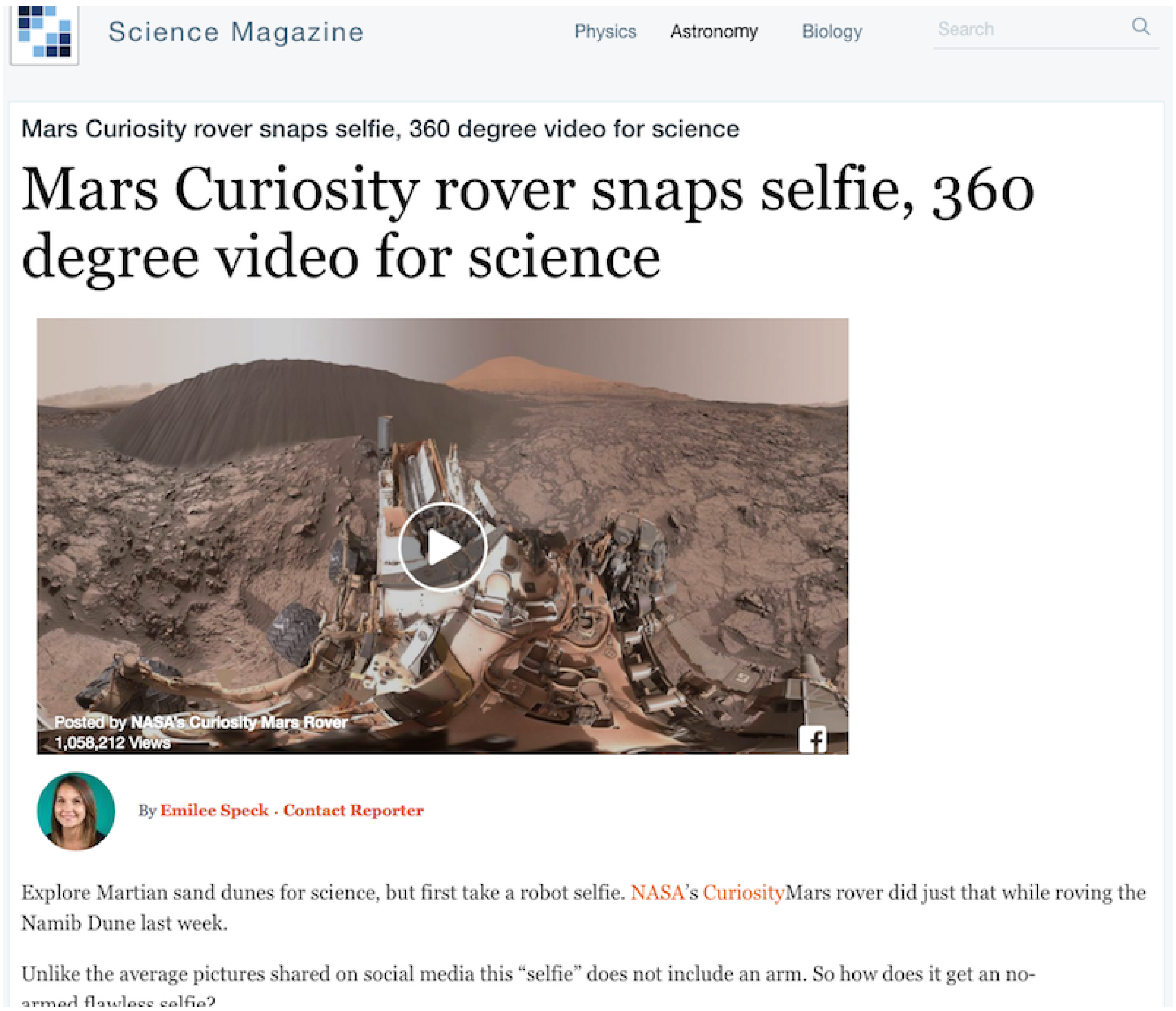
For example, the Liferay Portal CE 7.0 Classic theme includes three Portlet Decorators:

- Decorate: this is the default Application Decorator when using the Classic theme. When this decorator is applied, the portlet is wrapped in a white box with a border and the portlet title is displayed at the top.
- Borderless: when this decorator is applied, the portlet is no longer wrapped in a white box, but the portlet title is displayed at the top.
- Barebone: when this decorator is applied, neither the wrapping box nor the custom portlet title are shown. This option is recommended when you only want to display the bare portlet content.

Note: Upgrading to Liferay DXP will assign the *borderless* decorator automatically to those portlets that had the *Show Borders* option set to false in previous versions of Liferay.

As mentioned, you should consider updating your Liferay DXP Themes to provide at least one for the *decorate*, *borderless* and *barebone* use cases.

This tutorial demonstrates how to


Science Magazine

Mars Curiosity rover snaps selfie, 360 degree video for science

Mars Curiosity rover snaps selfie, 360 degree video for science

By Emilee Speck · Contact Reporter

Explore Martian sand dunes for science, but first take a robot selfie. NASA's CuriosityMars rover did just that while roving the Namib Dune last week.

Unlike the average pictures shared on social media this "selfie" does not include an arm. So how does it get an no-

Figure 109.3: The Classic theme's Decorate Application Decorator wraps the portlet in a white box.

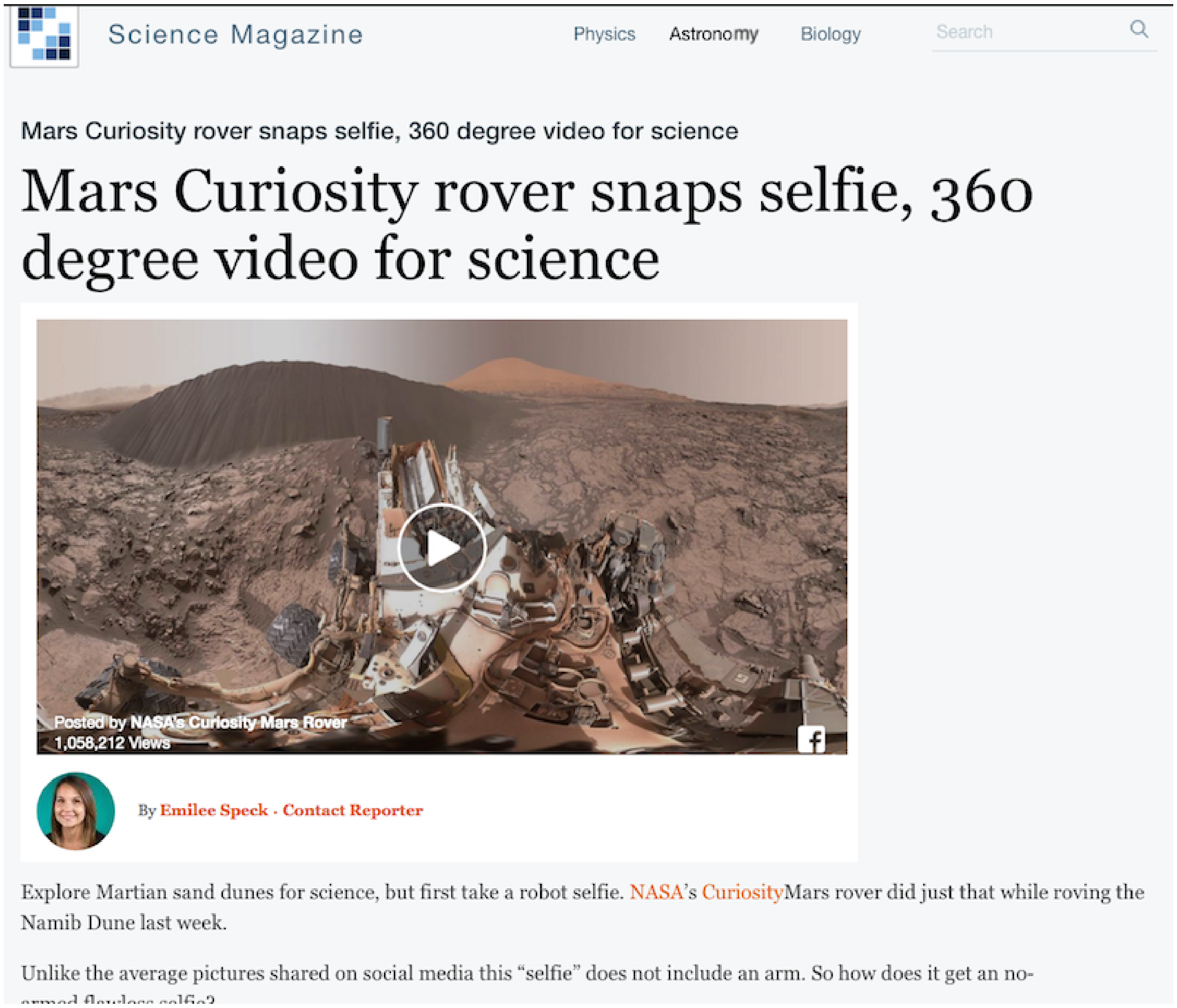
- Add Portlet Decorators to your theme
- Affect theme markup with Portlet Decorators

Now that you know why you should have Portlet Decorators in your theme, you can learn how to add them to your theme.

Adding Portlet Decorators to a Theme

Adding Portlet Decorators to your theme is similar to adding Color Schemes. You just have to follow these steps:

1. Configure your theme's `liferay-look-and-feel.xml`
2. Define the Application Decorator CSS styles
3. Optional: Add conditions to your theme's markup



Mars Curiosity rover snaps selfie, 360 degree video for science

Mars Curiosity rover snaps selfie, 360 degree video for science

By Emilee Speck · Contact Reporter

Explore Martian sand dunes for science, but first take a robot selfie. NASA's CuriosityMars rover did just that while roving the Namib Dune last week.

Unlike the average pictures shared on social media this "selfie" does not include an arm. So how does it get an no-

Figure 109.4: The Classic theme's Borderless Application Decorator displays the application's custom title.

Configuring liferay-look-and-feel.xml

The first thing you must do is declare the Portlet Decorators in your theme's `liferay-look-and-feel.xml`.

The Document Type Definition for the liferay-look-and-feel.xml in Liferay DXP contains the information and rules to add Portlet Decorators (in the code referred as portlet-decorators) to your theme.

Here is how the classic theme defines Portlet Decorators in its `liferay-look-and-feel.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE look-and-feel PUBLIC "-//Liferay//DTD Look and Feel 7.0.0//EN" "http://www.liferay.com/dtd/liferay-look-and-feel_7_0_0.dtd">

<look-and-feel>
    <compatibility>
        <version>7.0.0+</version>
    </compatibility>
    ...
        <theme id="classic" name="Classic">
        ...
        <portlet-decorator id="barebone" name="Barebone">
```

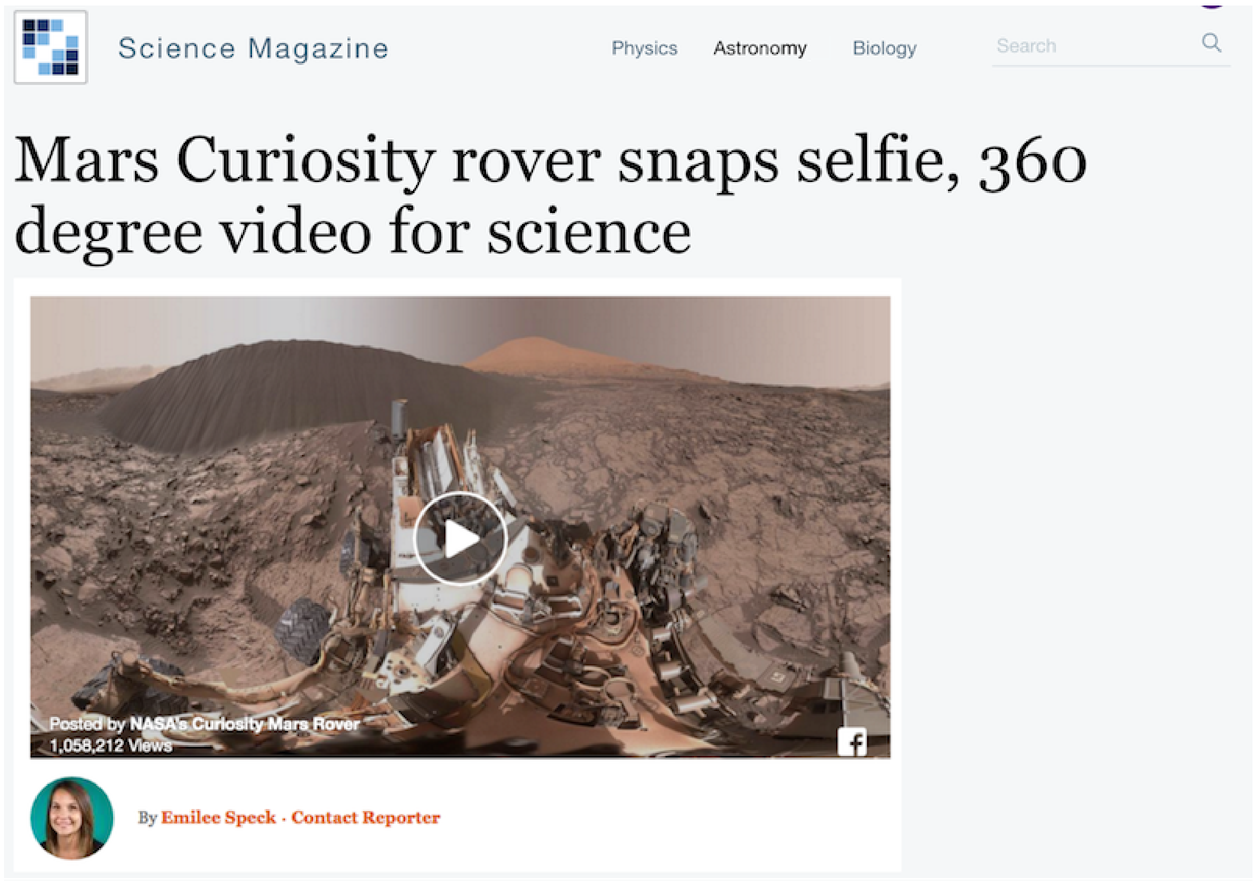

Science Magazine
Physics
Astronomy
Biology
Search
Mars Curiosity rover snaps selfie, 360 degree video for science
Posted by NASA's Curiosity Mars Rover
1,058,212 Views
By Emilee Speck · Contact Reporter

Figure 109.5: The Classic theme's Barebone Application Decorator displays only the application's content.

```
            <portlet-decorator-css-class>portlet-barebone</portlet-decorator-css-class>
        </portlet-decorator>
        <portlet-decorator id="borderless" name="Borderless">
            <portlet-decorator-css-class>portlet-borderless</portlet-decorator-css-class>
        </portlet-decorator>
        <portlet-decorator id="decorate" name="Decorate">
            <default-portlet-decorator>true</default-portlet-decorator>
            <portlet-decorator-css-class>portlet-decorate</portlet-decorator-css-class>
        </portlet-decorator>
        </theme>
</look-and-feel>
```

The `portlet-decorator` element contains all the information about the Application Decorator:

- id: this required attribute contains a unique string that identifies this specific Application Decorator. This is the value that is stored when applying an Application Decorator, and it can be used to refer to this decorator in your theme templates.
- name: this required attribute is a user friendly identifier for the Application Decorator to be displayed in the Look and Feel UI.
- portlet-decorator-css-class: this required element contains the name of the CSS class that is added to the application wrapping layer when this Application Decorator is applied.
- default-portlet-decorator: use this optional element to identify the default Application Decorator for your theme.

You can define as many Portlet Decorators as you want, but it's recommended to include at least one for the *decorate*, *borderless* and *barebone* use cases.

Define the Styles for Your Application Decorator CSS Class

Once you've declared your Portlet Decorators, it's time to define their effect in the application look and feel. While the previous step was straightforward, this depends on your creativity.

As an example, look at the `_portlet_decorator.scss` of the Classic theme:

```
.portlet-decorate .portlet-content {
    background: #FFF;
    border: 1px solid #DEEEEE;
}

.portlet-barebone .portlet-content {
    padding: 0;
}
```

Once your CSS styles are written, make sure to import the CSS file into your `_custom.scss`:

```
@import "portlet_decorator"
```

That's all that is required to add Portlet Decorators to your theme. If you want to modify your application's markup with your Portlet Decorators, read the next section.

Changing Your Application Markup with Portlet Decorators

So far you've seen how to use Portlet Decorators to change the look and feel of an application with styles.

It's possible to go a step further and alter the markup of your application based on the Application Decorator applied. For this, you must edit the `portlet.ftl` template for your theme, retrieve the `portletDecoratorId` of the current Application Decorator from the `portlet_display` object, and make some decisions based on it.

For example, this is how the Classic theme shows the application title when the *barebone* Application Decorator is not applied:

```
<#if portlet_display.getPortletDecoratorId() != "barebone">
        <h2 class="portlet-title-text">${portlet_title}</h2>
</#if>
```

There you have it! Now you know how to add Portlet Decorators to your theme. Let your creativity be your guide.

Related Topics

Themelets

Making Your Applications Configurable

109.2 Applying Portlet Decorators to Embedded Portlets

Once you have installed a theme that contains Portlet Decorators, site administrators can apply them to a portlet instance by selecting the Application Decorator in the Look and Feel Configuration dialog.

If your theme contains embedded portlets, it's also possible to apply an Application Decorator other than the default one by setting its preferences.

This tutorial demonstrates how to apply Portlet Decorators to Embedded Portlets in your theme.

Setting Application Decorator Preferences

To define a default Application Decorator for your theme's embedded portlets, you must set a default decorator in the portlet preferences.

For example, the Classic theme declares an Application Decorator with Id `barebone` and applies it to the embedded Navigation Menu portlet and Search portlet in its `navigation.ftl`:

```
<#assign VOID =
freeMarkerPortletPreferences.setValue("portletSetupPortletDecoratorId",
"barebone")>

<div aria-expanded="false" class="collapse navbar-collapse"
id="navigationCollapse">
    <#if has_navigation && is_setup_complete>
        <nav class="${nav_css_class} site-navigation"
        id="navigation" role="navigation">
            <div class="navbar-form navbar-right" role="search">
                <@liferay.search default_preferences=
                "${freeMarkerPortletPreferences}" />
            </div>

            <div class="navbar-right">
                <@liferay.navigation_menu default_preferences=
                "${freeMarkerPortletPreferences}" />
            </div>
        </nav>
    </#if>
</div>

<#assign VOID = freeMarkerPortletPreferences.reset()>
```

To set the default decorator for your embedded portlets, follow these steps:

1. Set the value for the `portletSetupPortletDecoratorId` to the Id of the Application Decorator you want to use:

   ```
   <#assign VOID =
   freeMarkerPortletPreferences.setValue("portletSetupPortletDecoratorId",
   "barebone")>
   ```

2. Next, set the `default_preferences` attribute of the portlet's tag to the `freeMarkerPortletPreferences` variable you just defined:

   ```
   <@liferay.search default_preferences= "${freeMarkerPortletPreferences}" />
   ```

Your embedded portlets now have a custom default Application Decorator!

Related Topics

Embedding Portlets in Themes

Providing Portlets to Manage Requests

109.3 Theming Portlets

Liferay DXP themes can provide additional styles to a portlet. You can change the markup for the portlet containers by modifying the `portlet.ftl` file.

This tutorial demonstrates how to style portlets with your themes.

Portlet FTL

Here is a quick look at the default `portlet.ftl` that's included in the default theme of 7.0:

```
<#assign
	portlet_display = portletDisplay

	portlet_back_url = htmlUtil.escapeHREF(portlet_display.getURLBack())
	portlet_content_css_class = "portlet-content"
	portlet_display_name = htmlUtil.escape(portlet_display.getPortletDisplayName())
	portlet_display_root_portlet_id = htmlUtil.escapeAttribute(portlet_display.getRootPortletId())
	portlet_id = htmlUtil.escapeAttribute(portlet_display.getId())
	portlet_title = htmlUtil.escape(portlet_display.getTitle())
/>
```

An explanation of each variable used in `portlet.ftl` is shown below:

- `portletDisplay`: is fetched from the `themeDisplay` object and contains information about the portlet.
- `portlet_back_url`: URL to return to the previous page with portlet `WindowState` is maximized.
- `portlet_display_name`: The "friendly" name of the portlet as displayed in the GUI.
- `portlet_display_root_portlet_id`: Sets the
- `portlet_id`: The ID of the portlet (not the same as the portlet namespace)
- `portlet_title`: The portlet name set in the portlet Java class (usually from a `Keys.java` class).

The following condition checks if the portlet header should be displayed. If the portlet has a portlet toolbar (Configuration, Permissions, Look and Feel), the condition is true:

```
<#if portlet_display.isPortletDecorate() && !portlet_display.isStateMax()
&& portlet_display.getPortletConfigurationIconMenu()??
&& portlet_display.getPortletToolbar()??>
```

Portlet title menus are used in portlets that allow you to add resources (Web Content Display, Media Gallery, Documents and Media). This is used to build a menu of items for adding resources:

```
portlet_title_menus = portlet_toolbar.getPortletTitleMenus(portlet_display_root_portlet_id, renderRequest, renderResponse)
```

The configuration below contains the information for the configuration menu (Configuration, Permissions, Look and Feel):

```
portlet_configuration_icons = portlet_configuration_icon_menu.getPortletConfigurationIcons(portlet_display_root_portlet_id, renderRequest, renderResponse)
```

The rest of the file contains the HTML markup for the portlet topper and the portlet content. It is possible to add CSS classes, change markup, or add custom information to the `portlet.ftl`. To provide a default style for all portlets, use the CSS classes found in this file, in conjunction with the portlet decorators to achieve the desired look and feel.

Portlet Decorators are explained in more detail next.

Portlet Decorators

In previous versions of Liferay DXP, administrators could display or hide the application borders through the Show Borders option of the look and feel configuration menu. In 7.0 this option has been replaced with Portlet Decorators, a more powerful mechanism to customize the style of the application wrapper.

The default portlet decorators are covered next.

Default Portlet Decorators

Themes come bundled with three default portlet decorators in their `liferay-look-and-feel.xml`. These are listed below:

- `Barebone`: when this decorator is applied, neither the wrapping box nor the custom application title are shown. This option is recommended when you only want to display the bare application content.
- `Borderless`: when this decorator is applied, the application is no longer wrapped in a white box, but the application custom title is displayed at the top.
- `Decorate`: this is the default Portlet Decorator when using the Classic theme. When this decorator is applied, the application is wrapped in a white box with a border and the application custom title is displayed at the top.

You can learn how to create and apply your own portlet decorators in the section dedicated to Portlet Decorators.

Now you know how to make your portlets stylish!

Related Topics

Portlet Decorators

Themes and Layout Templates

109.4 Embedding Portlets in Themes

One thing developers often want to do is embed a portlet in a theme. This makes the portlet visible on all pages where the theme is used. In the past, this was only possible by hard-coding a specific portlet into place, which has many drawbacks. Liferay DXP provides the *Portlet Providers* framework that requires you only specify the entity type and action to be displayed. Based on the given entity type and action, Liferay DXP determines which deployed portlet to use. This increases the flexibility and modularity of embedding portlets in Liferay DXP.

In this tutorial, you'll learn how to declare an entity type and action in a custom theme, and you'll create a module that finds the correct portlet to use based on those given parameters. You'll first learn how to embed portlets into a theme.

Adding a Portlet to a Custom Theme

The first thing you should do is open the template file for which you want to declare an embedded portlet. For example, the `portal_normal.ftl` template file is a popular place to declare embedded portlets. There are two ways two embed a portlet in a theme: by class name or by portlet name. Both methods are covered in this section.

Embedding a Portlet by Class Name

To embed a portlet by class name, insert the following declaration wherever you want to embed the portlet:

```
<@liferay_portlet["runtime"]
    portletProviderAction=ACTION
    portletProviderClassName="CLASS_NAME"
/>
```

This declaration expects two parameters: the type of action and the class name of the entity type the portlet should handle. Here's an example of an embedded portlet declaration that uses the class name:

```
<@liferay_portlet["runtime"]
    portletProviderAction=portletProviderAction.VIEW
    portletProviderClassName="com.liferay.portal.kernel.servlet.taglib.ui.LanguageEntry"
/>
```

This declares that the theme is requesting to view language entries. Liferay DXP determines which deployed portlet to use in this case by providing the portlet with the highest service ranking.

Note: In some cases, a default portlet is already provided to fulfill certain requests. You can override the default portlet with your custom portlet by specifying a higher service rank. To do this, set the following property in your class' `@Component` declaration:

```
property= {"service.ranking:Integer=20"}
```

Make sure you set the service ranking higher than the default portlet being used.

There are five different kinds of actions supported by the Portlet Providers framework: `ADD`, `BROWSE`, `EDIT`, `PREVIEW`, and `VIEW`. Specify the entity type and action in your theme's runtime declaration.

Great! Your theme declaration is complete. However, the Portal is not yet configured to handle this request. You must create a module that can find the portlet that fits the theme's request.

1. Create an OSGi module.

2. Create a unique package name in the module's src directory, and create a new Java class in that package. To follow naming conventions, name the class based on the entity type and action type, followed by *PortletProvider* (e.g., `SiteNavigationLanguageEntryViewPortletProvider`). The class should extend the `BasePortletProvider` class and implement the appropriate portlet provider interface based on the action you chose in your theme (e.g., ViewPortletProvider, BrowsePortletProvider, etc.).

3. Directly above the class's declaration, insert the following annotation:

   ```
   @Component(
       immediate = true,
       property = {"model.class.name=CLASS_NAME"},
       service = INTERFACE.class
   )
   ```

   The `property` element should match the entity type you specified in your theme declaration (e.g., `com.liferay.portal.kernel.servlet.taglib.ui.LanguageEntry`). Also, your `service` element should match the interface you're implementing (e.g., `ViewPortletProvider.class`). You can view an example of a similar `@Component` annotation in the RolesSelectorEditPortletProvider class.

4. Specify the methods you want to implement. Make sure to retrieve the portlet ID and page ID that should be provided when this service is called by your theme.

   A common use case is to implement the `getPortletId()` and `getPlid(ThemeDisplay)` methods. You can view the SiteNavigationLanguageViewPortletProvider for an example of how these methods can be implemented to provide a portlet for embedding in a theme. This example module returns the portlet ID of the Language portlet specified in SiteNavigationLanguagePortletKeys. Furthermore, it returns the PLID, which is the ID that uniquely identifies a page used by your theme. By retrieving these, your theme will know which portlet to use, and which page to use it on.

The only thing left to do is generate the module's JAR file and copy it to your Portal's `osgi/modules` directory. Once the module is installed and activated in your Portal's service registry, your embedded portlet is available for use wherever your theme is used.

You successfully requested a portlet based on the entity and action types required, and created and deployed a module that retrieves the portlet and embeds it in your theme.

Embedding a Portlet by Portlet Name

If you'd like to embed a specific portlet in the theme, you can hard code it by providing its instance ID and name:

```
<@liferay_portlet["runtime"]
    instanceId="INSTANCE_ID"
    portletName="PORTLET_NAME"
/>
```

Note: If your portlet is instanceable, an instance ID must be provided; otherwise, you can remove this line. To set your portlet to non-instanceable, set the property `com.liferay.portlet.instanceable` in the component annotation of your portlet to `false`.

The portlet name must be the same as `javax.portlet.name`'s value.

Here's an example of an embedded portlet declaration that uses the portlet name to embed a web content portlet:

```
<@liferay_portlet["runtime"]
    portletName="com_liferay_journal_content_web_portlet_JournalContentPortlet"
/>
```

You can also set default preferences for an application. This process is covered next.

Setting Default Preferences for an Embedded Portlet

Follow these steps to set default portlet preferences for an embedded portlet:

1. Temporarily assign the `VOID` variable to set portlet preferences using the `freeMarkerPortletPreferences` object as shown in the example below:

   ```
   <#assign VOID = freeMarkerPortletPreferences.setValue(
   "portletSetupPortletDecoratorId", "barebone") />
   ```

2. Set the `defaultPreferences` attribute to use the `freeMarkerPortletPreferences` object you just configured:

   ```
   <@liferay_portlet["runtime"]
       defaultPreferences="${freeMarkerPortletPreferences}"
       portletName="com_liferay_login_web_portlet_LoginPortlet"
   />
   ```

3. Once the preferences are set and passed to your portlet, reset the `freeMarkerPortletPreferences` object so it can be fresh for the next portlet:

   ```
   <#assign VOID = freeMarkerPortletPreferences.reset() />
   ```

Now you know how to set default preferences for embedded portlets! Next you can see the additional attributes you can use for your embedded portlets.

Additional Attributes for Portlets

Below are some additional attributes you can define for embedded portlets:

defaultPreferences: A string of Portlet Preferences for the application. This includes look and feel configurations.

instanceId: The instance ID for the app, if the application is instanceable.

persistSettings: Whether to have an application use its default settings, which will persist across layouts. The default value is *true*.

settingsScope: Specifies which settings use for the application. The default value is `portletInstance`, but can be set to `group` or `company`.

Now you know how to embed a portlet in your theme by class name and by portlet name and how to configure your embedded portlet!

Related Topics

Providing Portlets to Manage Requests

Portlets

Service Builder

CHAPTER 110

LEXICON CSS AND THEMES

Lexicon CSS is an extension of Bootstrap's CSS Framework. Bootstrap is by far the most popular CSS framework on the web. Also, it's open source, so anyone can use it. Built with Sass, Lexicon CSS fills the front-end gaps between Bootstrap and the specific needs of Liferay DXP.

These tutorials look briefly at Lexicon CSS and show you how to use it in your Liferay DXP themes.

110.1 Importing Lexicon CSS into a Theme

As mentioned before, Lexicon CSS fills the gaps between Bootstrap and the specific needs of Liferay DXP. Bootstrap features have been extended to cover more use cases. Here are some of the new components added by Lexicon CSS:

- Aspect Ratio
- Cards
- Dropdown Wide and Dropdown Full
- Figures
- Nameplates
- Sidebar / Sidenav
- Stickers
- SVG Icons
- Timelines
- Toggles

Several reusable CSS patterns have also been added to help accomplish time consuming tasks such as these:

- truncating text
- content filling the remaining container width
- truncating text inside table cells
- table cells filling remaining container width and table cells only being as wide as their content
- open and close icons inside collapsible panels
- nested vertical navigations
- slide out panels

- notification icons/messages
- vertical alignment of content

Next you can learn more about Lexicon's structure.

Lexicon CSS Structure

Lexicon CSS is bundled with two sub-themes: Lexicon Base and Atlas. Lexicon Base is Liferay DXP's Bootstrap API extension. It is also the theme that is used in Liferay DXP's Styled Theme. It adds all the features and components you need and inherits Bootstrap's styles. As a result, Lexicon Base is fully compatible with third party themes that leverage Bootstrap's Sass variable API. As a best practice, you should use the Lexicon Base as your base theme to integrate third party themes into Liferay DXP.

Atlas is Liferay DXP's custom Bootstrap theme that is used in the Classic Theme. Its purpose is to overwrite and manipulate Bootstrap and Lexicon Base to create Liferay DXP's classic look and feel. Atlas is equivalent to installing a Bootstrap third party theme.

Note: It is not recommended to integrate third party themes with Atlas, as it adds variables and styles that are outside the scope of Bootstrap's API.

You can learn how to customize the Atlas theme next.

Customizing Atlas in Liferay DXP

If you want to include all the Classic Theme's files, you can skip these steps and move on to the next section.

Follow these steps to customize the Atlas theme:

1. In your theme's `/src/css` directory (for legacy ant themes, place in `/_diff/css`) add a file named `aui.scss` with the code below and save:

```
@import "aui/lexicon/atlas";
```

2. Add a file named `_imports.scss` with the code below and save:

```
@import "bourbon";
@import "mixins";
@import "aui/lexicon/atlas-variables";
@import "aui/lexicon/bootstrap/mixins";
@import "aui/lexicon/lexicon-base/mixins";
@import "aui/lexicon/atlas-theme/mixins";
```

3. Add a file named `_aui_variables.scss` with the code below and save:

```
// Icon paths

$FontAwesomePath: "aui/lexicon/fonts/alloy-font-awesome/font";
$font-awesome-path: "aui/lexicon/fonts/alloy-font-awesome/font";
$icon-font-path: "aui/lexicon/fonts/";
```

All your Atlas, Bootstrap, and Lexicon Base variable modifications must be placed in this file.

4. Add a file named `_custom.scss` with the code below and save:

```
    /* Use these inject tags to dynamically create imports for
    themelet styles. You can place them where ever you like in this file. */

    /* inject:imports */
    /* endinject */

    /* This file allows you to override default styles in one central
    location for easier upgrade and maintenance. */
```

Place your custom CSS in this file. Next you can learn how to extend Atlas with the Classic theme.

Extending Atlas with the Classic Theme

To extend the Atlas theme with the Classic theme, copy all the files located in these directories into your theme:

```
frontend-theme-classic/src/css
frontend-theme-classic/src/images
frontend-theme-classic/src/js
frontend-theme-classic/src/templates
```

You can also automatically copy these files into your theme using the Liferay Theme Tasks module gulp kickstart command and following the prompts.

Next you can learn how to customize the Lexicon Base.

Customizing Lexicon Base

You can customize Lexicon Base with just a few imports.

In your custom theme's `/src/css` directory (for legacy ant themes, place in `/_diff/css`) add a file named `_aui_variables.scss` with the code below and save:

```
// Icon paths

    $FontAwesomePath: "aui/lexicon/fonts/alloy-font-awesome/font";
    $font-awesome-path: "aui/lexicon/fonts/alloy-font-awesome/font";
    $icon-font-path: "aui/lexicon/fonts/";
```

All your Atlas, Bootstrap, and Lexicon Base variable modifications must be placed in this file.

As mentioned earlier, any custom CSS should be placed in `_custom.scss`.

You can learn how to add third party themes in Liferay DXP next.

Adding a Third Party Theme

Third party themes must be built with Sass to be compatible with Liferay DXP. **Make sure the Sass files are included before making any theme purchase.**

Follow these steps to add a third party theme:

1. Follow the steps above in the Customizing Lexicon Base section.

2. Create a folder inside `/src/css` (for legacy ant themes, `/_diff/css`) that contains your third party theme (e.g. `/src/css/awesome-theme` or `/_diff/css/awesome-theme`)

3. Copy the contents of the theme to the folder you just created.

4. In `_aui_variables.scss`, import the file containing the theme variables. For example, `@import "awesome-theme/variables.scss";`

```
**Note:** You may omit the leading underscore when importing Sass files.
```

---

5. In `_custom.scss`, import the file containing the CSS. For example, `@import "awesome-theme/main.scss";`
6. Deploy your theme with `gulp deploy` (for legacy ant themes, use `ant deploy`)

Now you know how to use Lexicon CSS in your theme!

Related Topics

Applying Lexicon Styles to Your App

Chapter 111

Product Navigation

Liferay's product navigation consists of the main menus you use to customize, configure, and navigate your Liferay instance. Whether you edit a page, switch to a different site scope, access a user's credentials, etc., you're constantly using the default navigation menus. Liferay's product navigation is designed to be intuitive and extensive, but often times, providing a customization to a default menu is necessary to give your Liferay instance that unique touch you're searching for. Liferay allows developers to extend and customize the default product navigation to fit their needs.

There are four main sections of Liferay's product navigation that you can extend: Product Menu, Control Menu, Simulation Menu, and User Personal Bar.

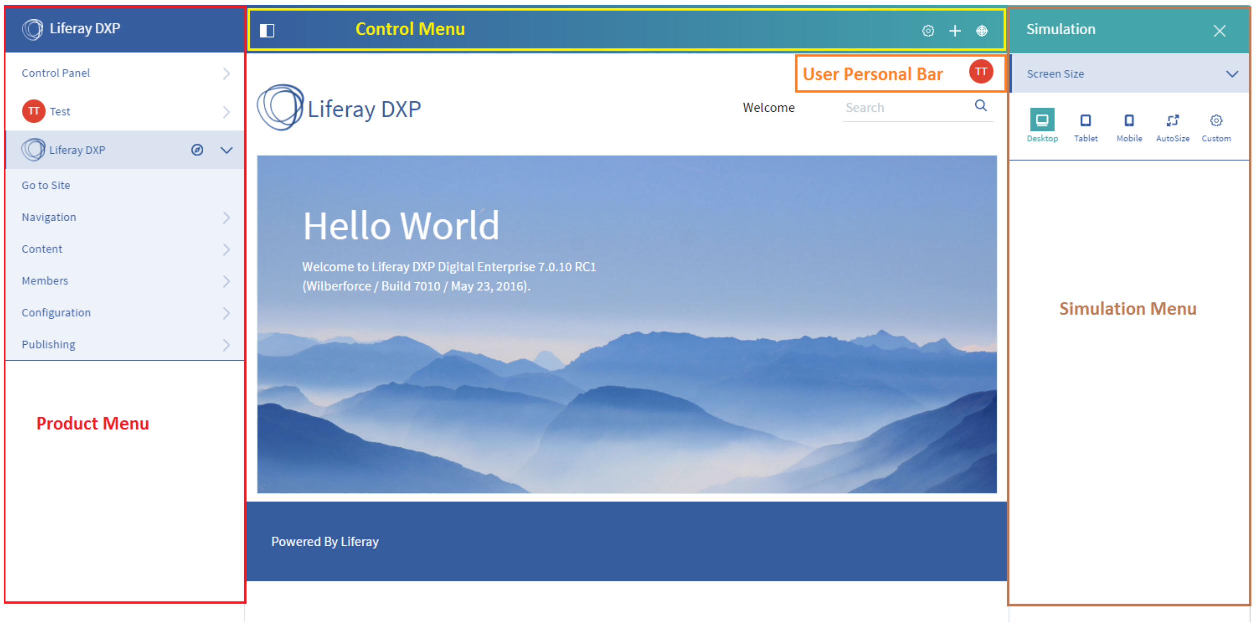


Figure 111.1: The main product navigation menus include the Product Menu, Control Menu, Simulation Menu, and User Personal Bar.

As you can see from the figure above, the Product Menu is the menu to the left that displays your's instance's Control Panel, user account settings, and Site Administration functionality. The Control Menu is the top menu offering navigation to the Product Menu, Simulation Menu (the right menu), and *Add* button.

When certain settings are enabled (e.g., Staging, Page Customization, etc.) more tools are offered. The Simulation Menu offers options to simulate your site's look for different scenarios (devices, user segments, etc.). Lastly, the User Personal Bar is used to hold selectable items that aid with a user's own account settings.

In this section of tutorials, you'll learn about the various ways you can extend and customize Liferay's product navigation to fit your needs.

111.1 Customizing the Product Menu

Liferay's Product Menu comes with three major sections to choose from, by default: the Control Panel, User Settings, and Site Administration. These options are called panel categories, which is the term used to differentiate between sections of the menu. For instance, the Control Panel is a single panel category, and when clicking on it, you're presented with four other child panel categories: *Users*, *Sites*, *Apps*, and *Configuration*. It you click on one of these child panel categories, you're presented with panel apps.

Note: The Product Menu cannot be changed by applying a new theme. To change the layout/style of the Product Menu, you must create and deploy a theme contributor. See the Theme Contributors tutorial for more details.

This construction of the Product Menu was designed to be intuitive and easy to use. For your instance of Liferay, however, you may want to add other panel categories with custom panel apps. Also, you may desire to change the order of your panel categories and/or apps. In this tutorial, you'll learn how to provide your own custom or modify existing panel categories and panel apps for the Product Menu.

Adding Custom Panel Categories

Liferay provides an easy way to extend the Product Menu and customize it to display what is most helpful in your particular situation. First, you'll learn how to add a panel category.

1. Create a generic OSGi module using your favorite third party tool, or use Blade CLI. Blade CLI offers a Panel App template, which you can use to generate a basic panel category and panel app.

2. Create a unique package name in the module's src directory and create a new Java class in that package. To follow naming conventions, give your class a unique name followed by *PanelCategory* (e.g., `ControlPanelCategory`).

3. Directly above the class's declaration, insert the following annotation:

```
@Component(
    immediate = true,
    property = {
        "panel.category.key=" + [Panel Category Key],
        "panel.category.order:Integer=[int]"
    },
    service = PanelCategory.class
)
```

   The `property` element designates two properties that should be assigned for your category. The `panel.category.key` specifies the parent category for your custom category. You can find popular parent categories to assign in the PanelCategoryKeys class. For instance, if you wanted to create a child category in the Control Panel, you could assign `PanelCategoryKeys.CONTROL_PANEL`. Likewise, if you wanted to create a root category, like the Control Panel or Site Administration, you could assign `PanelCategoryKeys.ROOT`.

The `panel.category.order:Integer` property specifies the order in which your category is displayed. The higher the number (integer), the lower your category is listed among other sibling categories assigned to a parent.

Lastly, your `service` element should specify the `PanelCategory.class` service. You can view an example of a similar `@Component` annotation for the `UserPanelCategory` class below.

```
@Component(
    immediate = true,
    property = {
        "panel.category.key=" + PanelCategoryKeys.ROOT,
        "panel.category.order:Integer=200"
    },
    service = PanelCategory.class
)
```

4. Implement the `PanelCategory` interface. A popular way to do this is by extending the BasePanelCategory or BaseJSPPanelCategory abstract classes. Typically, the `BasePanelCategory` is extended for basic categories (e.g., the Control Panel category) that only display the category name or other simple functionality. If you'd like to provide a custom UI for your panel, you can do so using any frontend technology, you only need to implement the methods `include()` or `includeHeader()` from the `PanelCategory` interface. The `includeHeader` method is used to render the header of the section and the `include` method is used to render the body. Implementing a custom UI gives you the flexibility to add more complex functionality. If you are going to use JSPs as the frontend technology, a base class called `BaseJSPPanelCategory` can be extended that already implements the methods `include()` and `includeHeader()` for you. This will be elaborated on more extensively later.

```
**Note:** In this tutorial, JSPs are used to describe how to provide
functionality to panel categories and apps. JSPs, however, are not the only
way to provide frontend functionality to your categories/apps. You can
create your own class implementing `PanelCategory` to use other
technologies, such as FreeMarker.
```

---

5. Since you're implementing the PanelCategory interface, you'll need to implement its methods if you're not extending a base class:

    - `getNotificationCount`: returns the number of notifications to be shown in the panel category.
    - `include`: renders the body of the panel category.
    - `includeHeader`: renders the panel category header.
    - `isActive`: whether the panel is selected.
    - `isPersistState`: whether to persist the panel category's state to the database. This is used to save the state of the panel category when navigating away from the menu.

6. Add any other methods that are necessary to create your custom panel category. As you learned earlier, you can extend the `BasePanelCategory` and `BaseJSPPanelCategory` abstract classes to implement `PanelCategory`.

If you'd like to provide something simple for your panel category like a name, extending `BasePanelCategory` is probably sufficient. For example, the ControlPanelCategory extends `BasePanelCategory` and specifies a `getLabel` method to set and display the panel category name.

```
@Override
public String getLabel(Locale locale) {
    return LanguageUtil.get(locale, "control-panel");
}
```

If you'd like to provide functionality that is more complex, you can use JSPs or any other similar technology to render the panel category. You can easily do this by extending `BaseJSPPanelCategory`. For example, the SiteAdministrationPanelCategory specifies the `getHeaderJspPath` and `getJspPath` methods. You could create a JSP with the UI you'd like to render and specify its path in methods like these:

```
@Override
public String getHeaderJspPath() {
    return "/sites/site_administration_header.jsp";
}

@Override
public String getJspPath() {
    return "/sites/site_administration_body.jsp";
}
```

One JSP is responsible for rendering the panel category's header (displayed when panel is collapsed) and the other for its body (displayed when panel is expanded).

You will also need to specify the servlet context from where you are loading the JSP files. If this is inside an OSGi module, make sure your `bnd.bnd` file has defined a web context path:

```
Bundle-SymbolicName: com.sample.my.module.web
Web-ContextPath: /my-module-web
```

And then reference the Servlet context using the symbolic name of your module like this:

```
@Override
@Reference(
    target = "(osgi.web.symbolicname=com.sample.my.module.web)",
    unbind = "-"
)
public void setServletContext(ServletContext servletContext) {
    super.setServletContext(servletContext);
}
```

Excellent! You've successfully created a custom panel category to display in the Product Menu. In many cases, a panel category holds panel apps for users to access. You'll learn about how to add a panel app to a panel category next.

Adding Custom Panel Apps

Just as adding panel categories is straight-forward and dynamic, so too is the process for adding panel apps. Panel apps are, by default, links provided in a panel category that allow you to access an application. For instance, if you navigate to the Site Administration → *Content* panel category, you can select the *Web Content* option, which is a panel app that allows you to access web content. Panel apps can also have a custom UI in the same way Panel categories could have a more complex UI.

Follow the steps below to add a panel app to your Liferay instance's Product Menu.

1. Create a generic OSGi module using your favorite third party tool, or use Blade CLI. Blade CLI offers a Panel App template, which you can use to generate a basic panel category and panel app.

2. Create a unique package name in the module's `src` directory and create a new Java class in that package. To follow naming conventions, give your class a unique name followed by *PanelApp* (e.g., `JournalPanelApp`).

3. Directly above the class's declaration, insert the following annotation:

```
@Component(
    immediate = true,
    property = {
        "panel.app.order:Integer=INTEGER"
        "panel.category.key=" + PANEL_CATEGORY_KEY,
    },
    service = PanelApp.class
)
```

   These properties and attributes are very similar to the ones discussed for panel categories. The `panel.app.order:Integer` property specifies the order your panel app is listed among other panel apps in the same category. The `panel.category.key` specifies the panel category your panel app will reside in. For example, if you want to add a panel app to Site Administration → *Content*, you would add the following property:

```
"panel.category.key=" + PanelCategoryKeys.SITE_ADMINISTRATION_CONTENT
```

   Visit the PanelCategoryKeys class for keys you can use to specify default panel categories in Liferay.

   Lastly, be sure to set the `service` attribute to `PanelApp.class`. You can view an example of a similar `@Component` annotation for the `JournalPanelApp` class below.

```
@Component(
    immediate = true,
    property = {
        "panel.app.order:Integer=100",
        "panel.category.key=" + PanelCategoryKeys.SITE_ADMINISTRATION_CONTENT
    },
    service = PanelApp.class
)
```

4. Implement the `PanelApp` interface. A popular way to do this is by extending the BasePanelApp abstract class. Just as you learned in the previous sub-section on panel categories, if you need to create a more complex UI to render in the panel, you can do so. If you want to use JSPs to render that UI, you can extend an additional abstract class which extends `BasePanelApp` called BaseJSPPanelApp. This provides

additional methods you can use to incorporate JSP functionality into your app's listing in the Product Menu.

JSPs are not the only way to provide frontend functionality to your panel apps. You can create your own class implementing `PanelCategory` to use other technologies, such as FreeMarker.

5. Since you're implementing the PanelApp interface, you must implement its methods if you're not extending a base class. The `BlogsPanelApp` is a simple example of how to specify your portlet as a panel app. This class extends the `BasePanelApp`, overriding the `getPortletId` and `setPortlet` methods. These methods specify and set the Blogs portlet as a panel app.

   Each panel app must belong to a portlet and each portlet can have at most one panel app. If more than one panel app is needed, another custom portlet must be created. By default, the panel app will only be shown if the user has permission to view the associated portlet.

   This is how those methods look for the Blogs portlet:

```
@Override
public String getPortletId() {
    return BlogsPortletKeys.BLOGS_ADMIN;
}

@Override
@Reference(
    target = "(javax.portlet.name=" + BlogsPortletKeys.BLOGS_ADMIN + ")",
    unbind = "-"
)
public void setPortlet(Portlet portlet) {
    super.setPortlet(portlet);
}
```

   Liferay DXP also lets you customize your panel app's appearance in the Product Menu. As you learned before, the `BaseJSPPanelApp` abstract class can be extended to provide further functionality with JSPs. For instance, the Navigation category in Site Administration offers a dynamic Pages panel app that provides much more than a simple link to access a portlet. This is accomplished by extending `BaseJSPPanelApp` in the GroupPagesPanelApp class, which provides this functionality in the Product Menu.

   In `GroupPagesPanelApp`, notice that the portlet ID is still returned similarly to the previous `BlogsPanelApp` example, but a `getJspPath` method is also called, which gives the panel app much more functionality provided by the `layouts_tree` JSP file:

```
@Override
public String getJspPath() {
    return "/panel/app/layouts_tree.jsp";
}
```

   Since you're including custom JSPs in your module, you'll also need to set the right `ServletContext`.

```
@Override
@Reference(
    target = "(osgi.web.symbolicname=com.liferay.layout.admin.web)",
    unbind = "-"
)
public void setServletContext(ServletContext servletContext) {
    super.setServletContext(servletContext);
}
```

Now you know how to add or modify a panel app in the Product Menu. Not only does Liferay provide a simple solution to add new panel categories and apps, it also gives you the flexibility to add a more complex UI to the Product Menu using any technology.

111.2 Customizing the Control Menu

The Control Menu is the most visible and accessible menu in Liferay. It is visible to the user in most places, always displaying helpful text or options at the top of the page. For example, on your home page, the Control Menu offers default options for accessing the Product Menu, Simulation Menu, and Add Menu. You can think of this menu as the gateway to configuring options in Liferay.

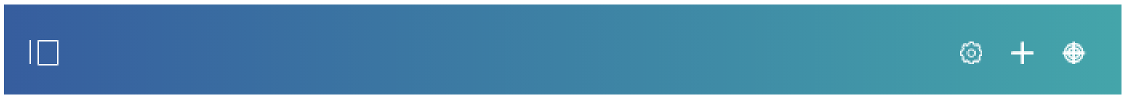


Figure 111.2: The Control Menu has three configurable areas: left, right, and middle.

If you navigate away from the home page, the Control Menu adapts and provides helpful functionality for what ever option you're using. For example, if you navigated to Site Administration → *Content* → *Web Content*, you'd be displayed a Control Menu with different functionality tailored for that option.

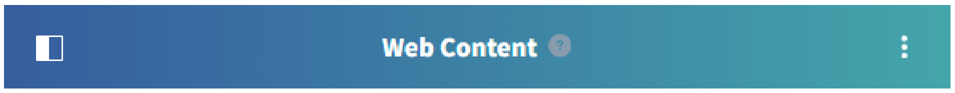


Figure 111.3: When switching your context to web content, the Control Menu adapts to provide helpful options for that area.

The default Control Menu is made up of three categories that represent the left, middle, and right portions of the menu. You can create navigation entries for each category, which can provide options or further navigation for the particular screen you're on.

Note: You can add the Control Menu to a custom theme by adding the following snippet into your `portal_normal.ftl`:

```
<@liferay.control_menu />
```

The other product navigation menus (e.g., Product Menu, Simulation Menu) are included in this tag, so specifying the above snippet will embed all three menus into your theme. Embedding the User Personal Bar is slightly different. Visit the Providing the User Personal Bar tutorial for more information.

You can reference a sample Control Menu Entry by visiting the Control Menu Entry article.

In this tutorial, you'll learn how to create your own entries to customize the Control Menu. Make sure to read the Adding Custom Panel Categories before beginning this tutorial. This tutorial assumes you have knowledge on creating a panel category. You'll begin by creating an entry for the Control Menu.

1. Create a generic OSGi module using your favorite third party tool, or use Blade CLI. Your module must contain a Java class, `bnd.bnd` file, and build file (e.g., `build.gradle` or `pom.xml`). You'll create your Java class next if your project does not already define one.

2. Create a unique package name in the module's src directory and create a new Java class in that package. To follow naming conventions, give your class a unique name followed by *ProductNavigationControlMenuEntry* (e.g., `StagingProductNavigationControlMenuEntry`).

3. Directly above the class's declaration, insert the following annotation:

```
@Component(
    immediate = true,
    property = {
        "product.navigation.control.menu.category.key=" + [Control Menu Category],
        "product.navigation.control.menu.category.order:Integer=[int]"
    },
    service = ProductNavigationControlMenuEntry.class
)
```

The `product.navigation.control.menu.category.key` property specifies the category your entry should reside in. As mentioned previously, the default Control Menu provides three categories: Sites (left portion), Tools (middle portion), and User (right portion).

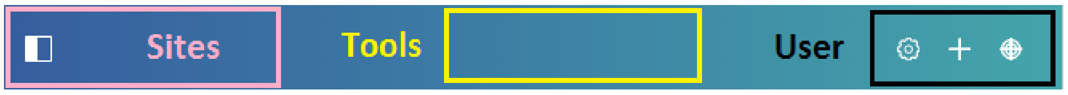


Figure 111.4: This image shows where your entry will reside depending on the category you select.

To specify the category, reference the appropriate key in the ProductNavigationControlMenuCategoryKeys class. For example, the following property would place your entry in the middle portion of the Control Menu:

```
"product.navigation.control.menu.category.key=" + ProductNavigationControlMenuCategoryKeys.TOOLS
```

Similar to panel categories, you'll also need to specify an integer for the order in which your entry will be displayed in the category. Entries are ordered from left to right. For example, an entry with order 1 will be listed to the left of an entry with order 2. If the order is not specified, it's chosen at random based on which service was registered first in the OSGi container. Lastly, your service element should specify the `ProductNavigationControlMenuEntry.class` service.

4. Implement the ProductNavigationControlMenuEntry interface. You can also extend the BaseProductNavigationControlMenuEntry or BaseJSPProductNavigationControlMenuEntry abstract classes. Typically, the `BaseProductNavigationControlMenuEntry` is extended for basic entries (e.g., `IndexingProductNavigationControlMenuEntry`) that only display a link with text or a simple icon. If you'd like to provide a more complex UI, like buttons or a sub-menu, you can do so by overriding the `include()` and `includeBody()` methods. If you are going to use JSPs for generating the UI, you can extend `BaseJSPProductNavigationControlMenuEntry` to save time. This will be elaborated on more extensively in the next step.

5. Define your Control Menu entry. You'll explore two examples to discover some options you have available for defining your entry. First, let's take a look at a simple example for providing text and an icon. The IndexingProductNavigationControlMenuEntry extends the `BaseProductNavigationControlMenuEntry` class and is used when Liferay is indexing. For this process, the indexing entry is displayed in the *Tools* (middle) area of the Control Menu with a *Refresh* icon and text stating *The Portal is currently indexing*. The icon is defined by calling the following method:

```
@Override
public String getIcon(HttpServletRequest request) {
    return "reload";
}
```

By default, Lexicon icons are expected to be returned. This is because the `BaseProductNavigationControlMenuEntry.getMa` method returns `lexicon`. To view all the Lexicon icons available, see https://liferay.github.io/clay /content/icons-lexicon/. You can also return FontAwesome icons, but you must implement the ProductNavigationControlMenuEntry.getMarkupView(...) method in your class and have it return `null`. Then you can return FontAwesome icons for the `getIcon(...)` method. To view all the FontAwesome icons available, see the FontAwesome 4.6.1 docs.

You can also provide a label for the Control Menu entry that displays when hovering over it with your pointer. This label is stored in the module's resource bundle, which you can learn more about in the Internationalization tutorials.

```
@Override
public String getLabel(Locale locale) {
    ResourceBundle resourceBundle = ResourceBundleUtil.getBundle(
        "content.Language", locale, getClass());

    return LanguageUtil.get(
        resourceBundle, "the-portal-is-currently-reindexing");
}
```

To do this, you'll need to create a `Language.properties` for your module.

You also have the option to provide a Lexicon or CSS icon in your `*ControlMenuEntry`. To use a Lexicon icon, you should override the methods in `ProductMenuProductNavigationControlMenuEntry` like the following:

```
public String getIconCssClass(HttpServletRequest request) {
    return "";
}

public String getIcon(HttpServletRequest request) {
    return "lexicon-icon";
}

public String getMarkupView(HttpServletRequest request) {
    return "lexicon";
}
```

Likewise, you can use a CSS icon by overriding the `ProductMenuProductNavigationControlMenuEntry` methods like the following:

```
public String getIconCssClass(HttpServletRequest request) {
    return "icon-css";
}

public String getIcon(HttpServletRequest request) {
    return "";
}

public String getMarkupView(HttpServletRequest request) {
    return "";
}
```

The icons used in the two examples for Lexicon and CSS icons can be found in the icons-lexicon and icons-font-awesome components, respectively.

The ProductMenuProductNavigationControlMenuEntry is a more advanced example. This entry displays in the *Sites* (left) area of the Control Menu, but unlike the previous example, it extends the

BaseJSPProductNavigationControlMenuEntry class. This provides several more methods that lets you use JSPs to define your entry's UI. There are two methods to pay special attention to:

```
@Override
public String getBodyJspPath() {
    return "/portlet/control_menu/product_menu_control_menu_entry_body.jsp";
}

@Override
public String getIconJspPath() {
    return "/portlet/control_menu/product_menu_control_menu_entry_icon.jsp";
}
```

The `getIconJspPath()` method provides the Product Menu icon (◧ → ◨) and the `getBodyJspPath()` method adds the UI body for the entry outside of the Control Menu. The latter method must be used when providing a UI outside the Control Menu. You can easily test this when you open and close the Product Menu on the home page.

Lastly, if you're planning on providing functionality that will stay exclusively inside the Control Menu, the `StagingProductNavigationControlMenuEntry` class calls its JSP like this:

```
@Override
public String getIconJspPath() {
    return "/control_menu/entry.jsp";
}
```

In particular, the `entry.jsp` is returned, which embeds the Staging Bar portlet into the Control Menu.

You will also need to specify the servlet context from where you are loading the JSP files. If this is inside an OSGi module, make sure your `bnd.bnd` file has defined a web context path:

```
Bundle-SymbolicName: com.sample.my.module.web
Web-ContextPath: /my-module-web
```

And then reference the Servlet context using the symbolic name of your module like this:

```
@Override
@Reference(
    target = "(osgi.web.symbolicname=com.sample.my.module.web)",
    unbind = "-"
)
public void setServletContext(ServletContext servletContext) {
    super.setServletContext(servletContext);
}
```

6. Define when to display your new entry in the Control Menu. As you've learned already, the Control Panel displays different entries depending on the page you've navigated to. You can specify when your entry should display using the `isShow(HttpServletRequest)` method.

   For example, the `IndexingProductNavigationControlMenuEntry` class queries the number of indexing jobs when calling `isShow`. If the query count is 0, then the indexing entry is not displayed in the Control Menu:

```
@Override
public boolean isShow(HttpServletRequest request) throws PortalException {
    int count = _indexWriterHelper.getReindexTaskCount(
        CompanyConstants.SYSTEM, false);

    if (count == 0) {
        return false;
    }

    return super.isShow(request);
}
```

The `StagingProductNavigationControlMenuEntry` class is selective over which pages to display for. The staging entry is configured to never display if the page is an administration page (e.g., *Site Administration*, *My Account*, etc.):

```
@Override
public boolean isShow(HttpServletRequest request) throws PortalException {
    ThemeDisplay themeDisplay = (ThemeDisplay)request.getAttribute(
        WebKeys.THEME_DISPLAY);

    Layout layout = themeDisplay.getLayout();

    // This controls if the page is an Administration Page

    if (layout.isTypeControlPanel()) {
        return false;
    }

    // This controls if Staging is enabled

    if (!themeDisplay.isShowStagingIcon()) {
        return false;
    }

    return true;
}
```

7. Define the dependencies for your Control Menu Entry. This should be completed in your build file (e.g., `build.grade` or `pom.xml`). For example, some popular dependencies (in Gradle format) are defined below:

```
dependencies {
    compileOnly group: "com.liferay", name: "com.liferay.product.navigation.control.menu.api", version: "[VERSION]"
    compile group: "com.liferay", name: "com.liferay.product.navigation.taglib", version: "[VERSION]"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "[VERSION]"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "[VERSION]"
    compile group: "javax.servlet.jsp", name: "javax.servlet.jsp-api", version: "[VERSION]"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "[VERSION]"
}
```

Your project may require more dependencies, depending on your module's functionality.

Excellent! You've created your entry in one of the three default panel categories in the Control Menu. You learned a basic way and an advanced way of providing that entry, and learned how to apply both.

111.3 Extending the Simulation Menu

When testing how Liferay pages and apps will appear for users, it's critical to simulate their views on as many useful ways as possible. By default, Liferay provides the Simulation Menu on the right-side of the main page. What if, however, you'd like to simulate something in Liferay that is not provided by the Simulation Menu? You'll need to extend the Simulation Menu, of course! Luckily, Liferay offers a simple way to extend and customize the Simulation Menu so you can test what you need. In this tutorial, you'll learn how to add additional functionality to the menu so you can do more simulating and less wondering.

The first thing you'll need to do is get accustomed to using panel categories/apps. This is covered in detail in the Customizing The Product Menu tutorial. Once you know the difference between panel categories and panel apps, and know how to create them, continue on in this tutorial.

There are few differences between the Simulation Menu and Product Menu, mostly because they extend the same base classes. The Simulation Menu, by default, is made up of only one panel category and one panel app. Liferay provides the SimulationPanelCategory class, which is a hidden category needed to hold the `DevicePreviewPanelApp`. This is the app and functionality you see in the Simulation Menu by default.

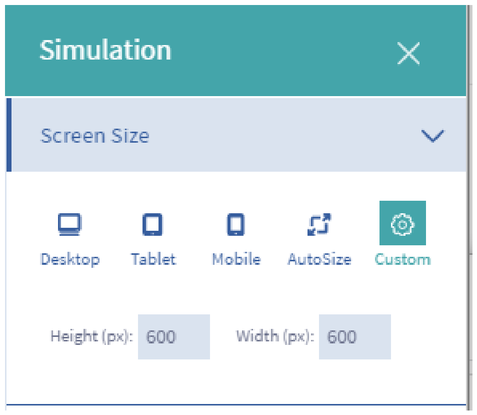


Figure 111.5: The Simulation Menu offers a device preview application.

To provide your own functionality in the Simulation Menu, you'll need to create a panel app in the `SimulationPanelCategory`. If you're looking to add extensive functionality, you can even create additional panel categories in the menu to divide up your panel apps. This tutorial will cover the simpler case of creating a panel app for the already present hidden category.

1. Follow the steps documented in the Adding Custom Panel Apps section for creating custom panel apps. Once you've created the foundation of your panel app, move on to learn how to tweak it so it customizes the Simulation Menu.

   You can easily generate a Simulation Panel App by using Blade CLI's Simulation Panel Entry template. You can also refer to the Simulation Panel App sample for a working example.

2. Since this tutorial assumes you're providing more functionality to the existing simulation category, set the simulation category in the `panel.category.key` of the `@Component` annotation:

   ```
   "panel.category.key=" + SimulationPanelCategory.SIMULATION
   ```

   In order to use this constant, you need to add a dependency on com.liferay.product.navigation.simulation. Be sure to also specify the order you'd like to display your new panel app, which was explained in the Adding Custom Panel Apps section.

3. This tutorial assumes you're using JSPs for creating a complex UI. Therefore, you should extend the BaseJSPPanelApp abstract class. This class implements the PanelApp interface and also provides additional methods necessary for specifying JSPs to render your panel app's UI. Remember that you can also implement your own `include()` method to use any frontend technology you want, if you'd like to use a technology other than JSP (e.g., FreeMarker).

4. Define your simulation view. For instance, in `DevicePreviewPanelApp`, the `getJspPath` method points to the `simulation-device.jsp` file in the `resources/META-INF/resources` folder, where the device simulation interface is defined. Optionally, you can also add your own language keys, CSS, or JS resources in your simulation module.

   The right servlet context is also provided implementing this method:

   ```
   @Override
   @Reference(
       target = "(osgi.web.symbolicname=com.liferay.product.navigation.simulation.device)",
       unbind = "-"
   )
   public void setServletContext(ServletContext servletContext) {
       super.setServletContext(servletContext);
   }
   ```

   As explained in Customizing The Product Menu, a panel app should be associated with a portlet. This makes the panel app visible only when the user has permission to view the portlet. This panel app is associated to the Simulation Device portlet using these methods:

   ```
   @Override
   public String getPortletId() {
       return ProductNavigationSimulationPortletKeys.
           PRODUCT_NAVIGATION_SIMULATION;
   }

   @Override
   @Reference(
       target = "(javax.portlet.name=" + ProductNavigationSimulationPortletKeys.PRODUCT_NAVIGATION_SIMULATION + ")",
       unbind = "-"
   )
   public void setPortlet(Portlet portlet) {
       super.setPortlet(portlet);
   }
   ```

   Audience Targeting also provides a good example of how to extend the Simulation Menu. When the Audience Targeting app is deployed, the Simulation Menu is extended to offer more functionality, in particular, for Audience Targeting User Segments and Campaigns. You can simulate particular scenarios for campaigns and users directly from the Simulation Menu. Its panel app class is very similar to `DevicePreviewPanelApp`, except it points to a different portlet and JSP.

5. You can combine your simulation options with the device simulation options by interacting with the device preview iFrame. To retrieve the device preview frame in an `aui:script` block of your custom simulation view's JavaScript, you can use the following:

   ```
   var iframe = A.one('#simulationDeviceIframe');
   ```

   Then you can modify the device preview frame URL like this:

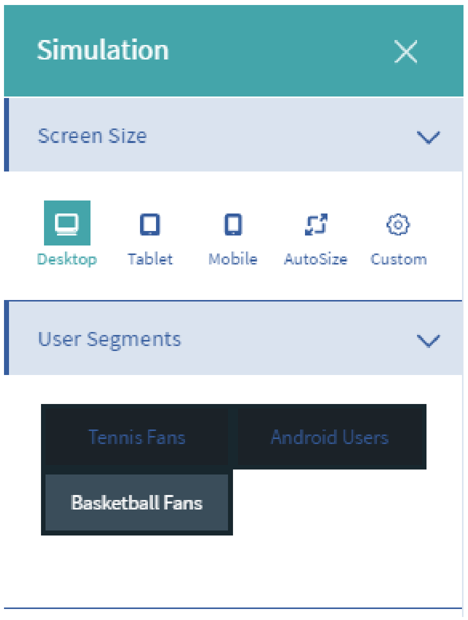


Figure 111.6: The Audience Targeting app extends the Simulation Menu to help simulate different users and campaign views.

```
    iframe.setAttribute('src', newUrlWithCustomParameters);
```

Now that you know how to extend the necessary panel categories and panel apps to modify the Simulation Menu, go ahead and create a module of your own and customize the Simulation Menu so it's most helpful for your needs.

111.4 Providing the User Personal Bar

Liferay offers a touch of personability with the User Personal Bar. This navigation menu is used to display options that are unique to the current logged in user. By default, Liferay displays this menu as a simple avatar button that expands the User Settings sub-menu in the Product Menu.

Although Liferay's default usage of the User Personal Bar is bare-bones, you can add more functionality to the user bar to fit your needs. Unlike other product navigation menus (e.g., Product Menu), the User Personal Bar does not require the extension/creation of panel categories and panel apps. It uses another common Liferay framework for providing functionality: Portlet Providers. Be sure to visit the linked tutorial to learn about how the Portlet Providers framework works in Liferay.

The User Personal Bar can be seen as a placeholder in every Liferay theme. By default, Liferay provides one sample *User Personal Bar* portlet that fills that placeholder, but the portlet Liferay provides can be easily replaced by other portlets.

Note: You can add the User Personal Bar to a custom theme by adding the following snippet into your `portal_normal.ftl`:

```
<@liferay.user_personal_bar />
```

In this tutorial, you'll learn how to customize the User Personal Bar. You'll create a single Java class where you'll specify a portlet to replace the existing default portlet.

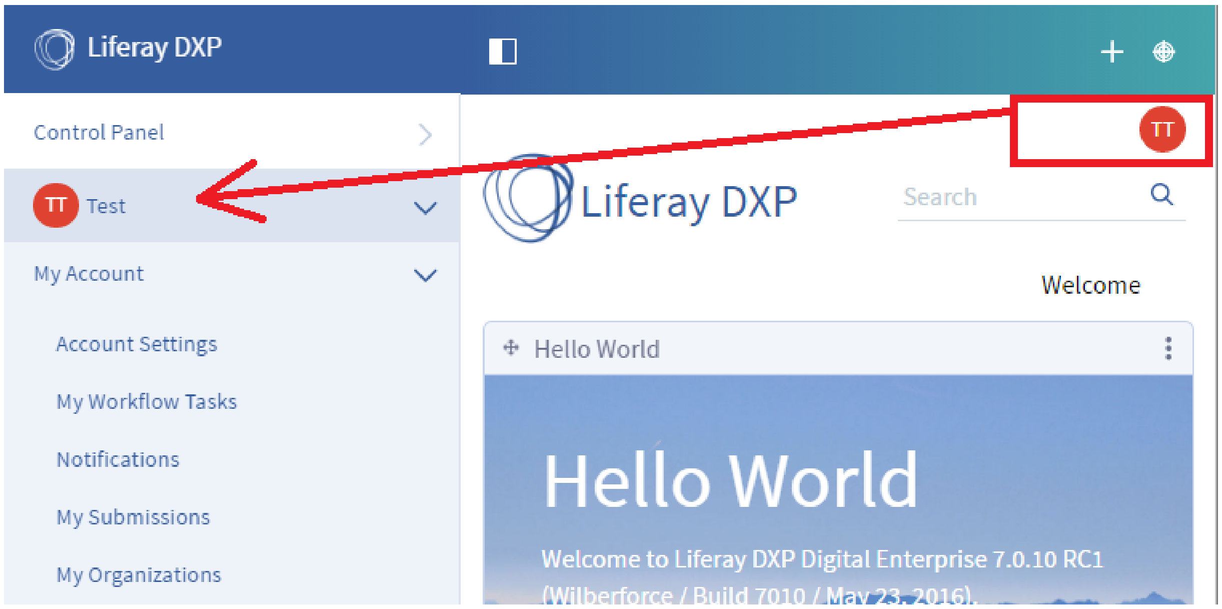


Figure 111.7: By default, the User Personal Bar contains the signed-in user's avatar, which navigates to the Product Menu when selected.

1. Create an OSGi module.

2. Create a unique package name in the module's src directory and create a new Java class in that package.

3. Directly above the class's declaration, insert the following annotation:

```
@Component(
    immediate = true,
    property = {
        "model.class.name=" + PortalUserPersonalBarApplicationType.UserPersonalBar.CLASS_NAME,
        "service.ranking:Integer=10"
    },
    service = ViewPortletProvider.class
)
```

   The `model.class.name` property must be set to the class name of the entity type you want the portlet to handle. In this case, you want your portlet to be provided based on whether or not it can be displayed in the User Personal Bar. You may recall from the Portlet Providers tutorial that you can request portlets in several different ways (e.g., *Edit*, *Browse*, etc.).

   You should also specify the service rank for your new portlet so it overrides the default one provided by Liferay DXP. Make sure to set the `service.ranking:Integer` property to a number that is ranked higher than the portlet being used by default.

   Since you're only wanting the User Personal Bar to display your portlet, you'll always have the service element be `ViewPortletProvider.class`.

4. Update the class's declaration to extend the BasePortletProvider abstract class and implement `ViewPortletProvider`:

```
public class ExampleViewPortletProvider extends BasePortletProvider implements ViewPortletProvider {
```

5. Specify the portlet you'd like to provide in the User Personal Bar by declaring the following method in your class:

   ```
   @Override
   public String getPortletName() {
       return PORTLET_NAME;
   }
   ```

   You should replace the `PORTLET_NAME` text with the portlet you want to provide Liferay when it requests one to be viewed in the User Personal Bar. For example, Liferay declares `com_liferay_product_navigation_user_personal_` for its default User Personal Bar portlet.

You've successfully provided a portlet to be displayed in the User Personal Bar. If you'd like to inspect the entire module used for Liferay's default User Personal Bar, see product-navigation-user-personal-bar-web. Besides the `*ViewPortletProvider` class, this module contains two classes defining constants and a portlet class defining the default portlet to provide. Although these additional classes are not required, your module should have access to the portlet you want to provide.

Chapter 112

Testing

Assuring top quality is paramount in producing awesome software. Test driven development plays a key role in this process. Liferay's tooling and integration with standard test frameworks support test driven development and help you reach quality milestones. Here are the ways Liferay facilitates testing:

- Unit testing: Using JUnit to unit test Liferay DXP modules in Gradle and Maven build environments and in IDEs that have JUnit plugins is seamless.
- Integration testing: The Arquillian Extension for Liferay lets you spin up a Liferay DXP instance, deploy modules whose components provide and consume services, and exercise their APIs. Liferay's `@Inject` annotation allows you to inject service instances into tests.
- Functional testing: Selenium and the Arquillian Extension for Liferay support functional UI testing.
- Code Coverage: JaCoCo analyzes and reports test code coverage.
- Slim Runtime: Liferay Slim Runtime facilitates testing modules (including Service Builder modules) in a fast, lightweight environment.

Unit testing is the first step in test driven development.

112.1 Unit Testing with JUnit

Test driven development is a best practice for any developer. Unit tests verify and validate functionality of classes and methods in isolation by "mocking" external dependencies. One of the most widely-used tools for test driven development on the Java platform is JUnit. You can use the JUnit framework to write unit tests for Liferay DXP applications.

JUnit integrates with build environments such as Maven and Gradle. JUnit plugins are available in IDEs such as Eclipse, IntelliJ, and NetBeans. And of course, Liferay Workspace supports running JUnit tests.

This tutorial covers the following topics:

- Writing good tests
- JUnit annotations
- Creating JUnit test classes
- Running JUnit tests

You'll start by learning best practices for writing unit tests.

Writing Good Tests

To write good tests, developers must understand assertions and follow best practices.

Assertion: an executable specification of the expected behavior of the software under test (SUT) given a scenario. The tests define the behavior in the scenario using several methods: a test setup method, a class setup method, and a test method. It's *executable* because it programmatically checks behavior and tracks requirements.

Best Practices for Unit Tests:

Rule	Description
A test should have only one reason to fail.	Resolving failures from a single root cause is easiest.
A test should check just one thing.	Tests that verify or validate one thing are easier to understand and maintain. Focusing on multiple things can lead to multiple failure points, thus breaking the *one reason to fail* rule.
Avoid conditional logic in tests.	Conditional test logic that uses loops or if/else clauses increases the probability of test bugs.
A test that asserts nothing or cannot fail is worthless.	Tests that can't fail create a false sense of security.Here's example test code that can't fail:`File f = new File ("foo");Assert.assertTrue(f ≠ null);`
A test that inaccurately advertises what it asserts is untrustworthy	A test's name should accurately express what it tests. A name that's inaccurate or that promises more than what the test does creates confusion. `@Testpublic void testAddUser() { // do something not related to user creation }`

Next, you'll learn JUnit's annotations for test methods.

Understanding JUnit Annotations

The following table describes the JUnit method annotations.

Method signature	Description
`@BeforeClasspublic static void method()`	The method is invoked once, before the class's entire suite of tests is executed. It should prepare the general test environment.
`@Beforepublic void method()`	The method is invoked before each test. It should prepare the environment for each test.
`@Testpublic void method()`	Marks the method as a test.
`@Test (expected = SomeException.class)public void method()`	The test fails if the method doesn't throw the exception.
`@Afterpublic void method()`	The method is invoked after each test. It should clean up the environment.
`@AfterClasspublic static void method()`	The method is invoked once, after the class's entire suite of tests is executed. It should cleanup the general test environment.
`@Ignore or @Ignore("Why disabled")public void method()`	The method is skipped. Adding the `@Ignore` annotation is an easy way to skip a test. The message (optional) can explain why the test is being ignored.

JUnit follows the algorithm below to execute the test class's methods.

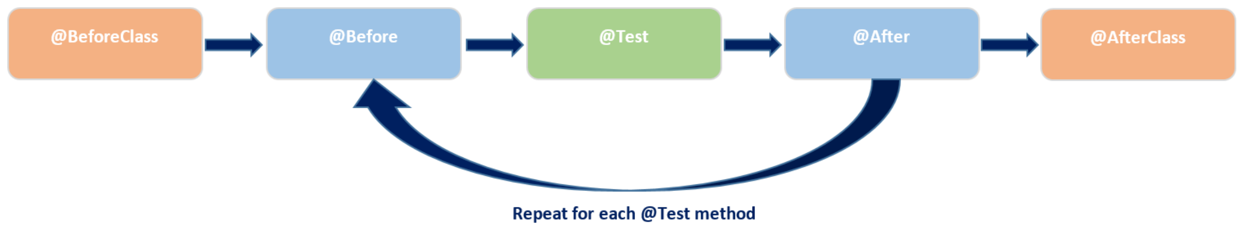


Figure 112.1: JUnit executes the annotated methods following this algorithm.

Let's create a JUnit test class.

Creating a JUnit Test Class

Here you'll create a JUnit test class and fill it with methods that both set up/clean up the test environment and assert the software's expected behavior.

To help illustrate creating unit tests, here's an example class to test:

```
public class MySampleNameClass {

   public MySampleNameClass(String firstName, String middleName, String lastName) {
      _firstName = firstName;
      _middleName = middleName;
      _lastName = lastName;
   }

   public int fullNameLength() {
      return _firstName.length() + _middleName.length() + _lastName.length();
   }

   public String getMiddleInitial() {
      return _middleName.charAt(0) + ".";
   }

   @Override
   public String toString() {
      return _firstName + " " + getMiddleInitial() + " " + _lastName;
   }

   private String _firstName;
   private String _middleName;
   private String _lastName;

}
```

In the sections that follow, you'll see setup, cleanup, and test methods that relate to this example class. Create a test class:

1. Open the module of the class you're testing.
2. Add a `src/test/java/` folder to the module.
3. In that folder, create a package path (ending in `test`) that mirrors the package path of the class you're testing.

   For example, if the class is in package `com.sample`, add a test package `com.sample.test`.
4. In that package, create a test class that ends in `Test` (e.g., `SomeTest.java`).

Your new test class is ready for test methods.

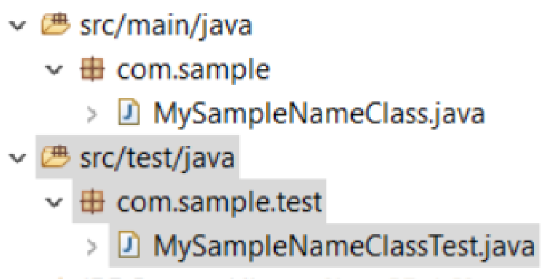


Figure 112.2: In this example module, the JUnit test class is in the same module of the class it tests. The test class resides in a source folder and package following standard test structure conventions.

Now create methods in the order of test flow execution.

@BeforeClass

Identify resources or computationally expensive tasks that must be completed prior to running all the tests. Create a method that initializes these resources and invokes these tasks. Apply the `@BeforeClass` annotation to the method.

@Before

Consider what needs to be done before running each individual test. Create a method that makes small preparations before each test case. Add the `@Before` annotation to the method.

For example, each of the tests for the class `MySampleNameClass` operate on a populated `MySampleNameClass` object. Implementing a method that instantiates such an object beforehand is appropriate. Adding the `@Before` annotation to the method ensures it's executed before each individual test.

Here's what the `MySampleNameClassTest` class might look like with such a method:

```
public class MySampleNameClassTest {

   @Before
   public void setUp() {
      _mySampleNameClass = new MySampleNameClass("Brian", "Edward", "Greenwald");
   }

   private MySampleNameClass _mySampleNameClass;

}
```

Note: Since this example class is immutable, it might make more sense to instantiate the object once in the `@BeforeClass` method and forgo the `@Before` method. It's probably more typical, however, that you'll be testing methods that change an object's state; so pretend this example object must be instantiated anew before each `@Test` method.

Now that you've instantiated objects each test needs, you can add `@Test` methods to assert expected output from the object's methods.

@Test

JUnit's `Assert` utility class contains static methods for comparing actual test results with expected results. When an assertion fails, an `AssertionException` is thrown and the test fails. If a test method completes execution without throwing an exception, the test succeeds.

For tests that contain a large amount of logic it's typically a best practice to use multiple assertions within the test to better identify the earliest point of failure. But since the example class is fairly simple, it's better to create a test for each of its methods:

```
@Test
public void testFullNameLength() {
   int length = _mySampleNameClass.fullNameLength();

   Assert.assertEquals(20, length);
}

@Test
public void testGetMiddleInitial() {
   String middleInitial = _mySampleNameClass.getMiddleInitial();

   Assert.assertEquals("E.", middleInitial);
}

@Test
```

```
public void testToString() {
   String fullName = _mySampleNameClass.toString();

   Assert.assertEquals("Brian E. Greenwald", fullName);
}
```

Since you know the test input (e.g., the parameters passed into the `MySampleNameClass` constructor), you can easily determine expected output. You can compare the computed value each method invocation returns with the expected value.

To test that a method throws a particular exception when given certain invalid inputs, add an `expected` attribute to the `@Test` annotation and assign the attribute the expected exception's class name. Important: refrain from making any assertions in the test method.

```
@Test(expected = MySampleException.class)
public void testToString() {
   String fullName = _mySampleNamenClass.toString();
}
```

Warning: Each test method should be independent. Since JUnit doesn't guarantee test order, you can't rely on a test being run before or after other tests.

@After

On finishing each individual test, you should clean up anything that was created or modified. Implement the cleanup instructions in a method and add the `@After` annotation to it.

@AfterClass

On finishing the entire set of tests, you should clean up any remaining test environment resources. Implement the final cleanup instructions in a method and add the `@AfterClass` annotation to it.

It's time to compile and run your JUnit tests.

Running JUnit Tests

Unit testing involves these things:

- Resolving test dependencies
- Executing the tests
- Analyzing test results

Add JUnit as a dependency. Here's a dependency on JUnit in Gradle:

```
testCompile group: "junit", name: "junit", version: "4.12"
```

If your tests require accessing classes outside the module, declare them as `testCompile` dependencies in the module's `build.gradle` file. Here's an example `testCompile` dependency.

```
testCompile group: "com.sample", name: "com.sample.external", version: "1.0.0"
```

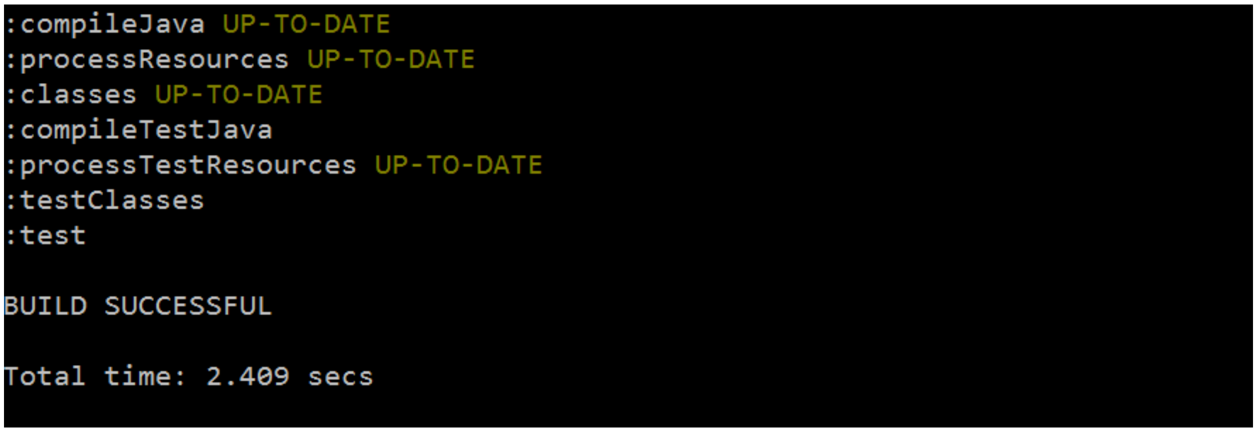
Because unit tests run independent of any running Liferay DXP instance, you can use external modules in tests but can't access their services.

Gradle and Maven commands execute module unit tests:

Gradle: `./gradlew test`

Maven: `mvn test`

The module's classes and tests compile and its tests run. The following figure shows command output for a successful test execution.



```
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava
:processTestResources UP-TO-DATE
:testClasses
:test

BUILD SUCCESSFUL

Total time: 2.409 secs
```

Figure 112.3: Command output of successful test execution looks like this.

Reports are generated to module subfolders based on the build environment and/or reporting mechanism.

Gradle:

- HTML report `build/reports/tests/index.html`
- XML report file in `build/test-results/`

Maven:

- SureFire plugin generates XML and text report files in `targets/surefire-reports/`

Gradle's HTML report, for example, shows overall test metrics and organizes test results by package. Clicking on a package name lists test class results. Clicking on a test class name lists test method results.

While it's certainly helpful to see successful test results, it's even more helpful to see results of failing tests.

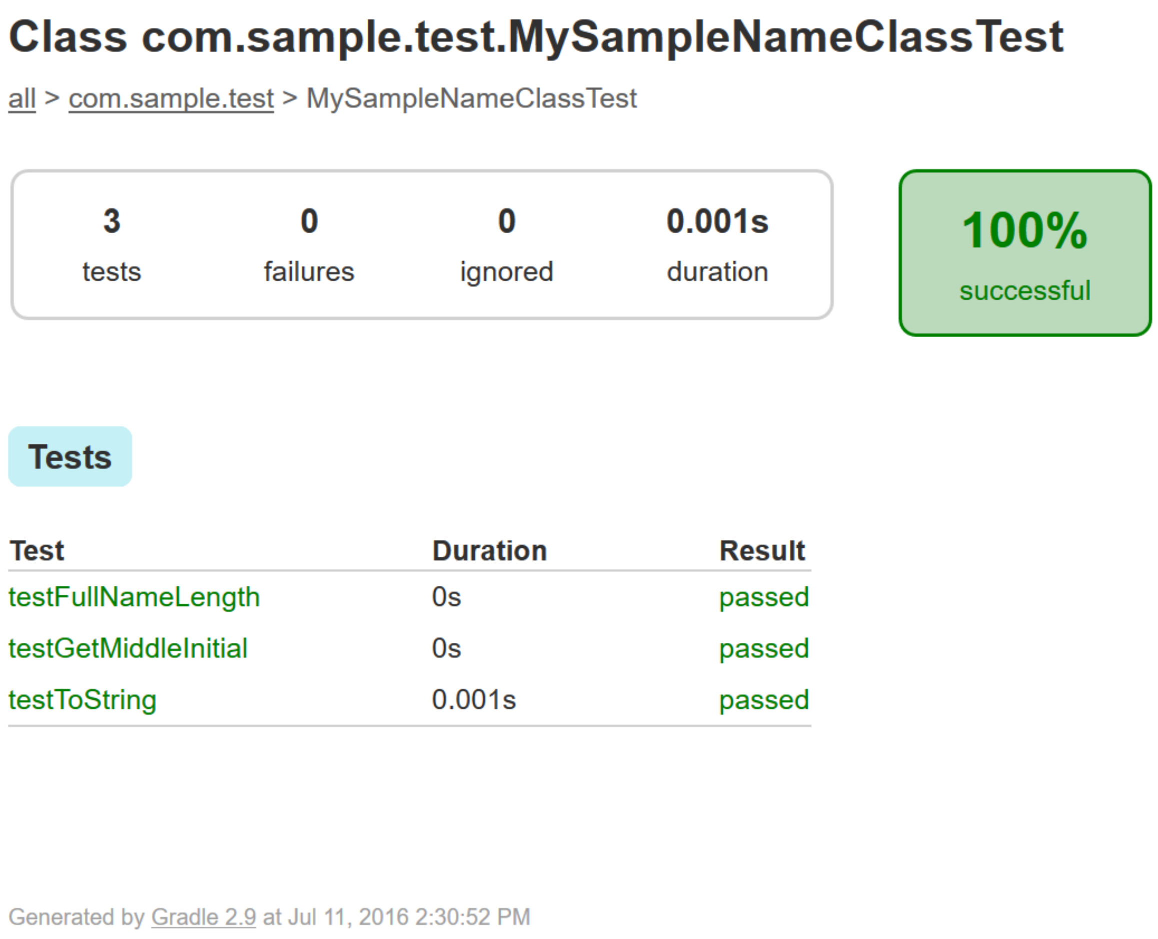
As an experiment, change the expected values of a test's assertions to force the test to fail and rerun the tests to yield the respective failures.

The command output shows the class and method name of each failed test, the JUnit assertion type, and the assertion's line number.

The generated HTML report shows each failing test's stack trace.

Stack traces show exactly why the test failed. They're essential for determining whether the failure is the result of faulty business logic or an incorrect expected value in the assertion. Using this information the developer can resolve the issue.

Congratulations on creating and executing unit tests with JUnit!


Class com.sample.test.MySampleNameClassTest

all > com.sample.test > MySampleNameClassTest

3	0	0	0.001s	100%
tests	failures	ignored	duration	successful

Tests

Test	Duration	Result
testFullNameLength	0s	passed
testGetMiddleInitial	0s	passed
testToString	0.001s	passed

Generated by Gradle 2.9 at Jul 11, 2016 2:30:52 PM

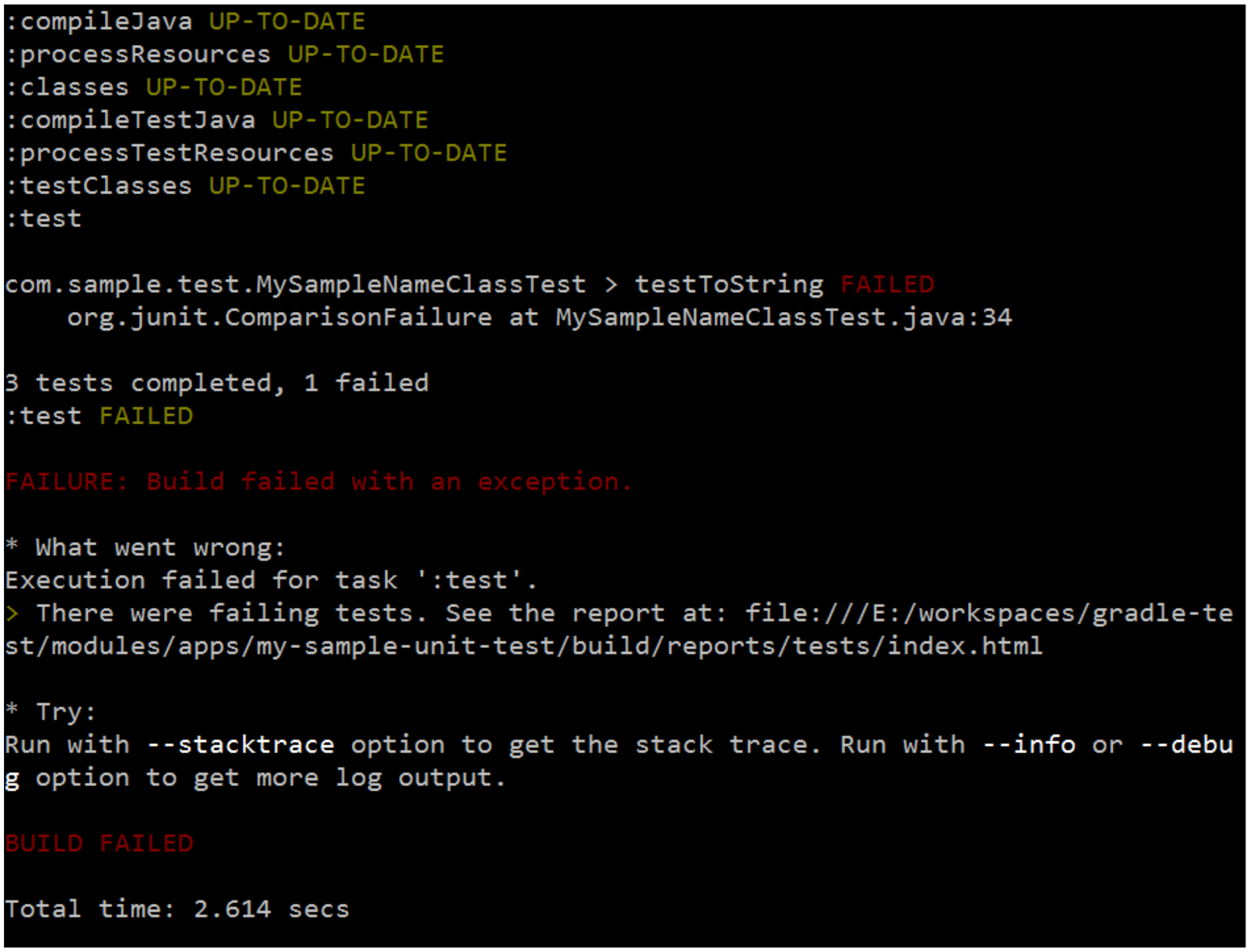
Figure 112.4: In Gradle environments, JUnit produces an HTML file named `index.html` that reports test result details.

Related Topics

Integration Testing with the Arquillian Extension

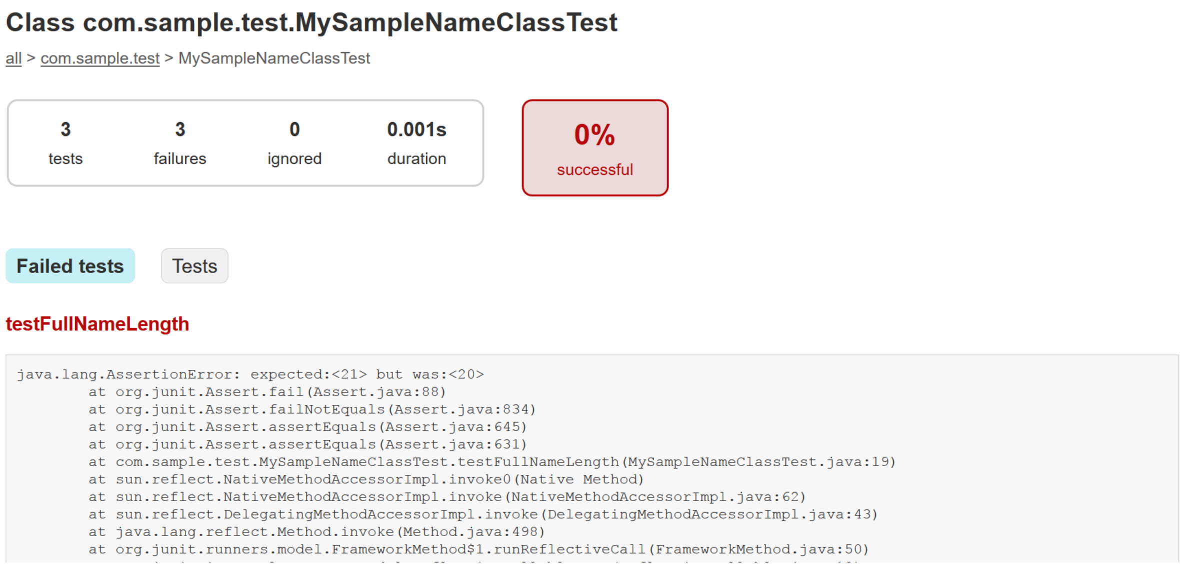
Liferay Workspace

Liferay @ide@



```
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test

com.sample.test.MySampleNameClassTest > testToString FAILED
    org.junit.ComparisonFailure at MySampleNameClassTest.java:34

3 tests completed, 1 failed
:test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at: file:///E:/workspaces/gradle-te
st/modules/apps/my-sample-unit-test/build/reports/tests/index.html

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debu
g option to get more log output.

BUILD FAILED

Total time: 2.614 secs
```

Figure 112.5: Command output of failing tests looks like this.



Class com.sample.test.MySampleNameClassTest

all > com.sample.test > MySampleNameClassTest

3	3	0	0.001s
tests	failures	ignored	duration

0%
successful

testFullNameLength

```
java.lang.AssertionError: expected:<21> but was:<20>
	at org.junit.Assert.fail(Assert.java:88)
	at org.junit.Assert.failNotEquals(Assert.java:834)
	at org.junit.Assert.assertEquals(Assert.java:645)
	at org.junit.Assert.assertEquals(Assert.java:631)
	at com.sample.test.MySampleNameClassTest.testFullNameLength(MySampleNameClassTest.java:19)
	at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
	at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
	at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
	at java.lang.reflect.Method.invoke(Method.java:498)
	at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
```

Figure 112.6: Here's a failed test's stack trace.

Chapter 113

Arquillian Extension for Liferay Example

Arquillian is an extensible Java testing platform that's designed to make integration testing easy. Arquillian manages the lifecycle of setting up, starting, or connecting to a container (e.g., Tomcat), packaging your test cases and any dependent classes or resources, deploying them to the container, running the tests in the container, and capturing and reporting the results. The Arquillian Extension for Liferay is a set of tools designed to help developers test their Liferay plugins.

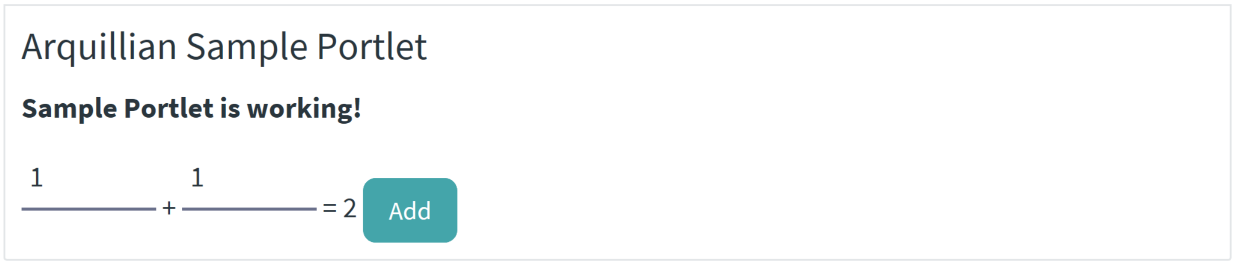
The Arquillian Blade Example project demonstrates performing integration and functional tests using the Arquillian Liferay Extension. Additionally it measures code coverage using JaCoCo. In this tutorial, you'll learn how the Arquillian Liferay Extension and JaCoCo work. You can download the Arquillian Blade Example project here or access its latest code on GitHub.

Here are the tutorial sections:

- Arquillian Sample Portlet
- Arquillian Integration Test Example
- Arquillian Functional Test Example
- JaCoCo Code Coverage Example
- Running the Arquillian Example

113.1 Arquillian Example Sample Portlet

The sample portlet calculates the sum of two numbers.


Arquillian Sample Portlet

Sample Portlet is working!

1 + 1 = 2

Figure 113.1: The Arquillian Sample Portlet calculates the sum of two numbers.

The portlet project comprises a portlet class, service classes, and JSPs. It follows the standard OSGi module folder structure with Java files in src/main/java/, resource files in src/main/resources/META-INF/resources, and build files in the project root.

Here are the primary files:

- SampleService.java: Provides an interface that defines method public long add(final int addend1, final int addend2) for returning the sum of two numbers.
- SampleServiceImpl.java: Uses OSGi Declarative Services to implement the SampleService interface.
- SamplePortlet.java: Extends Liferay MVCPortlet and processes portlet action commands and renders the result of executing the add service.
- bnd.bnd: Specifies the module's name, symbolic name, and version.
- init.jsp: Imports classes and tag libraries for the view layer.
- view.jsp: Provides a form for calculating the sum of two numbers.

You'll examine the tests next.

113.2 Arquillian Integration Test Example

Integration tests exercise module interaction. The following integration test validates the sample portlet using its API. Although the example's BasicPortletIntegrationTest class demonstrates invoking the sample module's SampleService.add method only, an Arquillian integration test could just as easily invoke many methods to test behavior across many modules. The test classes are in the src/testIntegration/java folder and test resources are in the src/testIntegration/resources folder.

Here's the BasicPortletIntegrationTest class:

```
package com.liferay.arquillian.test;

import com.google.common.io.Files;

import com.liferay.arquillian.containter.remote.enricher.Inject;
import com.liferay.arquillian.sample.service.SampleService;
import com.liferay.portal.kernel.exception.PortalException;

import java.io.File;
import java.io.IOException;

import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

@RunWith(Arquillian.class)
public class BasicPortletIntegrationTest {

    @Deployment
    public static JavaArchive create() throws Exception {
        final File tempDir = Files.createTempDir();

        String gradlew = "./gradlew";
        String osName = System.getProperty("os.name", "");
        if (osName.toLowerCase().contains("windows")) {
            gradlew = "./gradlew.bat";
        }

        final ProcessBuilder processBuilder = new ProcessBuilder(
            gradlew, "jar", "-Pdir=" + tempDir.getAbsolutePath());

        final Process process = processBuilder.start();

        process.waitFor();

        final File jarFile = new File(
            tempDir.getAbsolutePath() +
                "/com.liferay.arquillian.sample-1.0.0.jar");

        return ShrinkWrap.createFromZipFile(JavaArchive.class, jarFile);
    }

    @Test
    public void testAdd() throws IOException, PortalException {
        final long result = _sampleService.add(1, 3);

        Assert.assertEquals(4, result);
    }

    @Inject
    private SampleService _sampleService;

}
```

JUnit annotation `@RunWith(Arquillian.class)` marks the class for Arquillian to execute.

The `create` method packages the test class and resources in a Java archive (JAR). Invoking the project's `jar` Gradle task creates the test JAR Arquillian executes.

```
@Deployment
public static JavaArchive create() throws Exception {
    final File tempDir = Files.createTempDir();

    final ProcessBuilder processBuilder = new ProcessBuilder(
        "./gradlew", "jar", "-Pdir=" + tempDir.getAbsolutePath());

    final Process process = processBuilder.start();

    process.waitFor();

    final File jarFile = new File(
        tempDir.getAbsolutePath() +
            "/com.liferay.arquillian.sample-1.0.0.jar");

    return ShrinkWrap.createFromZipFile(JavaArchive.class, jarFile);
}
```

JUnit annotation `@Test` designates the `testAdd` method as a test. The method invokes the `SampleService` object's `add` method and asserts its result.

```
@Test
public void testAdd() throws IOException, PortalException {
    final long result = _sampleService.add(1, 3);

    Assert.assertEquals(4, result);
}
```

The Liferay Arquillian Extension injects the `_sampleService` field with a `SampleService` implementation (i.e., a `SampleServiceImpl` instance).

```
@Inject
private SampleService _sampleService;
```

The integration test has some dependencies, of course.

Dependencies

The project's `build.gradle` file specifies this test's dependencies on Liferay's Arquillian container, JUnit, and an Arquillian JUnit test container:

```
testIntegrationCompile group: "com.liferay.arquillian", name: "com.liferay.arquillian.arquillian-container-liferay", version: "1.0.6"
testIntegrationCompile group: "junit", name: "junit", version: "4.12"
testIntegrationCompile group: "org.jboss.arquillian.junit", name: "arquillian-junit-container", version: "1.1.11.Final"
```

Arquillian tests are configurable too.

Arquillian Configuration

Arquillian configuration file `src/testIntegration/resources/arquillian.xml` uses property `deploymentExportPath` (optional) to write a test archive (e.g., JAR file) to a folder before deploying the tests. You can inspect all the test files from this archive. To highlight the `deploymentExportPath` property, here's an abbreviated view of the `arquillian.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<arquillian xmlns="http://jboss.org/schema/arquillian"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://jboss.org/schema/arquillian http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

    <!-- More content here -->

    <engine>
        <property name="deploymentExportPath">build/deployments</property>
    </engine>
</arquillian>
```

The project uses Java Management Extensions (JMX) to deploy OSGi modules to Liferay DXP. Enabling JMX for the application server is next.

JMX Settings

Apache Aries JMX exposes the JMX API that Arquillian uses to install/deploy/start the modules. Since DXP Digital Enterprise 7.0 Fix Pack 16 and Liferay CE 7.0 GA4, Liferay Workspace's `startTestableTomcat` Gradle task installs the Apache Aries JMX modules automatically. In case you're using an earlier Liferay DXP version or haven't already installed the Aries modules, here's their group ID, artifact ID, and version information. You can install them using Apache Felix GoGo Shell:

```
"org.apache.aries.jmx:org.apache.aries.jmx:1.1.5"
"org.apache.aries:org.apache.aries.util:1.1.3"
```

JMX is enabled on the application server via Java runtime options. The following Apache Tomcat environment script excerpts demonstrate enabling JMX (without authentication) on port 8099.

`setenv.sh` JMX Settings

```
JMX_OPTS="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.port=8099 -
Dcom.sun.management.jmxremote.ssl=false"

CATALINA_OPTS="${CATALINA_OPTS} ${JMX_OPTS}"
```

`setenv.bat` JMX Settings

```
set "JMX_OPTS=-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.port=8099 -
Dcom.sun.management.jmxremote.ssl=false"

set "CATALINA_OPTS=%CATALINA_OPTS% %JMX_OPTS%"
```

Apache Tomcat's guide *Enabling JMX Remote* has more JMX configuration details.

You've seen how setting up the example integration test class is straightforward. Next, you'll discover how fun it is to develop functional browser-based tests using Arquillian.

113.3 Arquillian Functional Test Example

Where the integration test invokes the `SampleService`'s `add` method directly, the functional test invokes the `add` method indirectly using a web browser. The Arquillian Blade Example's functional tests interact with the portlet UI to verify content and validate behavior. The test classes are in the `src/testIntegration/java` folder and test resources are in the `src/testIntegration/resources` folder.

The example functional tests operate on the following view parameters:

- `firstParameter`: First number to add
- `secondParameter`: Second number to add
- `result`: Sum of the two numbers

The JSP file `view.jsp` and portlet class `BasicPortletFunctionalTest` use these parameters. Here's the `view.jsp` code:

```
<%@ include file="/init.jsp" %>

<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>

<%
int firstParameter = ParamUtil.getInteger(request, "firstParameter", 1);
int secondParameter = ParamUtil.getInteger(request, "secondParameter", 1);
int result = ParamUtil.getInteger(request, "result");
%>

<portlet:actionURL name="add" var="portletURL" />

<p>
<b>Sample Portlet is working!</b>
</p>

<aui:form action="<%= portletURL %>" method="post" name="fm">
<aui:input inlineField="<%= true %>" label="" name="firstParameter" size="4" type="int" value="<%= firstParameter %>" />
<span> + </span>
<aui:input inlineField="<%= true %>" label="" name="secondParameter" size="4" type="int" value="<%= secondParameter %>" />
<span> = </span>
<span class="result"><%= result %></span>

<aui:button type="submit" value="add" />
</aui:form>
```

Users enter numbers in the `firstParameter` and `secondParameter` input fields and click on the add button to show the sum to the `result` field.

Functional test class `BasicPortletFunctionalTest` uses Selenium to interact with the portlet's UI. Here's the `BasicPortletFunctionalTest` class:

```
package com.liferay.arquillian.test;

import com.google.common.io.Files;

import com.liferay.arquillian.portal.annotation.PortalURL;
import com.liferay.portal.kernel.exception.PortalException;

import java.io.File;
import java.io.IOException;

import java.net.URL;

import org.jboss.arquillian.container.test.api.Deployment;
import org.jboss.arquillian.container.test.api.RunAsClient;
import org.jboss.arquillian.drone.api.annotation.Drone;
import org.jboss.arquillian.junit.Arquillian;
import org.jboss.shrinkwrap.api.ShrinkWrap;
import org.jboss.shrinkwrap.api.spec.JavaArchive;

import org.junit.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;

import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;

@RunAsClient
@RunWith(Arquillian.class)
public class BasicPortletFunctionalTest {

    @Deployment
    public static JavaArchive create() throws Exception {
        final File tempDir = Files.createTempDir();

        String gradlew = "./gradlew";

        String osName = System.getProperty("os.name", "");
        if (osName.toLowerCase().contains("windows")) {
            gradlew = "./gradlew.bat";
        }

        final ProcessBuilder processBuilder = new ProcessBuilder(
            gradlew, "jar", "-Pdir=" + tempDir.getAbsolutePath());

        final Process process = processBuilder.start();

        process.waitFor();

        final File jarFile = new File(
            tempDir.getAbsolutePath() +
                "/com.liferay.arquillian.sample-1.0.0.jar");

        return ShrinkWrap.createFromZipFile(JavaArchive.class, jarFile);
    }

    @Test
    public void testAdd()
        throws InterruptedException, IOException, PortalException {
```

```
        _browser.get(_portlerURL.toExternalForm());

        _firstParamter.clear();

        _firstParamter.sendKeys("2");

        _secondParameter.clear();

        _secondParameter.sendKeys("3");

        _add.click();

        Thread.sleep(5000);

        Assert.assertEquals("5", _result.getText());
    }

    @Test
    public void testInstallPortlet() throws IOException, PortalException {
        _browser.get(_portlerURL.toExternalForm());

        final String bodyText = _browser.getPageSource();

        Assert.assertTrue(
            "The portlet is not well deployed",
            bodyText.contains("Sample Portlet is working!"));
    }

    @FindBy(css = "button[type=submit]")
    private WebElement _add;

    @Drone
    private WebDriver _browser;

    @FindBy(css = "input[id$='firstParameter']")
    private WebElement _firstParamter;

    @PortalURL("arquillian_sample_portlet")
    private URL _portlerURL;

    @FindBy(css = "span[class='result']")
    private WebElement _result;

    @FindBy(css = "input[id$='secondParameter']")
    private WebElement _secondParameter;

}
```

Arquillian annotation `@RunAsClient` and JUnit annotation `@RunWith` mark the class as a web client that runs on Arquillian.

Similar to the integration test class, this class's create method packages the test as a JAR file for Arquillian to execute.

Method `testInstallPortlet` verifies portlet content.

```
@Test
public void testInstallPortlet() throws IOException, PortalException {
    _browser.get(_portlerURL.toExternalForm());

    final String bodyText = _browser.getPageSource();

    Assert.assertTrue(
        "The portlet is not well deployed",
        bodyText.contains("Sample Portlet is working!"));
}
```

This test class uses the following fields:

- `_browser`: Arquillian annotation `@Drone` sets this field as a Selenium `WebDriver` (browser).

  ```
  @Drone
  private WebDriver _browser;
  ```

- `_portlerURL`: Liferay Arquillian annotation `@PortalURL` assigns the portlet's URL to this field.

  ```
  @PortalURL("arquillian_sample_portlet")
  private URL _portlerURL;
  ```

- `_firstParamter` and `_secondParameter`: JavaScript selectors and Selenium annotation `@FindBy` map these fields to the form's inputs.

  ```
  @FindBy(css = "input[id$='firstParameter']")
  private WebElement _firstParamter;

  @FindBy(css = "input[id$='secondParameter']")
  private WebElement _secondParameter;
  ```

- `_add`: The form's `submit` button.

  ```
  @FindBy(css = "button[type=submit]")
  private WebElement _add;
  ```

- `_result`: Sum of the two numbers.

  ```
  @FindBy(css = "span[class='result']")
  private WebElement _result;
  ```

Using the `*Parameter` fields, the `testAdd` method injects numbers `2` and `3` into the form, submits the form, and asserts `5` as the result.

```
@Test
public void testAdd()
    throws InterruptedException, IOException, PortalException {

    _browser.get(_portlerURL.toExternalForm());

    _firstParameter.clear();

    _firstParameter.sendKeys("2");

    _secondParameter.clear();

    _secondParameter.sendKeys("3");

    _add.click();

    Thread.sleep(5000);

    Assert.assertEquals("5", _result.getText());
}
```

Testing portlets via a web client is that simple!

Functional tests typically require more setup than integration tests.

Dependencies

In addition to the Liferay Arquillian container, JUnit, and an Arquillian JUnit test container artifacts that the integration test required, the `BasicPortletFunctionalTest` class requires Arquillian's Graphine extension to use the web client. Here's the dependency from the Gradle file `build.gradle`:

```
testIntegrationCompile group: "org.jboss.arquillian.graphene", name: "graphene-webdriver", version: "2.1.0.Final"
```

Note: To learn more about functional testing using Graphine, see this guide.

The test requires additional Arquillian configuration elements too.

Arquillian Configuration

In addition to the `deploymentExportPath` property introduced with the integration test, this functional test specifies the browser type using `phantomjs` and the portal's URL. Here's the project's `arquillian.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<arquillian xmlns="http://jboss.org/schema/arquillian"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://jboss.org/schema/arquillian http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

    <extension qualifier="webdriver">
        <property name="browser">phantomjs</property>
    </extension>

    <extension qualifier="graphene">
        <property name="url">http://localhost:8080</property>
    </extension>

    <engine>
        <property name="deploymentExportPath">build/deployments</property>
    </engine>
</arquillian>
```

Note: The Arquillian Liferay Extension provides these options for injecting the URL of the container (e.g., Apache Tomcat):

1. In a test class deployment method or field, use the annotation `@ArquillianResource` to designate the URL.
2. Configure Arquillian using the graphene URL property (via `arquillian.xml`, `arquillian.properties`, or System Properties).

You should use Portal properties to prevent Liferay DXP from launching a browser and the Setup Wizard.

Portal Properties

The Arquillian Blade Example specifies the following Portal properties in file `src/testIntegration/resources/portal-ext.properties`:

```
browser.launcher.url=
setup.wizard.enabled=false
```

Browsers other than the ones the functional tests launch can interfere with tests. Setting `browser.launcher.url` to an empty value prevents Liferay DXP from launching a browser on its own. You don't need Liferay DXP's setup wizard either. Setting `setup.wizard.enabled=false` bypasses launching the Setup Wizard.

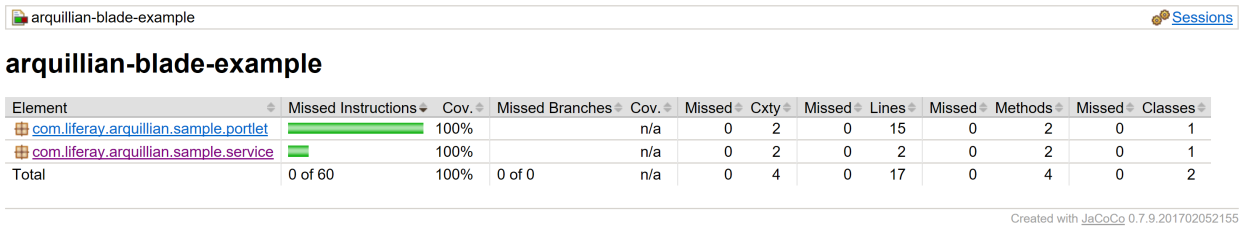
The example project's Gradle task `copyPortalExt`, copies the Portal properties file into the Liferay DXP installation.

As you develop tests, you might want to track the parts of the product your tests cover. The example project uses JaCoCo to measure code coverage. JaCoCo is explained next.

If you'd rather launch the Arquillian Blade Example before investigating JaCoCo, skip to Running the Arquillian Example. Otherwise, jump into JaCoCo!

113.4 JaCoCo Code Coverage Example

JaCoCo measures Java code coverage. The Arquillian Blade Example uses JaCoCo to report parts and percentages of the product code the tests execute.



arquillian-blade-example Sessions

arquillian-blade-example

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.liferay.arquillian.sample.portlet	🟩	100%		n/a	0	2	0	15	0	2	0	1
com.liferay.arquillian.sample.service	🟩	100%		n/a	0	2	0	2	0	2	0	1
Total	0 of 60	100%	0 of 0	n/a	0	4	0	17	0	4	0	2

Created with JaCoCo 0.7.9.201702052155

Figure 113.2: JaCoCo reports lines of code tests execute in methods and classes.

Enabling JaCoCo

JaCoCo requires attaching an agent to the JVM. To attach the JaCoCo agent, append the following JaCoCo options to the Tomcat environment script's `CATALINA_OPTS` variable.

`setenv.sh` JaCoCo Settings

```
JACOCO_OPTS="-javaagent:PATH_TO_JACOCO_AGENT_JAR/jacocoagent.jar=destfile=JACOCO_EXEC_FILE,output=file,append=true,jmx=true"

CATALINA_OPTS="${CATALINA_OPTS} ${JACOCO_OPTS}"
```

`setenv.bat` JaCoCo Settings

```
set "JACOCO_OPTS=-javaagent:PATH_TO_JACOCO_AGENT_JAR/jacocoagent.jar=destfile=JACOCO_EXEC_FILE,output=file,append=true,jmx=true"

set "CATALINA_OPTS=%CATALINA_OPTS% %JACOCO_OPTS%"
```

Replace `PATH_TO_JACOCO_AGENT_JAR` with the path to the `jacocoagent.jar` file and `JACOCO_EXEC_FILE` with the path to the JaCoCo result dump file.

JaCoCo Build Instructions

The Gradle build file `build.gradle` specifies several JaCoCo-related instructions:

1. Apply the plugin to the build:

```
apply plugin: 'jacoco'

jacoco {
    toolVersion = '0.7.9'
}
```

2. Copy the JaCoCo agent into the project build.

```
task copyJacocoAgent(type: Copy) {
println configurations.jacocoAgent

    configurations.jacocoAgent.asFileTree.each {
        from(zipTree(it))
    }

    into "${rootDir}/build/jacoco"
}
```

3. Dump the code coverage data:

```
task dumpJacoco {
    doLast {
        def serverUrl = 'service:jmx:rmi:///jndi/rmi://localhost:8099/jmxrmi'
        String beanName = "org.jacoco:type=Runtime"
        def server = JmxFactory.connect(new JmxUrl(serverUrl)).MBeanServerConnection
        def gmxb = new GroovyMBean(server, beanName)

        println "Connected to:\n$gmxb\n"
        println "Executing dump()"
        gmxb.dump(true)
    }
}
```

4. Generate JaCoCo reports:

```
jacocoTestReport {
    dependsOn dumpJacoco
    group = "Reporting"
    reports {
        xml.enabled true
        csv.enabled false
        html.destination "${buildDir}/reports/coverage"
    }
    executionData = files("${rootDir}/build/jacoco/testIntegration.exec")
}
```

JaCoCo code coverage reporting runs as part of the project's `testIntegration` Gradle task. You'll run the tests and JaCoCo next.

113.5 Running the Arquillian Example

You're ready to run the Arquillian Blade Example tests. Open a terminal to the project root and execute the following command:

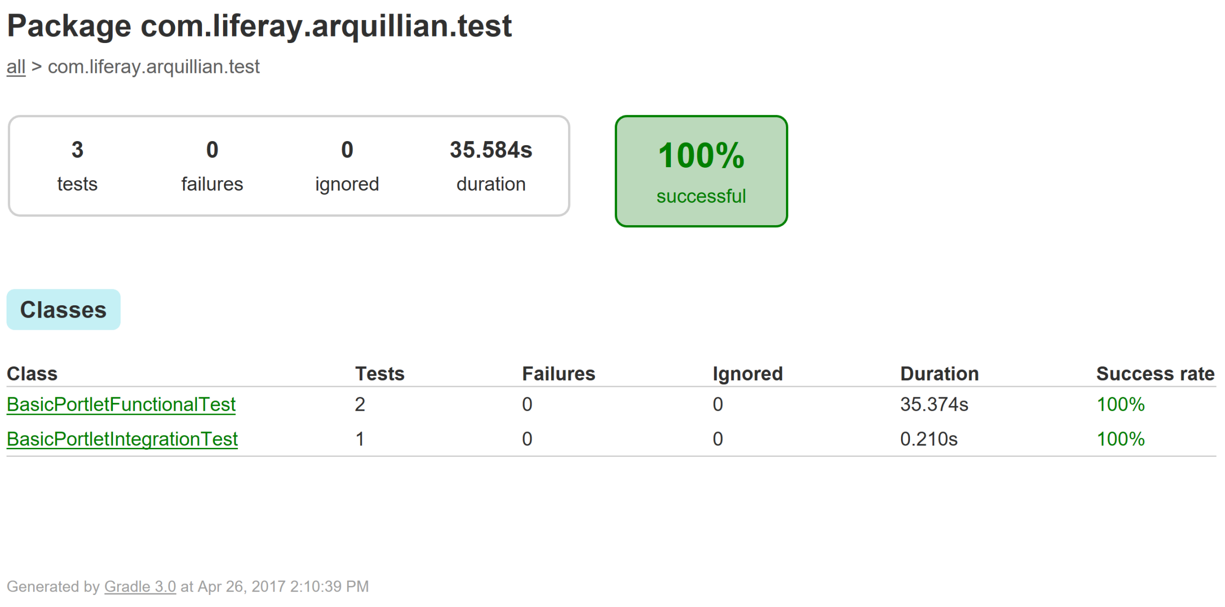
```
gradlew testIntegration
```

The command does these things:

1. Downloads and installs 7.0 bundled with Apache Tomcat
2. Starts a 7.0 server
3. Runs the tests, including the functional browser-based tests
4. Shuts down the server
5. Reports test and code coverage results

For `testIntegration` task details, examine the `build.gradle` file in the project root. The command can take several minutes to execute because of all it does. Test results are found in these locations:

- *Tests*: `build\reports\tests\testIntegration\index.html`
- *Code Coverage*: `build\reports\coverage\index.html`



Package com.liferay.arquillian.test

all > com.liferay.arquillian.test

3	0	0	35.584s
tests	failures	ignored	duration

100% successful

Classes

Class	Tests	Failures	Ignored	Duration	Success rate
BasicPortletFunctionalTest	2	0	0	35.374s	100%
BasicPortletIntegrationTest	1	0	0	0.210s	100%

Generated by Gradle 3.0 at Apr 26, 2017 2:10:39 PM

Figure 113.3: Open the test reports to analyze the results.

Note: before rerunning the tests, you must delete the `build/reports/` and `build/test-results/` folders.

Now that you've examined Arquillian functional and integration tests and JaCoCo code coverage capabilities, you can create similar tests and improve test code coverage in your projects.

Related Topics
Liferay's Slim Runtime
Unit Testing with JUnit

113.6 Liferay Slim Runtime

The Liferay Slim Runtime provides the bare necessities for running Service Builder modules. It's useful for testing applications quickly in a Liferay runtime environment free of Liferay add-ons.

The Liferay Slim Runtime provides

- Caching infrastructure
- Database infrastructure
- HTTP support
- JAX-RS support
- Limited set of Liferay utility classes
- OSGi framework for running modules
- Service Builder runtime for Service Builder modules
- Spring infrastructure
- Transaction infrastructure

It does **not** provide

- Authentication/Authorization layers
- Layout templates
- Permissions
- Portlet support (no portlet container)
- Sites
- Themes
- etc.

Building and launching a Liferay Slim Runtime is much quicker than a typical Liferay DXP bundle. Because of decreased build and startup times, the Slim Runtime provides a great environment for testing. You'll learn how to build one next.

Build

To build the Slim Runtime, you must have the liferay-portal Github repository forked and cloned to your local machine. Navigate to the repository's root folder and execute the following Ant command:

```
ant all -Dbuild.profile=slim
```

It's built in the server directory specified by the `app.server.properties` file's `app.server.parent.dir` property. Note that the Slim Runtime only supports Apache Tomcat 8+. This limitation simplifies packaging and configuration.

Launch

To launch the Slim Runtime, run the Tomcat start scripts found in the runtime's `<tomcat>/bin` directory:

```
./startup.[sh|bat]
```

Deploying Modules

You can deploy modules from any of the default directories the portal.properties file defines (see properties below) or from a custom auto-deploy directory you add to the `module.framework.auto.deploy.dirs` property.

```
module.framework.base.dir=${liferay.home}/osgi

module.framework.configs.dir=${module.framework.base.dir}/configs
module.framework.marketplace.dir=${module.framework.base.dir}/marketplace
module.framework.modules.dir=${module.framework.base.dir}/modules
module.framework.war.dir=${module.framework.base.dir}/war

module.framework.auto.deploy.dirs=\
    ${module.framework.configs.dir},\
    ${module.framework.marketplace.dir},\
    ${module.framework.modules.dir},\
    ${module.framework.war.dir}
```

By default, a pristine Slim Runtime has no UI or apps. Requests to it result in 404 errors. The modules you add provide all the functionality.

Adding Functionality

A web endpoint is the simplest type of function.

The following snippet demonstrates a simple servlet that responds to all requests to `http://localhost:8080[/*]`:

```
package web.sample;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.http.whiteboard.HttpWhiteboardConstants;

@Component(
    immediate = true,
    property = {
        HttpWhiteboardConstants.HTTP_WHITEBOARD_SERVLET_PATTERN + "=/*"
    },
    service = Servlet.class
)
public class SampleServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    protected void service(
            HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter writer = response.getWriter();

        writer.println("<h2>Hello You!</h2>");
    }

}
```

The Database

The Slim Runtime creates the database schema automatically the first time it runs.

```
MariaDB [lportal]> show tables;
+------------------+
```

```
 Tables_in_lportal|
```

---+——————+

```
 ClassName_       |
 Configuration_   |
 Counter          |
 Release_         |
 ServiceComponent |
```

---+——————+ 5 rows in set (0.00 sec)

Only the following core services are available:

- `ClassNameLocalService`
- `CounterLocalService`
- `ReleaseLocalService`
- `ServiceComponentLocalService`

The Slim Runtime provides no other services! To test your services, therefore, you must deploy modules that provide the capabilities they depend on.

Service Builder

The Service Builder runtime bootstraps all deployed Service Builder services (API and service modules).

For example, deploying the `com.liferay.contacts.api` and `com.liferay.contacts.service` modules adds the `Contacts_Entry` table to the database:

```
MariaDB [lportal]> show tables;
+------------------+
```

```
 Tables_in_lportal|
```

---+——————+

```
 ClassName_       |
 Configuration_   |
 Contacts_Entry   |
 Counter          |
 Release_         |
 ServiceComponent |
```

---+——————+ 6 rows in set (0.00 sec)

A Basic Service Builder Web App

The servlet in the following snippet implements a simple web app that uses the contacts service.

```
package web.sample;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;

import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;
import org.osgi.service.http.whiteboard.HttpWhiteboardConstants;

import com.liferay.contacts.model.Entry;
import com.liferay.contacts.service.EntryLocalService;
import com.liferay.counter.kernel.service.CounterLocalService;
import com.liferay.portal.kernel.dao.orm.DynamicQuery;
import com.liferay.portal.kernel.dao.orm.QueryUtil;
import com.liferay.portal.kernel.dao.orm.RestrictionsFactoryUtil;
import com.liferay.portal.kernel.util.ParamUtil;
import com.liferay.portal.kernel.util.Validator;

@Component(
    immediate = true,
    property = {
        HttpWhiteboardConstants.HTTP_WHITEBOARD_SERVLET_PATTERN + "=/*"
    },
    service = Servlet.class
)
public class SampleServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    protected void service(
            HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");

        PrintWriter writer = response.getWriter();

        String fullNameParameter = ParamUtil.getString(request, "fullName");

        if (Validator.isNull(fullNameParameter)) {
            writer.println("<h2>Hello You!</h2>");
            writer.println("Do you want to sign up for this thing?<br/>");
            writer.println("<form action='/join' method='post'>");
            writer.println("<input type='text' name='fullName' placeholder='Full Name'><br>");
            writer.println("<input type='text' name='emailAddress' placeholder='Email Address'><br>");
            writer.println("<input type='submit' value='Sign Up'><br>");
            writer.println("</form>");

            List<Entry> entries = _entryLocalService.getEntries(QueryUtil.ALL_POS, QueryUtil.ALL_POS);

            if (entries.isEmpty()) {
                writer.println("I'm so lonely! :(<br/>");
            }
            else {
                writer.println("Here's a list of others who've already signed up:<br/>");
                for (Entry entry : _entryLocalService.getEntries(QueryUtil.ALL_POS, QueryUtil.ALL_POS)) {
                    writer.println(String.format("%s &lt;%s><br/>", entry.getFullName(), entry.getEmailAddress()));
                }
            }

            return;
        }

        String emailAddressParameter = ParamUtil.getString(request, "emailAddress");

        if (Validator.isNull(emailAddressParameter)) {
            writer.println(String.format("Ooops! %s, you forgot your emailAddress :(<br/>", fullNameParameter));
            writer.println("<a href='/'>Retry?</a>");

            return;
        }

        DynamicQuery dynamicQuery = _entryLocalService.dynamicQuery();

        dynamicQuery.add(RestrictionsFactoryUtil.eq("emailAddress", emailAddressParameter));

        long count = _entryLocalService.dynamicQueryCount(dynamicQuery);

        if (count > 0) {
            writer.println(String.format("Ooops! Someone already registered with the email address &lt;%s> :(<br/>", emailAddressParameter));
            writer.println("<a href='/'>Retry?</a>");

            return;
        }

        long entryId = _counterLocalService.increment();

        Entry entry = _entryLocalService.createEntry(entryId);

        entry.setFullName(fullNameParameter);
        entry.setEmailAddress(emailAddressParameter);

        _entryLocalService.updateEntry(entry);

        writer.println(String.format("Great! Thanks for signing up %s :D<br/>", fullNameParameter));
        writer.println("<a href='/'>Go Back!</a>");
    }

    @Reference
    private CounterLocalService _counterLocalService;

    @Reference
    private EntryLocalService _entryLocalService;

}
```

Note how it uses OSGi Declarative Services to reference an instance of Portal Kernel's `CounterLocalService` and Contacts API's `EntryLocalService`.

Related Topics

Arquillian Extension for Liferay Example
Unit Testing with JUnit

113.7 Injecting Service Components into Tests

You can use Liferay DXP's `@Inject` annotation to inject service components into a test, like you use the `@Reference` annotation to inject service components into a module component.

`@Inject` uses reflection to inject a field with a service component object matching the field's interface. Test rule `LiferayIntegrationTestRule` provides this annotation. The annotation accepts `filter` and `type` parameters, which you can use separately or together.

DXP Digital Enterprise 7.0 Fix Pack 30 and Liferay CE Portal 7.0 GA5 introduced the `@Inject` annotation.

To fill a field with a particular implementation or sub-class object, set the `type` with it.

```
@Inject(type = SubClass.class)
```

Replace `SubClass` with the name of the service interface to inject.

Here's an example test class that injects a `DDLServiceUpgrade` object into an `UpgradeStepRegistrator` interface field:

```
public class Test {

    @ClassRule
    @Rule
    public static final AggregateTestRule aggregateTestRule =
        new LiferayIntegrationTestRule();

    @Test
    public void testSomething() {
        // your test code here
    }

    @Inject(
        filter = "(&(objectClass=com.liferay.dynamic.data.lists.internal.upgrade.DDLServiceUpgrade))"
    )
    private static UpgradeStepRegistrator _upgradeStepRegistrator;

}
```

Here's how to inject a service component into a test class:

1. In your test class, add a rule field of type `com.liferay.portal.test.rule.LiferayIntegrationTestRule`. For example,

   ```
   @ClassRule
   @Rule
   public static final AggregateTestRule aggregateTestRule =
       new LiferayIntegrationTestRule();
   ```

2. Add a field to hold a service component. Making the field static improves efficiency, because the container injects static fields once before test runs and nulls them after all tests run. Non-static fields are injected before each test run but stay in memory till all tests finish.

3. Annotate the field with an `@Inject` annotation. By default, the container injects the field with a service component object matching the field's type.

4. Optionally add a `filter` string or `type` parameter to further specify the service component object to inject.

At runtime, the `@Inject` annotation blocks the test until a matching service component is available. The block has a timeout and messages are logged regarding the test's unavailable dependencies.

Important: If you're publishing the service component you are injecting, the test might never run. If you must publish the service component from the test class, use Service Trackers to access service components.

Great! Now you can inject service components into your tests.

Related Articles

Service Trackers
Finding and Invoking Liferay Services
Unit Testing with JUnit

Chapter 114

Modularity and OSGi

Things we use every day are made of carefully designed, created, and tested subsystems. For example, a car has an engine, suspension, and air conditioner. Teams of engineers, machinists, and technicians make these subsystems the best they can be separately before combining them to create a high quality car. This is modularity in action–creating things from smaller well-designed, well-tested parts.

Liferay DXP is modular too. It comprises code modules created and tested independently and in parallel. It's a platform on which modules and modular applications are installed, started, used, stopped, and uninstalled. Liferay DXP's components use the OSGi modularity standard.

These tutorials demonstrate developing OSGi services and components to customize Liferay DXP and create applications on it. As Liferay's developers used modules to create Liferay DXP and its applications, you and your teammates can enjoy developing your own modules, applications, and customizations in parallel on Liferay DXP.

114.1 The Benefits of Modularity

Dictionary.com defines modularity as *the use of individually distinct functional units, as in assembling an electronic or mechanical system*. The distinct functional units are called *modules*.

NASA's Apollo spacecraft, for example, comprised three modules, each with a distinct function:

- *Lunar Module*: Carried astronauts from the Apollo spacecraft to the moon's surface and back.
- *Service Module*: Provided fuel for propulsion, air conditioning, and water.
- *Command Module*: Housed the astronauts and communication and navigation controls.

The spacecraft and its modules exemplified these modularity characteristics:

- **Distinct functionality**: Each module provides a distinct function (purpose); modules can be combined to provide an entirely new collective function.

  The Apollo spacecraft's modules were grouped together for a distinct collective function: take astronauts from the Earth's atmospheric rim, to the moon's surface, and back to Earth. The previous list identifies each module's distinct function.

- **Dependencies**: Modules can require capabilities other modules satisfy.

  The Apollo modules had these dependencies:

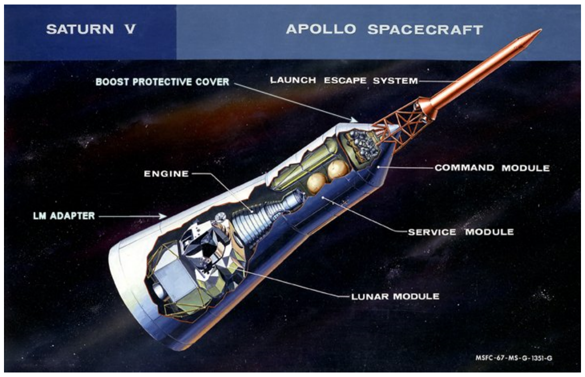


Figure 114.1: The Apollo spacecraft's modules collectively took astronauts to the moon's surface and back to Earth.

  - Lunar Module depended on the Service Module to get near the moon.
  - Command Module depended on the Service Module for power and oxygen.
  - Service Module depended on the Command Module for instruction.

- **Encapsulation**: Modules hide their implementation details but publicly define their capabilities and interfaces.

  Each Apollo module was commissioned with a contract defining its capabilities and interface, while each module's details were encapsulated (hidden) from other modules. NASA integrated the modules based on their interfaces.

- **Reusability**: A module can be applied to different scenarios.

  The Command Module's structure and design were reusable. NASA used different versions of the Command Module, for example, throughout the Apollo program, and in the Gemini Program, which focused on Earth orbit.

NASA used modularity to successfully complete over a dozen missions to the moon. Can modularity benefit software too? Yes! The following sections show you how:

- Modularity benefits for software
- Example: How to design a modular application

Modularity Benefits for Software

Java applications have predominantly been monolithic: they're developed in large code bases. In a monolith, it's difficult to avoid tight coupling of classes. Modular application design, conversely, facilitates loose coupling, making the code easier to maintain. It's much easier and more fun to develop small amounts of cohesive code in modules. Here are some key benefits of developing modular software.

Distinct Functionality

It's natural to focus on developing one piece of software at a time. In a module, you work on a small set of classes to define and implement the module's function. Keeping scope small facilitates writing high quality, elegant code. The more cohesive the code, the easier it is to test, debug, and maintain. Modules can be combined to provide a new function, distinguishable from each module's function.

Encapsulation

A module encapsulates a function (capability). Module implementations are hidden from consumers, so you can create and modify them as you like. Throughout a module's lifetime, you can fix and improve the implementation or swap in an entirely new one. You make the changes behind the scenes, transparent to consumers. A module's contract defines its capability and interface, making the module easy to understand and use.

Dependencies

Modules have requirements and capabilities. The interaction between modules is a function of the capability of one satisfying the requirement of another and so on. Modules are published to artifact repositories, such as Maven Central. Module versioning schemes let you specify dependencies on particular module versions or version ranges.

Reusability

Modules that do their job well are hot commodities. They're reusable across projects, for different purposes. As you discover helpful reliable modules, you'll use them again and again.

It's time to design a modular application.

Example: Designing a Modular Application

Application design often starts out simple but gets more complex as you determine capabilities the application requires. If a third party library already provides the capability, you can deploy it with your app. You can otherwise implement the capability yourself.

As you design various aspects of your app to support its function, you must decide how those aspects fit into the code base. Putting them in a single monolithic code base often leads to tight coupling, while designating separate modules for each aspect fosters loose coupling. Adopting a modular approach to application design lets you reap the modularity benefits.

For example, you can apply modular design to a speech recognition app. Here are the app's function and required capabilities:

Function: interface with users to translate their speech into text for the computer to understand.

Required capabilities: - Translates user words to text - Uses a selected computer voice to speak to users. - Interacts with users based on a script of instructions that include questions, commands, requests, and confirmations.

You could create modules to provide the required capabilities:

- *Speech to text*: Translates spoken words to text the computer understands.
- *Voice UI*: Interacts with users based on stored questions, commands, and confirmations.
- *Instruction manager*: Stores and provides the application's questions, commands, and confirmations.
- *Computer voice*: Stores and provides computer voices for users to choose from.

The following diagram contrasts a monolithic design for the speech recognition application with a modular design.

Designing the app as a monolith lumps everything together. There are no initial boundaries between the application aspects, whereas the modular design distinguishes the aspects.

Developers can create the modules in parallel, each one with its own particular capability. Designing applications that comprise modules fosters writing cohesive pieces of code that represent capabilities. Each module's capability can potentially be *reused* in other scenarios too.

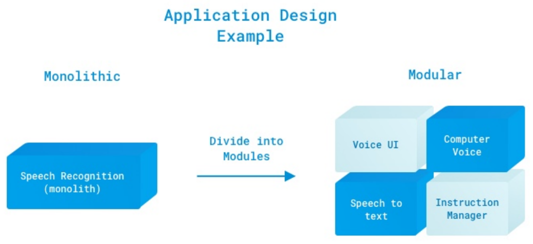


Figure 114.2: The speech recognition application can be implemented in a single monolithic code base or in modules, each focused on a particular function.

For example, the *Instruction manager* and *Computer voice* modules can be *reused* by a navigation app.

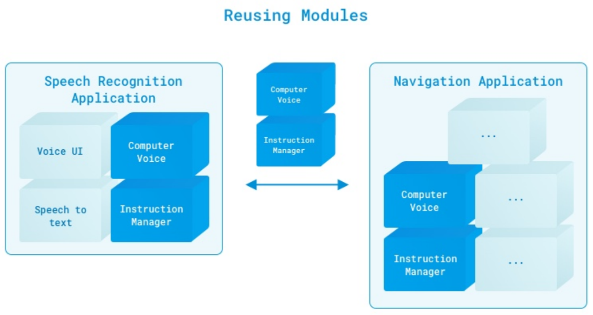


Figure 114.3: The *Instruction manager* and *Computer voice* modules designed for the speech recognition app can be used (or *reused*) by a navigation app.

Here are the benefits of designing the speech recognition app as modules:

- Each module represents a capability that contributes to the app's overall function.
- The app depends on modules, that are easy to develop, test, and maintain.
- The modules can be reused in different applications.

In conclusion, modularity has literally taken us to the moon and back. It benefits software development too. The example speech recognition application demonstrated how to design an app that comprises modules.

Next you'll learn how OSGi facilitates creating modules that provide and consume services.

114.2 OSGi and Modularity

Modularity makes writing software, especially as a team, fun! Here are some benefits to modular development on DXP:

- Liferay DXP's runtime framework is lightweight, fast, and secure.
- The framework uses the OSGi standard. If you have experience using OSGi with other projects, you can apply your existing knowledge to developing on DXP.
- Modules publish services to and consume services from a service registry. Service contracts are loosely coupled from service providers and consumers, and the registry manages the contracts automatically.
- Modules' dependencies are managed automatically by the container, dynamically (no restart required).

- The container manages module life cycles dynamically. Modules can be installed, started, updated, stopped, and uninstalled while Liferay DXP is running, making deployment a snap.
- Only a module's classes whose packages are explicitly exported are publicly visible; OSGi hides all other classes by default.
- Modules and packages are semantically versioned and declare dependencies on specific versions of other packages. This allows two applications that depend on different versions of the same packages to each depend on their own versions of the packages.
- Team members can develop, test, and improve modules in parallel.
- You can use your existing developer tools and environment to develop modules.

There are many benefits to modular software development with OSGi, and we can only scratch the surface here. Once you start developing modules, you might find it hard to go back to developing any other way.

Modules

It's time to see what module projects look like and see Liferay DXP's modular development features in action. To keep things simple, only project code and structure are shown: you can create modules like these anytime.

These modules collectively provide a command that takes a String and uses it in a greeting. Consider it "Hello World" for modules.

API

The API module is first. It defines the contract that a provider implements and a consumer uses. Here is its structure:

- `greeting-api`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/greeting/api`
      - `Greeting.java`
  - `bnd.bnd`
  - `build.gradle`

Very simple, right? Beyond the Java source file, there are only two other files: a Gradle build script (though you can use any build system you want), and a configuration file called `bnd.bnd`. The `bnd.bnd` file describes and configures the module:

```
Bundle-Name: Greeting API
Bundle-SymbolicName: com.liferay.docs.greeting.api
Bundle-Version: 1.0.0
Export-Package: com.liferay.docs.greeting.api
```

The module's name is *Greeting API*. Its symbolic name–a name that ensures uniqueness–is `com.liferay.docs.greeting.api`. Its semantic version is declared next, and its package is *exported*, which means it's made available to other modules. This module's package is just an API other modules can implement.

Finally, there's the Java class, which in this case is an interface:

```
package com.liferay.docs.greeting.api;

import aQute.bnd.annotation.ProviderType;

@ProviderType
public interface Greeting {

        public void greet(String name);

}
```

The interface's `@ProviderType` annotation tells the service registry that anything implementing the interface is a provider. The interface's one method asks for a `String` and doesn't return anything.

That's it! As you can see, creating modules is not very different from creating other Java projects.

Provider

An interface only defines an API; to do something, it must be implemented. This is what the provider module is for. Here's what a provider module for the Greeting API looks like:

- `greeting-impl`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/greeting/impl`
      - `GreetingImpl.java`
  - `bnd.bnd`
  - `build.gradle`

It has the same structure as the API module: a build script, a `bnd.bnd` configuration file, and an implementation class. The only differences are the file contents. The `bnd.bnd` file is a little different:

```
Bundle-Name: Greeting Impl
Bundle-SymbolicName: com.liferay.docs.greeting.impl
Bundle-Version: 1.0.0
```

The bundle name, symbolic name, and version are all set similarly to the API.

Finally, there's no `Export-Package` declaration. A client (which is the third module you'll create) just wants to use the API: it doesn't care how its implementation works as long as the API returns what it's supposed to return. The client, then, only needs to declare a dependency on the API; the service registry injects the appropriate implementation at runtime.

Pretty cool, eh?

All that's left, then, is the class that provides the implementation:

```
package com.liferay.docs.greeting.impl;

import com.liferay.docs.greeting.api.Greeting;

import org.osgi.service.component.annotations.Component;

@Component(
    immediate = true,
    property = {
    },
    service = Greeting.class
)
public class GreetingImpl implements Greeting {

    @Override
    public void greet(String name) {
        System.out.println("Hello " + name + "!");
    }

}
```

The implementation is simple. It uses the `String` as a name and prints a hello message. A better implementation might be to use Liferay DXP's API to collect all the names of all the users in the system and send each user a greeting notification, but the point here is to keep things simple. You should understand, though, that there's nothing stopping you from replacing this implementation by deploying another module whose Greeting implementation's `@Component` annotation specifies a higher service ranking property (e.g., `"service.ranking:Integer=100"`).

This `@Component` annotation defines three options: `immediate = true`, an empty property list, and the service class that it implements. The `immediate = true` setting means that this module should not be lazy-loaded; the service registry loads it as soon as it's deployed, instead of when it's first used. Using the `@Component` annotation declares the class as a Declarative Services component, which is the most straightforward way to create components for OSGi modules. A component is a POJO that the runtime creates automatically when the module starts.

To compile this module, the API it's implementing must be on the classpath. If you're using Gradle, you'd add the `greetings-api` project to your `dependencies { ... }` block. In a Liferay Workspace module, the dependency looks like this:

```
compileOnly project (':modules:greeting-api')
```

That's all there is to a provider module.

Consumer

The consumer or client uses the API that the API module defines and the provider module implements. DXP has many different kinds of consumer modules. Portlets are the most common consumer module type, but since they are a topic all by themselves, this example stays simple by creating an command for the Apache Felix Gogo shell. Note that consumers can, of course, consume many different APIs to provide functionality.

A consumer module has the same structure as the other module types:

- `greeting-command`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/greeting/command`
      - `GreetingCommand.java`
  - `bnd.bnd`
  - `build.gradle`

Again, you have a build script, a `bnd.bnd` file, and a Java class. This module's `bnd.bnd` file is almost the same as the provider's:

```
Bundle-Name: Greeting Command
Bundle-SymbolicName: com.liferay.docs.greeting.command
Bundle-Version: 1.0.0
```

There's nothing new here: you declare the same things you declared for the provider.

Your Java class has a little bit more going on:

```
package com.liferay.docs.greeting.command;

import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Reference;

import com.liferay.docs.greeting.api.Greeting;

@Component(
    immediate = true,
    property = {
        "osgi.command.scope=greet",
        "osgi.command.function=greet"
    },
    service = Object.class
)
public class GreetingCommand {

    public void greet(String name) {
        Greeting greeting = _greeting;

        greeting.greet(name);
    }

    @Reference
    private Greeting _greeting;

}
```

The `@Component` annotation declares the same attributes, but specifies different properties and a different service. As in Java, where every class is a subclass of `java.lang.Object` (even though you don't need to specify it by default), in Declarative Services, the runtime needs to know the type of class to register. Because you're not implementing any particular type, your parent class is `java.lang.Object`, so you must specify that class as the service. While Java doesn't require you to specify `Object` as the parent when you're creating a class that doesn't inherit anything, Declarative Services does.

The two properties define a command scope and a command function. All commands have a scope to define their context, as it's common for multiple APIs to have similar functions, such as `copy` or `delete`. These properties specify you're creating a command called `greet` in a scope called `greet`. While you get no points for imagination, this sufficiently defines the command.

Since you specified `osgi.command.function=greet` in the `@Component` annotation, your class must have a method named `greet`, and you do. But how does this `greet` method work? It obtains an instance of the `Greeting` OSGi service and invokes its `greet` method, passing in the `name` parameter. How is an instance of the `Greeting` OSGi service obtained? The `GreetingCommand` class declares a private service bean, `_greeting` of type `Greeting`. This is the OSGi service type that the provider module registers. The `@Reference` annotation tells the OSGi runtime to instantiate the service bean with a service from the service registry. The runtime binds the `Greeting` object of type `GreetingImpl` to the private field `_greeting`. The `greet` method uses the `_greeting` field value.

Just like the provider, the consumer needs to have the API on its classpath in order to compile, but at runtime, since you've declared all the dependencies appropriately, the container knows about these dependencies, and provides them automatically.

If you were to deploy these modules to a DXP instance, you'd be able to attach to the Gogo Shell and execute a command like this:

```
greet:greet "Captain\ Kirk"
```

The shell would then return your greeting:

```
Hello Captain Kirk!
```

This most basic of examples should make it clear that module-based development is easy and straightforward. The API-Provider-Consumer contract fosters loose coupling, making your software easy to manage, enhance, and support.

A Typical Liferay Application

If you look at a typical application from Liferay DXP's source, you'll generally find at least four modules:

- An API module
- A Service (provider) module
- A Test module
- A Web (consumer) module

This is exactly what you'll find for some smaller applications, like the Mentions application that lets users mention other users with the `@username` nomenclature in comments, blogs, or other applications. Larger applications like the Documents and Media library have more modules. In the case of the Documents and Media library, there are separate modules for different document storage back-ends. In the case of the Wiki, there are separate modules for different Wiki engines.

Encapsulating capability variations as modules facilitates extensibility. If you have a document storage back-end that Liferay DXP doesn't yet support, you can implement Liferay's document storage API for your solution by developing a module for it and thus extend Liferay's Documents and Media library. If there's a Wiki dialect that you like better than what Liferay's wiki provides, you can write a module for it and extend Liferay's wiki.

Are you excited yet? Are you ready to start developing? Here are some resources for you to learn more.

Related Topics

Liferay IDE

Liferay Workspace

Blade CLI

Maven

Developing a Web Application

Planning a Plugin Upgrade to Liferay 7

Chapter 115

OSGi Basics for Liferay Development

Liferay leverages the OSGi framework to provide a development environment for modular applications. There are many OSGi best practices that Liferay DXP follows to provide an easy-to-develop-for platform. Here, you're introduced to some OSGi basics and common Liferay best practices for your bundle's (module) development.

115.1 Liferay Portal Classloader Hierarchy

All Liferay DXP applications live in its OSGi container. Portal is a web application deployed on your application server. Portal's Module Framework bundles (modules) live in the OSGi container and have class loaders. All the classloaders from Java's Bootstrap classloader to classloaders for bundle classes and JSPs are part of a hierarchy.

This tutorial explains Liferay DXP's classloader hierarchy and describes how it works in the following contexts:

- Web application, such as Liferay Portal, deployed on the app server
- OSGi bundle deployed in the Module Framework

The following diagram shows Liferay DXP's classloader hierarchy.

Here are the classloader descriptions:

- **Bootstrap**: The JRE's classes (from packages `java.*`) and Java extension classes (from `$JAVA_HOME/lib/ext`). No matter the context, loading all `java.*` classes is delegated to the Bootstrap classloader.
- **System**: Classes configured on the `CLASSPATH` and or passed in via the application server's Java classpath (`-cp` or `-classpath`) parameter.
- **Common**: Classes accessible globally to web applications on the application server.
- **Web Application**: Classes in the application's `WEB-INF/classes` folder and `WEB-INF/lib/*.jar`.
- **Module Framework**: Liferay DXP's OSGi module framework classloader which is used to provide controlled isolation for the module framework bundles.
- **bundle**: Classes from a bundle's packages or from packages other bundles export.

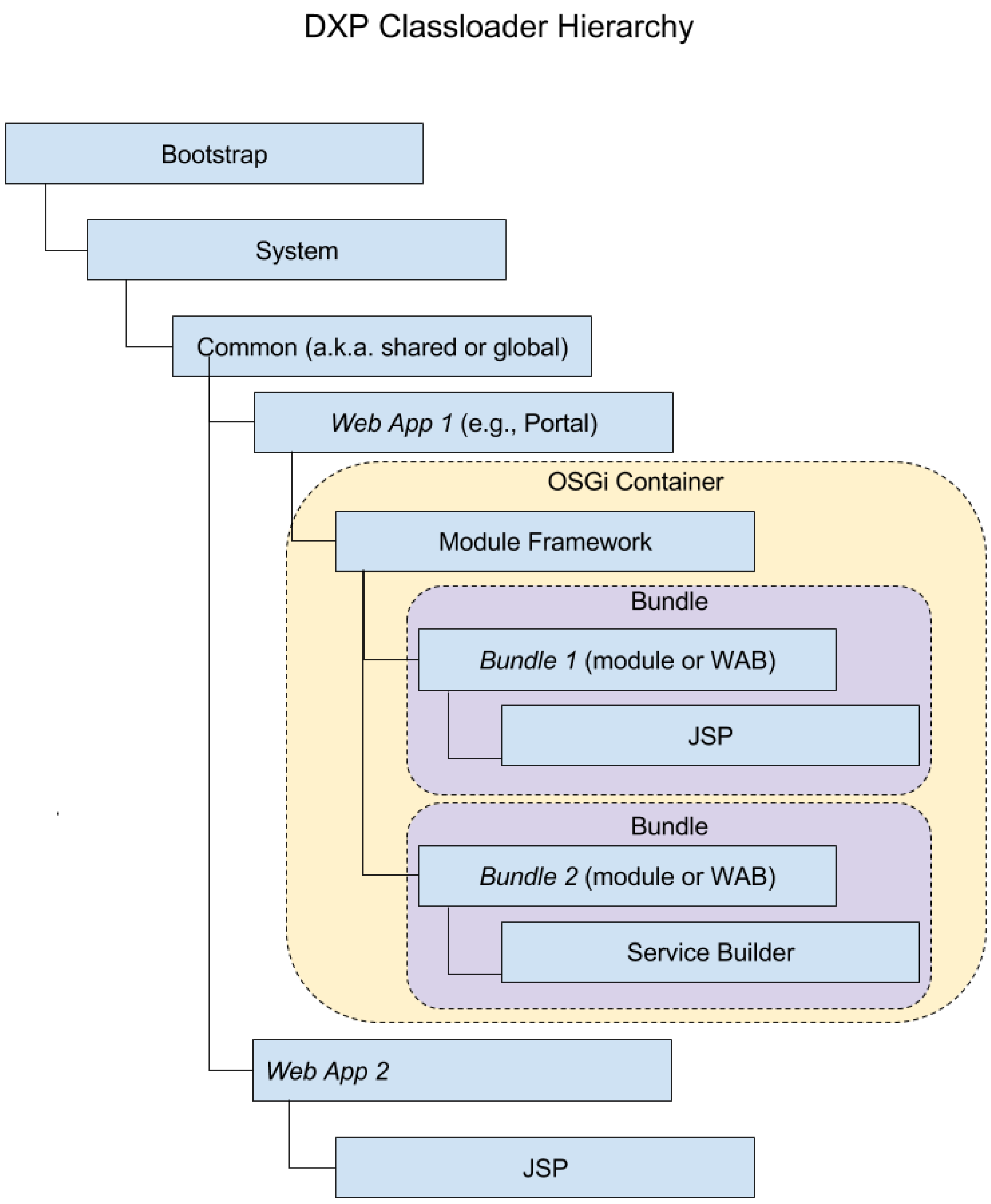


Figure 115.1: 0: Here is Liferay DXP's classloader hierarchy.

- **JSP**: A classloader that aggregates the following bundle and classloaders:
  - Bundle that contains the JSPs' classloader
  - JSP servlet bundle's classloader
  - Javax Expression Language (EL) implementation bundle's classloader
  - Javax JSTL implementation bundle's classloader
- **Service Builder**: Service Builder classes

The classloader used depends on context. Classloading rules vary between application servers. Classloading in web applications and OSGi bundles differs too. In all contexts, however, the Bootstrap classloader loads classes from `java.*` packages.

Classloading from a web application perspective is up next.

Web Application Classloading Perspective

Application servers dictate where and in what order web applications, such as Liferay DXP, search for classes and resources. Application servers such as Apache Tomcat enforce the following default search order:

1. Bootstrap classes
2. Web app's `WEB-INF/classes`
3. web app's `WEB-INF/lib/*.jar`
4. System classloader
5. Common classloader

First, the web application searches Bootstrap. If the class/resource isn't there, the web application searches its own classes and JARs. If the class/resource still isn't found, it checks the System classloader and then Common classloader. Except for the web application checking its own classes and JARs, it searches the hierarchy in parent-first order.

Application servers such as Oracle WebLogic and IBM WebSphere have additional classloaders. They may also have a different classloader hierarchy and search order. Consult your application server's documentation for classloading details.

Other Classloading Perspectives

The Bundle Classloading Flow tutorial explains classloading from an OSGi bundle perspective.

Classloading for JSPs and Service Builder classes is similar to that of web applications and OSGi bundle classes.

You now know Liferay DXP's classloading hierarchy, understand it in context of web applications, and have references to information on other classloading perspectives.

Related Topics

Bundle Classloading Flow

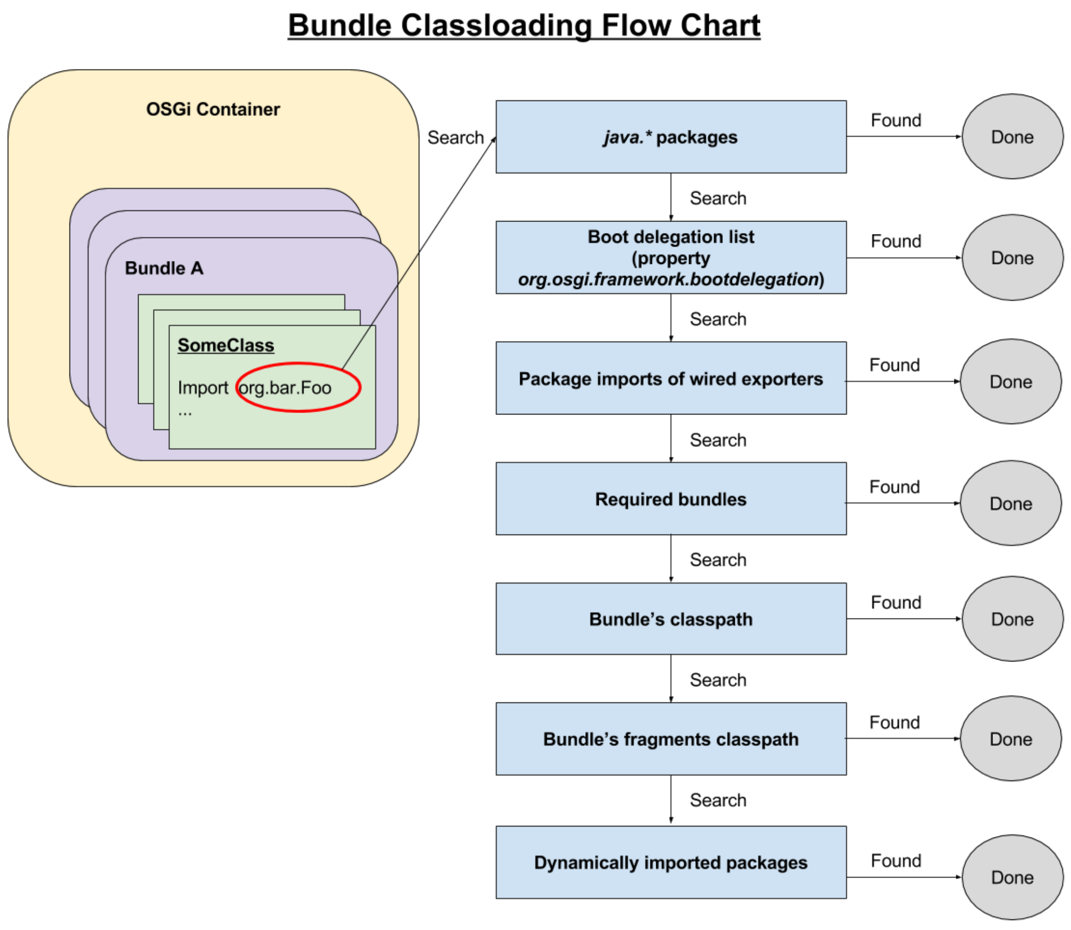


Figure 115.2: 0: This flow chart illustrates classloading in a bundle.

115.2 Bundle Classloading Flow

The OSGi container searches several places for imported classes. It's important to know where it looks and in what order. Liferay DXP's classloading flow for OSGi bundles follows the OSGi Core specification. It's straightforward, but complex. The figure below illustrates the flow and this tutorial walks you through it.

Here is the algorithm for classloading in a bundle:

1. If the class is in a `java.*` package, delegate loading to the parent classloader. Otherwise, continue.

2. If the class is in the OSGi Framework's boot delegation list, delegate loading to the parent classloader. Otherwise, continue.

3. If the class is in one of the packages the bundle imports from a wired exporter, the exporting bundle's classloader loads it. A *wired exporter* is another bundle's classloader that previously loaded the package. If the class isn't found, continue.

4. If the class is imported by one of the bundle's required bundles, the required bundle's classloader loads it.

5. If the class is in the bundle's classpath (manifest header `Bundle-ClassPath`), the bundle's classloader loads it. Otherwise, continue.

6. If the class is in the bundle's fragments classpath, the bundle's classloader loads it.

7. If the class is in a package that's dynamically imported using `DynamicImport-Package` and a wire is established with the exporting bundle, the exporting bundle's classloader loads it. Otherwise, the class isn't found.

Congratulations! Now you know how Liferay DXP finds and loads classes OSGi bundles use.

115.3 Resolving Third Party Library Package Dependencies

The OSGi framework lets you build applications composed of multiple modules. The modules must resolve their Java package dependencies for the framework to assemble the modules into a working system. In a perfect world, every Java library would be an OSGi bundle (module), but many libraries aren't. So how do you resolve the packages your OSGi module needs from non-OSGi third party libraries?

Here is the main workflow for resolving third party Java library packages:

Step 1 - Find an OSGi module of the library: Projects, such as Eclipse Orbit and ServiceMix Bundles, convert hundreds of traditional Java libraries to OSGi modules. Their artifacts are available at these locations:

- Eclipse Orbit
- ServiceMix Bundles

Deploying the module to Liferay's OSGi framework lets you share it on the system. If you find a module for the library you need, deploy it. Then add a `compileOnly` (Gradle) or `<scope>provided</scope>` (Maven) dependency for it in your module. When you deploy your module, the OSGi framework wires the dependency module to your module. If there's no OSGi module based on the Java library, go to Step 2.

Tip: Refrain from embedding library JARs that provide the same packages that Liferay DXP or existing modules provide already.

Note: If you're developing a WAR that requires a different version of a third-party package that Liferay DXP or another module exports, specify that package in your `Import-Package:` list. If the package provider is an OSGi module, publish its exported packages by deploying that module. Otherwise, rename the third-party library (not an OSGi module) differently from the JAR that the WAB generator excludes and embed the JAR in your project.

Step 2 - Resolve the Java packages privately in your module: You can copy required library packages into your OSGi module or embed them wholesale, if you must. The rest of the tutorial shows you how to do this.

Note: Liferay's Gradle plugin `com.liferay.plugin` automates several third party library configuration steps. The plugin is applied to Liferay Workspace Gradle module projects created in Liferay @ide@ or using Liferay Blade CLI automatically.

To leverage the `com.liferay.plugin` plugin outside of Liferay Workspace, add code like the listing below to your Gradle project:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins", version: "3.2.29"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.plugin"
```

If you use Gradle without the `com.liferay.plugin` plugin, you'll have to embed the third party libraries wholesale.

The recommended package resolution workflow is next.

Library Package Resolution Workflow

When you depend on a library JAR, much of the time you only need parts of it. Explicitly specifying only the Java packages you need makes your bundle more modular. This also insulates modules that depend on your module from unneeded packages.

Here's a configuration workflow that minimizes dependencies and Java package imports:

1. Add the library as a compile-only dependency (e.g., `compileOnly` in Gradle).

2. Copy only the library packages you need by specifying them in a conditional package instruction (`Conditional-Package`) in your `bnd.bnd` file. Here are some examples:

   `Conditional-Package: foo.common*` adds packages your module uses such as `foo.common`, `foo.common-messages`, `foo.common-web` to your module's class path.

   `Conditional-Package: foo.bar.*` adds packages your module uses such as `foo.bar` and all its sub-packages (e.g., `foo.bar.baz`, `foo.bar.biz`, etc.) to your module's class path.

   Deploy your module. If a class your module needs or class its dependencies need isn't found, go back to main workflow **Step 1 - Find an OSGi module version of the library** to resolve it.

   Important: Resolving packages by using compile-only dependencies and conditional package instructions assures you use only the packages you need and avoids unnecessary transitive dependencies. It's recommended to use the steps up to this point, as much as possible, to resolve required packages.

3. If a library package you depend on requires non-class files (e.g., DLLs, descriptors) from the library, then you might need to embed the library wholesale in your module. This adds the entire library to your module's classpath.

Next you'll learn how to embed libraries in your module.

Embedding Libraries in a Module

You can use Gradle, Maven, or Ivy to embed libraries in your module. Below are examples for adding Apache Shiro using all three build utilities.

Embedding a Library Using Gradle

Open your module's `build.gradle` file and add the library as a dependency in the `compileInclude` configuration:

```
dependencies {
    compileInclude group: 'org.apache.shiro', name: 'shiro-core', version: '1.1.0'
}
```

The `com.liferay.plugin` plugin's `compileInclude` configuration is transitive. The `compileInclude` configuration embeds the artifact and all its dependencies in a `lib` folder in the module's JAR. Also, it adds the artifact JARs to the module's `Bundle-ClassPath` manifest header.

Note: The `compileInclude` configuration does not download transitive optional dependencies. If your module requires such artifacts, add them as you would another third party library.

Note: If the library you've added as a dependency in your `build.gradle` file has transitive dependencies, you can reference them in an `-includeresource:` instruction by name without having to add them explicitly to the list of dependencies. See how it's used in the Maven section next.

Embedding a Library Using Maven or Ivy

Follow these steps:

1. Open your module's build file and add the library as a dependency in the `provided` scope:

   Maven:

   ```
   <dependency>
     <groupId>org.apache.shiro</groupId>
     <artifactId>shiro-core</artifactId>
     <version>1.1.0</version>
     <scope>provided</scope>
   </dependency>
   ```

   Ant/Ivy:

   ```
   <dependency conf="provided" name="shiro-core" org="org.apache.shiro" rev="1.1.0" />
   ```

2. Open your module's `bnd.bnd` file and add the library to an `-includeresource` instruction:

   ```
   -includeresource: META-INF/lib/shiro-core.jar=shiro-core-[0-9]*.jar;lib:=true
   ```

   This instruction adds the `shiro-core-[version].jar` file as an included resource in the module's `META-INF/lib` folder. The `META-INF/lib/shiro-core.jar` is your module's embedded library. The expression `[0-9]*` helps the build tool match the library version to make available on the module's classpath. The `lib:=true` directive adds the embedded JAR to the module's classpath via the `Bundle-Classpath` manifest header.

Lastly, if after embedding a library you get unresolved imports when trying to deploy to Liferay, you may need to blacklist some imports:

```
Import-Package:\
    !foo.bar.baz,\
    *
```

The * character represents all packages that the module refers to explicitly. Bnd detects the referenced packages.

Congratulations! Resolving all of your module's package dependencies, especially those from traditional Java libraries, is a quite an accomplishment.

Related Topics

Importing Packages
Exporting Packages
Creating Modules with Blade CLI

115.4 Overriding Reluctant Service References

When there's an existing service that you want to customize or implement differently, you can override the existing one. To do this, you create and deploy a new, higher-ranked service implementation. But how do you replace a component's service that's bound by a static and reluctant reference? Reactivating the component would bind it to the new service but would render the component temporarily inactive. To replace the service and keep the component active, you can change the component's service reference to target your new service.

Here are the steps for overriding a component's service reference:

1. Find the component and service details.
2. Create a custom service.
3. Configure the component to use the custom service.

Throughout the tutorial, example modules `override-my-service-reference` and `overriding-service-reference` are used. You can download them and build them using Gradle (bundled with each module) or you can apply the tutorial steps to configure your own customization. Executing `gradlew jar` in each example module root generates the module JAR to the `build/libs` folder.

- `override-my-service-reference` (download): This module's portlet component `OverrideMyServiceReferencePortlet`'s field `_someService` references a service of type `SomeService`. The reference's policy is static and reluctant. By default, it binds to an implementation called `SomeServiceImpl`.

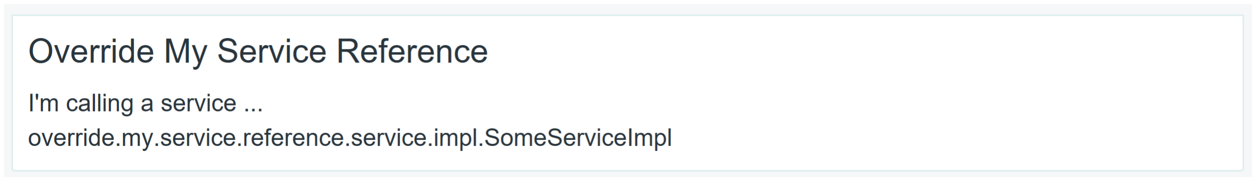
- `overriding-service-reference` (download): Provides a custom `SomeService` implementation called `CustomServiceImpl`. The module's configuration file overrides `OverrideMyServiceReferencePortlet`'s `SomeService` reference so that it binds to `CustomServiceImpl`.

The first step to overriding a service reference is finding the name of the component, service reference, and service interface. If you already have them, you can skip the next section.

Find the Component and Service Reference

You must have the following information to create a custom service and configure the component to use it.

- *Component name*: Name of the component whose service to replace.
- *Reference name*: Name of the component field that references the service.


Override My Service Reference

I'm calling a service ...
override.my.service.reference.service.impl.SomeServiceImpl

Figure 115.3: Prior to overriding the service reference in example portlet module `override-my-service-reference`, the portlet's message indicates it's calling the default service implementation `override.my.service.reference.service.impl.SomeServiceImpl`

- *Service interface*: Fully qualified name of the referenced service interface.

You can find the component using Liferay DXP's Application Manager and find the service reference information using Felix Gogo Shell.

Gogo Shell's Service Component Runtime (SCR) commands help you inspect components. The Gogo Shell command `scr:info [componentName]` lists the component's attributes, including the services it uses. Execute the command using Liferay Blade CLI or in Gogo Shell via telnet.

Here's an example of executing the `scr:info` command in a Gogo Shell telnet session:

```
scr:info override.my.service.reference.OverrideMyServiceReference
```

The resulting SCR information includes the component's service references.

For example, the following information describes the component's reference to service `SomeService`:

```
...
Component Description:
    Name: override.my.service.reference.portlet.OverrideMyServiceReferencePortlet
...
Reference: _someService
    Interface Name: override.my.service.reference.service.api.SomeService
    Cardinality: 1..1
    Policy: static
    Policy option: reluctant
    Reference Scope: bundle
...
```

Copy the following values from the command results. You'll use them in the custom service and service reference configuration you create later.

- *Component*: The component name you passed to the `scr:info` command.
- *Reference*: The *Reference* value.
- *Interface*: The *Interface Name* in the *Reference* section.

Note that the example result confirm's that the reference's policy and policy option are `static` and `reluctant`, respectively.

Here are the values for the example:

- *Component name*: `override.my.service.reference.portlet.OverrideMyServiceReferencePortlet`
- *Reference name*: `_someService`
- *Service interface*: `override.my.service.reference.service.api.SomeService`

The `scr:info` result's component configuration describes the service component implementation bound to the reference.

```
...
Component Configuration:
ComponentId: 2399
State: active
SatisfiedReference: _someService
  Target: null
  Bound to:        6840
      Properties:
        component.id = 2400
        component.name = override.my.service.reference.service.impl.SomeServiceImpl
        objectClass = [override.my.service.reference.service.api.SomeService]
        service.bundleid = 524
        service.id = 6840
        service.scope = bundle
...
```

The example's reference is bound to a component named `override.my.service.reference.service.impl.SomeServiceImpl`. By the end of this tutorial, the reference will be reconfigured to bind to a custom service implementation.

Note: OSGi Configuration Admin makes all Declarative Services components configurable, even if they don't explicitly declare anything about configuration. Each `@Reference` annotation in the source code has a name property, either *explicitly* set in the annotation or *implicitly* derived from the name of the member on which the annotation is used.

- If no reference name property is used and the `@Reference` is on a field, then the reference name is the field name.
- If the `@Reference` is on a method, then heuristics derive the reference name. Method name prefixes such as `set`, `add`, and `put` are ignored. If `@Reference` is on a method called`setSearchEngine(SearchEngine se)`, for example, then the reference name is `SearchEngine`.

Once you've found the referenced service component implementation, you can implement a replacement for it. If you've already created one, you can skip this section.

Create Your Service

It's time to create your own service implementation. Refer to the appropriate app, app suite, and Liferay DXP module Javadoc for service interface details.

Create a module and implement your service in it. Use the `@Component` annotation to make the service a Declarative Services component.

The example custom implementation for service `SomeService` looks like this:

```
@Component(
    immediate = true,
    service = SomeService.class
)
public class CustomServiceImpl implements SomeService {

    @Override
    public String doSomething() {

        StringBuilder sb = new StringBuilder();
        sb.append(this.getClass().getName());
        sb.append(", which delegates to ");
        sb.append(_defaultService.doSomething());

        return sb.toString();
    }
    @Reference  (
        unbind = "-",
        target = "(component.name=override.my.service.reference.service.impl.SomeServiceImpl)"
    )
    private SomeService _defaultService;
}
```

The service component above refers to the default service so that it can delegate work to it in its `doSomething` method. The reference targets the default service by its component name `override.my.service.reference.service.impl.SomeServiceImpl`.

To register your service with the Liferay DXP's OSGi runtime framework, deploy its module. To bind the component reference to your custom service, you must create and deploy instructions that configure the component reference to target your custom service.

Configure the Component to Use Your Service

You're ready to change the component's service reference to target your service. Liferay DXP's Configuration Admin lets you use configuration files to swap in service references on the fly.

1. Create a configuration file named after the referencing component. Here's the example component's configuration file name:

   ```
   override.my.service.reference.portlet.OverrideMyServiceReferencePortlet.config
   ```

```
**Note:** Liferay DXP DE 7.0 Fix Pack 8 and later, and Liferay CE Portal 7.0
GA4 and later support the Apache Felix ConfigAdmin implementation of OSGi
Configuration Admin files. Felix ConfigAdmin uses the file suffix `.config`
and supports additional types, such as arrays and vectors. The syntax for
`.config` and `.cfg` files can be found
[here](https://sling.apache.org/documentation/bundles/configuration-installer-factory.html).
```

---

2. In the configuration file, add a reference target entry that filters on your custom service. Follow this format for the entry:

   ```
   [reference].target=[filter]
   ```

   Replace `[reference]` with the name of the reference you're overriding. Replace `[filter]` with service properties that filter on your custom service.

```
**Tip**: You can use a `component.name` or `objectClass` reference to filter
on your custom implementation.
```

```
A `.config` file reference target entry for the example looks like this:

    _someService.target="(component.name\=overriding.service.reference.service.CustomServiceImpl)"

[The `.config` file syntax](https://sling.apache.org/documentation/bundles/configuration-installer-factory.html#configuration-files-
config)
requires surrounding the value in double quotes and escaping the value's
equals sign.

A `.cfg` file entry for the example looks like this:

    _someService.target=(component.name=overriding.service.reference.service.CustomServiceImpl)
```

3. Optionally, you can add a `cardinality.minimum` entry to specify the number of services the reference can use. Here's the format:

   ```
   [reference].cardinality.minimum=[int]
   ```

   Here's an example cardinality minimum:

   ```
   _someService.cardinality.minimum=1
   ```

4. Deploy the configuration by copying the configuration file into the folder `[Liferay_Home]/osgi/configs`.

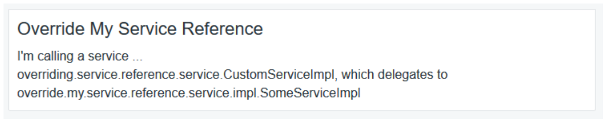
Executing `scr:info` on your component shows that the custom service implementation is now bound to the reference.

For example, executing `scr:info override.my.service.reference.portlet.OverrideMyServiceReferencePortlet` reports the following information:

```
...
Component Description:
  Name: override.my.service.reference.portlet.OverrideMyServiceReferencePortlet
  ...
  Reference: _someService
    Interface Name: override.my.service.reference.service.api.SomeService
    Cardinality: 1..1
    Policy: static
    Policy option: reluctant
    Reference Scope: bundle
    ...
  Component Configuration:
    ComponentId: 2399
    State: active
    SatisfiedReference: _someService
      Target: (component.name=overriding.service.reference.CustomServiceImpl)
      Bound to:        6841
          Properties:
            _defaultService.target = (component.name=overriding.service.reference.service.CustomServiceImpl)
            component.id = 2398
            component.name = overriding.service.reference.service.CustomServiceImpl
            objectClass = [override.my.service.reference.service.api.SomeService]
            service.bundleid = 525
            service.id = 6841
            service.scope = bundle
      Component Configuration Properties:
        _someService.target = (component.name=overriding.service.reference.service.CustomServiceImpl)
        ...
```

The example component's `_someService` reference targets custom service component `overriding.service.reference.servi`
`CustomServiceImpl` references default service `SomeServiceImpl` to delegates work to it.

Liferay DXP processed the configuration file and injected the service reference, which in turn bound the custom service to the referencing component!


Override My Service Reference

I'm calling a service ...
overriding.service.reference.service.CustomServiceImpl, which delegates to
override.my.service.reference.service.impl.SomeServiceImpl

Figure 115.4: Because the example component's service reference is overridden by the configuration file deployment, the portlet indicates it's calling the custom service.

Related Topics

- Finding Extension Points
- Using Felix Gogo Shell

115.5 Using the WAB Generator

Developers creating applications for 7.0 can choose to create them as Java EE-style Web Application ARchive (WAR) artifacts or as Java ARchive (JAR) OSGi bundle artifacts. Some plugin developers, however, don't have that flexibility. Portlets like Spring MVC and JSF must be packaged as WAR artifacts because their frameworks are designed for Java EE. Therefore, they expect a WAR layout and require Java EE resources such as the `WEB-INF/web.xml` descriptor.

Liferay provides a way for these WAR-styled plugins to be deployed and treated like OSGi modules by Liferay's OSGi runtime. They can be converted to *WABs*.

7.0 supports the OSGi Web Application Bundle (WAB) standard for deployment of Java EE style WARs. Simply put, a WAB is an archive that has a WAR layout and contains a `META-INF/MANIFEST.MF` file with the `Bundle-SymbolicName` OSGi directive. A WAB is an OSGi bundle. Although the source of the project has a WAR layout, the artifact filename may end with either the `.jar` or `.war` extension.

Liferay only supports the use of WABs that have been auto-generated by the WAB Generator. The WAB Generator transforms a general WAR-style plugin into a WAB during its deployment process. So what exactly does the WAB Generator do to a WAR file to transform it into a WAB?

The WAB Generator detects packages referenced in a plugin WAR's JSPs, descriptor files, and classes (in `WEB-INF/classes` and embedded JARs). The descriptor files include `web.xml`, `liferay-web.xml`, `portlet.xml`, `liferay-portlet.xml`, and `liferay-hook.xml`. The WAB Generator verifies whether the detected packages are in the plugin's `WEB-INF/classes` folder or in an embedded JAR in the `WEB-INF/lib` folder. Packages that aren't found in either location are added to an `Import-Package` header in the WAB's `META-INF/MANIFEST.MF` file.

To import a package that is only referenced in the following types of locations, you must add an `Import-Package` OSGi header to the plugin's `WEB-INF/liferay-plugin-package.properties` file and add the package to the header's list of values. - Unrecognized descriptor file - Custom or unrecognized descriptor element or attribute - Reflection code - Class loader code

---

Note: A known issue is preventing packages referenced in `web.xml` file `listener-class` elements from being detected and added to WAB `META-INF/MANIFEST.MF` file `Import-Package` headers. To import such packages, add them to an `Import-Package` header in the plugin's `WEB-INF/liferay-plugin-package.properties` file.

The WAB folder structure and WAR folder structure differ. Consider the following folder structure of a WAR-style portlet:

- my-war-portlet
    - src

        * main

            - java
            - webapp
            - WEB-INF
            - classes
            - lib
            - resources
            - views
            - faces-config.xml
            - liferay-display.xml
            - liferay-plugin-package.properties
            - liferay-portlet.xml
            - portlet.xml
            - web.xml

When a WAR-style portlet is deployed to Liferay DXP and processed by the WAB Generator, the portlet's folder structure is transformed to something like this

- my-war-portlet-that-is-now-a-wab

    - META-INF

        * MANIFEST.MF

    - WEB-INF

        * classes
        * lib
        * resources
        * views
        * faces-config.xml
        * liferay-display.xml
        * liferay-plugin-package.properties
        * liferay-portlet.xml
        * portlet.xml
        * web.xml

The major difference is the addition of the `META-INF/MANIFEST.MF` file. The WAB Generator automatically generates an OSGi-ready `MANIFEST.MF` file. If you want to affect the content of the manifest file, you can place BND directives and OSGi headers directly into the `liferay-plugin-package.properties` file. A `bnd.bnd` and/or a build-time plugin (e.g., `bnd-maven-plugin`) should not be provided for your WAR plugin, because the generated WAB cannot make use of them.

Do you want to try this out for yourself? Follow the steps below to see the WAB Generator in action.

1. Create a WAR-style plugin that follows a similar structure to the one outlined above. You can download an example WAR-style portlet here, for demonstration.

2. Open your Liferay DXP instance in a file explorer and add a `portal-ext.properties` file with the following properties:

```
module.framework.web.generator.generated.wabs.store=true
module.framework.web.generator.generated.wabs.store.dir=${module.framework.base.dir}/wabs
```

These properties store your generated WAB into your Liferay DXP instance's `osgi/wabs` folder. You can learn more about these properties in the Module Framework Web Application Bundles properties section. Restart Liferay DXP for these changes to be recognized.

3. Copy your WAR plugin in your Liferay DXP instance's `deploy` folder.

4. Navigate to your Liferay DXP instance's `osgi/wabs` folder and inspect the generated WAB.

Awesome! You've seen the WAB Generator in action!

Related Topics

Generating a JSF Application
Customizing the Product Menu
Configuration

115.6 Importing Packages

Your modules will often need to use Java classes from packages exported by other modules. When a module is set up to import, the OSGi framework finds other registered modules that export the needed packages and wires them to the importing module. At run time, the importing module gets the class from the wired module that exports the class's package.

For this to happen, a module must specify the `Import-Package` OSGi manifest header with a comma-separated list of the Java packages it needs. For example, if a module needs classes from the `javax.portlet` and `com.liferay.portal.kernel.util` packages, it must specify them like so:

```
Import-Package: javax.portlet,com.liferay.portal.kernel.util,*
```

The * character represents all packages that the module refers to explicitly. Bnd detects the referenced packages.

Import packages must sometimes be specified manually, but not always. Conveniently, Liferay DXP project templates and tools automatically detect the packages a module uses and add them to the package imports in the module JAR's manifest. Let's explore how package imports are specified in different scenarios.

Gradle and Maven module projects created using Blade CLI, Liferay's Maven archetypes, or Liferay @ide@ use bnd. On building such a project's module JAR, bnd detects the packages the module uses and generates a `META-INF/MANIFEST.MF` file whose `Import-Package` header specifies the packages.

Note: Liferay's Maven module archetypes use the `bnd-maven-plugin`. Liferay's Gradle module project templates use a third-party Gradle plugin to invoke bnd.

For example, suppose you're developing a Liferay module using Maven or Gradle. In most cases, you specify your module's dependencies in your `pom.xml` or `build.gradle` file. At build time, the Maven or Gradle bundle plugin reads your `pom.xml` or `build.gradle` file and bnd adds the required `Import-Package` headers to your module JAR's `META-INF/MANIFEST.MF`.

Here's an example dependencies section from a module's `build.gradle` file:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

And here's the `Import-Package` header that's generated in the module JAR's `META-INF/MANIFEST.MF` file:

```
Import-Package: com.liferay.portal.kernel.portlet.bridges.mvc;version=
"[1.0,2)",com.liferay.portal.kernel.util;version="[7.0,8)",javax.nami
ng,javax.portlet;version="[2.0,3)",javax.servlet,javax.servlet.http,j
avax.sql
```

Note that your build file need only specify JAR file dependencies. bnd examines your module's class path to determine which packages from those JAR files contain classes your application uses and imports the packages. The examination includes all classes found in the class path–even those from embedded third party library JARs.

Regarding classes used by a traditional Liferay plugin WAR, Liferay's WAB Generator detects their use in the WAR's JSPs, descriptor files, and classes (in `WEB-INF/classes` and embedded JARs). The WAB Generator searches the `web.xml`, `liferay-web.xml`, `portlet.xml`, `liferay-portlet.xml`, and `liferay-hook.xml` descriptor files. It adds package imports for classes that are neither found in the plugin's `WEB-INF/classes` folder nor in embedded JARs.

The WAB Generator and bnd don't add package imports for classes referenced in these places:

- Unrecognized descriptor file
- Custom or unrecognized descriptor element or attribute
- Reflection code
- Class loader code

In such cases, you must manually determine these packages and specify an `Import-Package` OSGi header that includes these packages and the packages that Bnd detects automatically. The `Import-Package` header belongs in the location appropriate to your project type:

Project type	`Import-Package` header location
Module (uses bnd)	`[project]/bnd.bnd`
Module (doesn't use bnd)	`[module JAR]/META-INF/MANIFEST.MF`
Traditional Liferay plugin WAR	`WEB-INF/liferay-plugin-package.properties`

Here's an example of adding a package called `com.liferay.docs.foo` to the list of referenced packages that Bnd detects automatically:

```
Import-Package:\
    com.liferay.docs.foo,\
    *
```

Note: The WAB Generator refrains from adding WAR project embedded third-party JARs to a WAB if Liferay DXP already exports the JAR's packages.

If your WAR requires a different version of a third-party package that Liferay DXP exports, specify that package in your `Import-Package:` list. Then if the package provider is an OSGi module, publish its exported packages by deploying the module. If the package provider is not an OSGi module, follow the instructions for adding third-party libraries.

Please see the `Import-Package` header documentation for more information.

Congratulations! Now you can import all kinds of packages for your modules and plugins to use.

Related Topics

Configuring Dependencies

Resolving a Plugin's Dependencies

Using the WAB Generator

Tooling

115.7 Exporting Packages

An OSGi module's Java packages are private by default. To expose a package, you must explicitly export it. This way you share only the classes you want to share. Exporting a package in your OSGi module JAR's manifest makes all the package's classes available for other modules to import.

To export a package, add it to your module's or plugin's `Export-Package` OSGi header. A header exporting `com.liferay.petra.io` and `com.liferay.petra.io.unsync` would look like this:

```
Export-Package:\
com.liferay.petra.io,\
com.liferay.petra.io.unsync
```

The correct location for the header depends on your project's type:

Project Type	`Export-Package` header location
Module (uses bnd)	`[project]/bnd.bnd`
Module (doesn't use bnd)	`[module JAR]/META-INF/MANIFEST.MF`
Traditional Liferay plugin WAR	`WEB-INF/liferay-plugin-package.properties`

Module projects created using Blade CLI, Liferay's Maven archetypes, or Liferay @ide@ use bnd. On building such a project's module JAR, bnd propagates the OSGi headers from the project's `bnd.bnd` file to the JAR's `META-INF/MANIFEST.MF`.

In module projects that don't use bnd, you must manually add package exports to an `Export-Package` header in the module JAR's `META-INF/MANIFEST.MF`.

In traditional Liferay plugin WAR projects, you must add package exports to an `Export-Package` header in the project's `liferay-plugin-package.properties`. On copying the WAR into the `[Liferay Home]/deploy` folder, the WAB Generator propagates the OSGi headers from the WAR's `liferay-plugin-package.properties` file to the `META-INF/MANIFEST.MF` file in the generated Web Application Bundle (WAB).

Note: bnd makes a module's exported packages *substitutable*. That is, the OSGi framework can substitute your module's exported package with a compatible package of the same name, but potentially different version, that's exported from a different module. bnd enables this for your module by automatically making your module import every package it exports. In this way, your module can work on its own, but can also work in conjunction with modules that provide a different (compatible) version, or even the same version, of the package. A package from another module might provide better "wiring" opportunities with other modules. Peter Kriens' blog post provides more details on how substitutable exports works.

Important: Don't export the same package from different JARs. Multiple exports of the same package leads to "split package" issues, whose side affects differ from case to case.

Now you can share your module's or plugin's terrific [EDITOR: or terrible!] packages with other modules!

Related Topics

Using the WAB Generator
Tooling

115.8 Semantic Versioning

Semantic Versioning is a three tiered versioning system that increments version numbers based on the type of API change introduced to a releasable software component. It's a standard way of communicating programmatic compatibility of a package or module for dependent consumers and API implementations. If a package is programmatically (i.e., semantically) incompatible with a project, Bnd (used when building modules) fails that project's build immediately.

The semantic version format looks like this:

```
MAJOR.MINOR.MICRO
```

Certain events force each tier to be incremented:

- *MAJOR:* an incompatible, API-breaking change is made
- *MINOR:* a change that affects only providers of the API, or new backwards- compatible functionality is added
- *MICRO:* a backwards-compatible bug fix is made

For more details on semantic versioning, see the official Semantic Versioning site and OSGi Alliance's Semantic Versioning technical whitepaper.

All of Liferay DXP's modules use Semantic Versioning.

Following Semantic Versioning is especially important because Liferay DXP is a modular platform containing hundreds of independent OSGi modules. With many independent modules containing a slew of dependencies, releasing new package versions can quickly become terrifying. With this complex intertwined system of dependencies, you must meticulously manage your own project's API versions to ensure compatibility for those who leverage it. With Semantic Versioning's straightforward system and the help of Liferay tooling, managing your project's versions is easy.

Baselining Your Project

Following Semantic Versioning manually seems deceptively easy. There's a sad history of good-intentioned developers updating their projects' semantic versions manually, only to find out later they made a mistake. The truth is, it's hard to anticipate the ramifications of a simple update. To avoid this, you can *baseline* your project after it has been updated. Baselining verifies that the Semantic Versioning rules are obeyed by your project. This can catch many obvious API changes that are not so obvious to humans. Care must always be taken, however, when making any kind of code change because this tool is not smart enough to identify compatibility changes not represented in the signatures of Java classes or interfaces, or in API *use* changes (e.g., assumptions about method call order, or changes to input and/or output encoding). Baseline, as the name implies, does give you a certain measure of *baseline* comfort that a large class of compatibility issues won't sneak past you.

You can use Liferay's Baseline Gradle plugin to provide baselining capabilities. Add it to your Gradle build configuration and execute the following command:

```
./gradlew baseline
```

See the Baseline Gradle Plugin article for configuration details. This plugin is not provided in Liferay Workspace by default.

When you run the `baseline` command, the plugin baselines your new module against the latest released non-snapshot module (i.e., the baseline). That is, it compares the public exported API of your new module with the baseline. If there are any changes, it uses the OSGi Semantic Versioning rules to calculate the minimum new version. If your new module has a lower version, errors are thrown.

With baselining, your project's Semantic Versioning is as accurate as its API expresses.

Managing Artifact and Dependency Versions

There are two ways to track your project's artifact and dependency versions with Semantic Versioning:

- Range of versions
- Exact version (one-to-one)

You should track a range of versions if you intend to build your project for multiple versions of Liferay DXP and maintain maximum compatibility. In other words, if several versions of a package work for an app, you can configure the app to use any of them. What's more, Bnd automatically determines the semantically compatible range of each package a module depends on and records the range to the module's manifest.

For help with version range syntax, see the OSGi Specifications.

A version range for imported packages in an OSGi bundle's `bnd.bnd` looks like this:

```
Import-Package: com.liferay.docs.test; version="[1.0.0,2.0.0)"
```

Popular build tools also follow this syntax. In Gradle, a version range for a dependency looks like this:

```
compile group: "com.liferay.portal", name: "com.liferay.portal.test", version: "[1.0.0,2.0.0)"
```

In Maven, it looks like this:

```
<groupId>com.liferay.portal</groupId>
<artifactId>com.liferay.portal.test</artifactId>
<version>[1.0.0,2.0.0)</version>
```

Specifying the latest release version can also be considered a range of versions with no upper limit. For example, in Gradle, it's specified as `version: "latest.release"`. This can be done in Maven 2.x with the usage of the version marker `RELEASE`. This is not possible if you're using Maven 3.x. See Gradle and Maven's respective docs for more information.

Tracking a range of versions comes with a price. It's hard to reproduce old builds when you're debugging an issue. It also comes with the risk of differing behaviors depending on the version used. Also, relying on the latest release could break compatibility with your project if a major change is introduced. You should proceed with caution when specifying a range of versions and ensure your project is tested on all included versions.

Tracking a dependency's exact version is much safer, but is less flexible. This might limit you to a specific version of Liferay DXP. You would also be locked in to APIs that only exist for that specific version. This means your module is much easier to test and has less chance for unexpected failures.

Note: When specifying package versions in your `bnd.bnd` file, exact versions are typically specified like this: `version="1.1.2"`. However, this syntax is technically a range; it is interpreted as [1.1.2, ∞). Therefore, if a higher version of the package is available, it's used instead of the version you specified. For these cases, it may be better to specify a version range for compatible versions that have been tested. If you want to specify a true exact match, the syntax is like this: `[1.1.2]`. See the Version Range section in the OSGi specifications for more info.

Gradle and Maven use exact versions when only one version is specified.

You now know the pros and cons for tracking dependencies as a range and as an exact match.

115.9 Service Trackers

Now that Liferay is promoting more modular plugins deployed into an OSGi runtime, you have to consider how your own code, living in its own module, can rely on services in other modules for functionality. You must account for the possibility of service implementations being swapped out or removed entirely if your module is to survive and thrive in an OSGi environment. It's easy for 7.0 developers who need to call services from their `@Component` classes. They just use another Declarative Services (DS) annotation, `@Reference`, to get a service reference. The component activates when the referenced service is available.

If you're able to use DS and leverage the `@Component` and `@Reference` annotations, you should. DS handles much of the complexity of handling service dynamism for you transparently.

If you can't use DS to create a Component, keep reading to learn how to implement a Service Tracker to look up services in the service registry.

Note: When using Service Trackers in your WAR-style project, you must configure the required `org.osgi.core` dependency carefully in your build file (e.g., `build.gradle`, `pom.xml`, etc.) to avoid errors. Since it's included in Liferay DXP by default, it must be configured as `provided`. See the Third Party Packages Portal Exports tutorial for more information.

What scenarios might require using a service tracker? Keep in mind we're focusing on scenarios where DS *can't* be used. This typically involves a non-native (to OSGi) Dependency Injection framework.

- Calling OSGi services from a Spring MVC portlet
- Calling OSGi services from a JSF portlet
- Calling OSGi services from a WAR-packaged portlet that's been upgraded to run on 7.0, but not fully modularized and made into an OSGi module

---

Note: The static utility classes (e.g., `UserLocalServiceUtil`) that were useful in Liferay Portal 6.2 (and earlier) exist for compatibility but should not be called, if possible. Static utility classes cannot account for the OSGi runtime's dynamic environment. Using a static class, for example, you might attempt calling a service that has stopped or hasn't been deployed or started–this could cause unrecoverable runtime errors. Service Tracker, however, helps you make OSGi-friendly service calls.

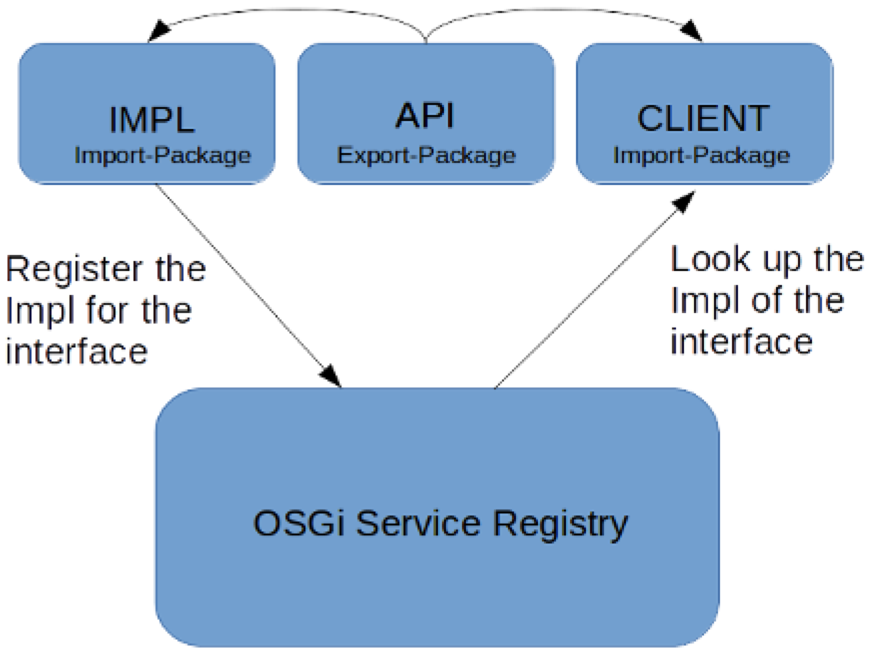


Figure 115.5: Service implementations that are registered in the OSGi service registry can be accessed using Service Trackers.

Using a Service Tracker, your non-OSGi application can access any service registered in the OSGi runtime, including your own Service Builder services and the services published by Liferay's modules (like the popular `UserLocalService`).

Implementing a Service Tracker

Although you don't have the luxury of using DS to manage your service dependencies, you can call services from the service registry with a little bit of code.

To implement a service tracker you can do this:

```
import org.osgi.framework.Bundle;
import org.osgi.framework.FrameworkUtil;
import org.osgi.util.tracker.ServiceTracker;

Bundle bundle = FrameworkUtil.getBundle(this.getClass());
BundleContext bundleContext = bundle.getBundleContext();
ServiceTracker<SomeService, SomeService> serviceTracker =
    new ServiceTracker(bundleContext, SomeService.class, null);
serviceTracker.open();
SomeService someService = serviceTracker.waitForService(500);
```

To simplify your code, you can create a class that extends `org.osgi.util.tracker.ServiceTracker`.

```
public class SomeServiceTracker
    extends ServiceTracker<SomeService, SomeService> {

    public SomeServiceTracker(Object host) {
        super(
            FrameworkUtil.getBundle(host.getClass()).getBundleContext(),
            SomeService.class, null);
    }
}
```

From the initialization part of your logic that uses the service, call your service tracker constructor. The Object host parameter is used to obtain your own bundle context and in order to give accurate results must be an object from your own bundle.

```
ServiceTracker<SomeService, SomeService> someServiceTracker =
    new SomeServiceTracker(this);
```

Remember to open the service tracker before using it, typically as early as you can.

```
someServiceTracker.open();
```

The most basic usage of a Service Tracker is to interrogate the service's state. In your program logic, for example, check whether the service is null before using it:

```
SomeService someService = someServiceTracker.getService();

if (someService == null) {
    _log.warn("The required service 'SomeService' is not available.");
}
else {
    someService.doSomethingCool();
}
```

Service Trackers have several other utility functions for introspecting tracked services.

Later when your application is being destroyed or undeployed, close the service tracker.

```
someServiceTracker.close();
```

Implementing a Callback Handler for Services

If there's a strong possibility the service might not be available, or if you need to track multiple services, the Service Tracker API provides a callback mechanism which operates on service *events*. To use this, override ServiceTracker's addingService and removedService methods. Their ServiceReference parameter references an active service object.

Here's an example ServiceTracker implementation from the OSGi Alliance's OSGi Core Release 7 specification:

```
new ServiceTracker<HttpService, MyServlet>(context, HttpService.class, null) {

    public MyServlet addingService(ServiceReference<HttpService> reference) {
        HttpService httpService = context.getService(reference);
        MyServlet myServlet = new MyServlet(httpService);
        return myServlet;
    }

    public void removedService(
        ServiceReference<HttpService> reference, MyServlet myServlet) {
        myServlet.close();
        context.ungetService(reference);
    }
}
```

When the HttpService is added to the OSGi registry, this ServiceTracker creates a new wrapper class, MyServlet, which uses the newly added service. When the service is removed from the registry, the removedService method cleans up related resources.

As an alternative to directly overloading ServiceTracker methods, create a org.osgi.util.tracker.ServiceTrackerCustomiz

```
class MyServiceTrackerCustomizer
    implements ServiceTrackerCustomizer<SomeService, MyWrapper> {

    private final BundleContext bundleContext;

    MyServiceTrackerCustomizer(BundleContext bundleContext) {
        this.bundleContext = bundleContext;
    }

    @Override
    public MyWrapper addedService(
        ServiceReference<SomeService> serviceReference) {

        // Determine if the service is one that's interesting to you.
        // The return type of this method is the `tracked` type. Its type
        // is what is returned from `getService*` methods; useful for wrapping
        // the service with your own type (e.g., MyWrapper).
        if (isInteresting(serviceReference)) {
            MyWrapper myWrapper = new MyWrapper(
                serviceReference, bundleContext.getService());

            // trigger the logic that requires the available service(s)
            triggerServiceAddedLogic(myWrapper);

            return myWrapper;
        }

        // If the return is null, the tracker is effectively ignoring any further
        // events for the service reference
        return null;
    }

    @Override
    public void modifiedService(
        ServiceReference<SomeService> serviceReference, MyWrapper myWrapper) {
        // handle the modified service
    }

    @Override
    public void removedService(
        ServiceReference<SomeService> serviceReference, MyWrapper myWrapper) {

        // finally, trigger logic when the service is going away
        triggerServiceRemovedLogic(myWrapper);
    }

}
```

Register the `ServiceTrackerCustomizer` by passing it as the `ServiceTracker` constructor's third parameter.

```
ServiceTrackerCustomizer<SomeService, MyWrapper> serviceTrackerCustomizer =
    new MyServiceTrackerCustomizer();

ServiceTracker<SomeService, MyWrapper> serviceTracker =
    new ServiceTracker<>(
        bundleContext, SomeService.class, serviceTrackerCustomizer);
```

There's a little boilerplate code you need to produce, but now you can look up services in the service registry, even if your plugins can't take advantage of the Declarative Services component model.

115.10 Waiting on Lifecycle Events

Liferay registers lifecycle events like portal and database initialization into the OSGi service registry. Your OSGi Component or non-component class can listen for these events by way of their service registrations.

The `ModuleServiceLifecycle` interface defines these names for the lifecycle event services:

- DATABASE_INITIALIZED
- PORTAL_INITIALIZED
- SPRING_INITIALIZED

Here you'll learn how to wait on lifecycle event services to act on them from within a component or non-component class.

Taking action from a component

Declarative Services (DS) facilitates waiting for OSGi services and acting on them once they're available.

Here's a component whose `doSomething` method is invoked once the `ModuleServiceLifecycle.PORTAL_INITIALIZED` lifecycle event service and other services are available.

```
@Component
public class MyXyz implements XyzApi {

    // Plain old OSGi service
    @Reference
    private SomeOsgiService _someOsgiService;

    // Service Builder generated service
    @Reference
    private DDMStructureLocalService _ddmStructureLocalService;

    // Liferay lifecycle service
    @Reference(target = ModuleServiceLifecycle.PORTAL_INITIALIZED)
    private ModuleServiceLifecycle _portalInitialized;

    @Activate
    public void doSomething() {
        // `@Activate` method is only executed once all of
        // `_someOsgiService`,
        // `_ddmStructureLocalService` and
        // `_portalInitialized`
        // are set.
    }
}
```

Here's how to act on services in your component:

1. For each lifecycle event service and OSGi service your component uses, add a field of that service type and add an `@Reference` annotation to that field. The OSGi framework binds the services to your fields. This field, for example, binds to a standard OSGi service.

   ```
   @Reference
   SomeOsgiService _someOsgiService;
   ```

2. To bind to a particular lifecycle event service, target its name as the `ModuleServiceLifecycle` interface defines. This field, for example, targets database initialization.

   ```
   @Reference(target = ModuleServiceLifecycle.DATABASE_INITIALIZED)
   ModuleServiceLifecycle _dataInitialized;
   ```

3. Create a method that's triggered on the event(s) and add the `@Activate` annotation to that method. It's invoked when all the service objects are bound to the component's fields.

Your component fires (via its `@Activate` method) after all its service dependencies resolve. DS components are the easiest way to act on lifecycle event services.

Taking action from a non-component class

Classes that aren't DS components can use a `org.osgi.util.tracker.ServiceTracker` or `org.osgi.util.tracker.ServiceTracker` as a service callback handler for the lifecycle event. If you depend on multiple services, add logic to your `ServiceTracker` or `ServiceTrackerCustomizer` to coordinate taking action when all the services are available.

To target a lifecycle event service, create a service tracker that filters on that service. Use `org.osgi.framework.FrameworkUtil` to create an `org.osgi.framework.Filter` that specifies the service. Then pass that filter as a parameter to the service tracker constructor. For example, this service tracker filters on the lifecycle service `ModuleServiceLifecycle.PORTAL_INITIALIZED`.

```
import org.osgi.framework.Filter;
import org.osgi.framework.FrameworkUtil;

Filter filter = FrameworkUtil.createFilter(
    String.format(
        "(&(objectClass=%s)%s)",
        ModuleServiceLifecycle.class.getName(),
        ModuleServiceLifecycle.PORTAL_INITIALIZED));

new ServiceTracker<>(bundleContext, filter, null);
```

Acting on lifecycle event services in this way requires service callback handling and some boilerplate code. Using DS components is easier and more elegant, but at least service trackers provide a way to work with lifecycle events outside of DS components.

Related Topics

Service Trackers

Chapter 116

Troubleshooting FAQ

When coding on any platform, you can sometimes run into issues that have no clear resolution. This can be particularly frustrating. If you have issues building, deploying, or running modules, you want to resolve them fast. These frequently asked questions and answers help you troubleshoot problems that arise based on the underlying OSGi platform, and then correct them.

Here are the troubleshooting sections:

- Modules
- Services and Components

Click a question to view the answer.

116.1 Modules

How can I configure dependencies on Liferay Portal artifacts?

```
<p>See <a href="/docs/7-0/tutorials/-/knowledge_base/t/configuring-dependencies">Configuring Dependencies</a>. </p>
```

What are optional package imports and how can I specify them?

```
<p>When developing Liferay Portal modules, you can declare <em>optional</em> package imports. An optional package import is one your module can use if it's
0/tutorials/-/knowledge_base/t/declaring-optional-import-package-requirements">Specifying optional package imports</a> is straightforward. </p>
```

How can I connect to a JNDI data source from my module?

```
<p>Connecting to an application server's JNDI data sources from Liferay Portal's OSGi environment is almost the same as connecting to them from the Java EE
0/tutorials/-/knowledge_base/t/connecting-to-data-sources-using-jndi">use Liferay Portal's class loader to load the application server's JNDI classes</a>. <
```

How can I make sure my module works?

```
<p><a href="/docs/7-0/tutorials/-/knowledge_base/t/testing">The Testing tutorials demonstrate several ways to test Liferay Portal modules</a>:</p>
<ul> <li>Unit testing</li> <li>Integration testing</li> <li>Functional testing</li> </ul>
```

My module has an unresolved requirement. What can I do?

<p>If one of your bundles imports a package that no other bundle in the Liferay OSGi runtime exports, Liferay Portal reports an unresolved requirement:</p>
<pre><code>! could not resolve the bundles: ...
Unresolved requirement: Import-Package: ...
...
Unresolved requirement: Require-Capability ...
</code></pre>
<p>To satisfy the requirement, <a href="/docs/7-0/tutorials/-/knowledge_base/t/resolving-bundle-requirements">find a module that provides the capability, ad

An IllegalContextNameException reports that my bundle's context name does not follow Bundle-SymbolicName syntax. How can I fix the context name?

<p><a href="/docs/7-0/tutorials/-/knowledge_base/t/resolving-bundle-symbolicname-syntax-issues">Adjust the <code>Bundle-SymbolicName</code> to adhere to the

Why aren't my module's JavaScript and CSS changes showing?

<p><a href="/docs/7-0/tutorials/-/knowledge_base/t/why-arent-my-modules-javascript-and-css-changes-showing">Incorrect component properties or stale browser

Why aren't my fragment's JSP overrides showing?

<p><a href="/docs/7-0/tutorials/-/knowledge_base/t/why-arent-jsp-overrides-i-made-using-fragments-showing">Make sure your <code>Fragment-Host</code>'s bundle version is compatible with the host's bundle version</a>. </p>

The application server and database started, but Liferay DXP failed to connect to the database. What happened and how can I fix this?

<p>Liferay Portal initialization can fail while attempting to connect to a database server that isn't ready. <a href="/docs/7-0/tutorials/-/knowledge_base/t/portal-failed-to-initialize-because-the-database-wasnt-ready">Configuring Liferay Portal startup to retry JDBC connections</a

How can I adjust my module's logging?

<p>See <a href="/docs/7-0/tutorials/-/knowledge_base/t/adjusting-module-logging">Adjusting Module Logging</a>. </p>

How can I implement logging in my module or plugin?

<p><a href="/docs/7-0/tutorials/-/knowledge_base/t/implementing-logging">Use Simple Logging Facade for Java (SLF4J) to log messages</a>.</p>

Why did the entity sort order change when I migrated to a new database type?

<p><a href="/docs/7-0/tutorials/-/knowledge_base/t/sort-order-changed-with-a-different-database">Your new database uses a different default query result ord
you should be able to configure a different order</a>.</p>

116.2 Services and Components

How can I detect unresolved OSGi components?

<p>Liferay Portal module components that use Service Builder use Dependency Manager (DM) and most other Liferay Portal module components use Declarative Ser
0/tutorials/-/knowledge_base/t/detecting-unresolved-osgi-components">Gogo shell commands and tools help you find and inspect unsatisfied component reference

What is the safest way to call non-OSGi code that uses OSGi services?

<p>See <a href="/docs/7-0/tutorials/-/knowledge_base/t/calling-non-osgi-code-that-uses-osgi-services">Calling Non-OSGi Code that Uses OSGi Services</a>. </p

How can I use files to configure components?

<p>See <a href="/docs/7-0/tutorials/-/knowledge_base/t/using-files-to-configure-product-modules">Using Files to Configure Module Components</a>. </p>

How can I use OSGi services from Ext Plugins?

<p><a href="/docs/7-0/tutorials/-/knowledge_base/t/using-osgi-services-from-ext-plugins">The registry API lets Ext Plugins use OSGi services </a>. </p>

116.3 Resolving Bundle Requirements

If one of your bundles imports a package that no other bundle in the Liferay OSGi runtime exports, you get a bundle exception. Here's an example exception:

```
! could not resolve the bundles: [com.liferay.messaging.client.command-1.0.0.201707261701 org.osgi.framework.BundleException: Could not resolve module: com.
Unresolved requirement: Import-Package: com.liferay.messaging.client.api; version="[1.0.0,2.0.0)"
-> Export-Package: com.liferay.messaging.client.api; bundle-symbolic-name="com.liferay.messaging.client.provider"; bundle-version="1.0.0.201707261701"; vers
com.liferay.messaging.client.provider [2]
Unresolved requirement: Import-Package: com.liferay.messaging; version="[1.0.0,2.0.0)"
-> Export-Package: com.liferay.messaging; bundle-symbolic-name="com.liferay.messaging.api"; bundle-version="1.0.0"; version="1.0.0"; uses:="com.liferay.petr
com.liferay.messaging.api [12]
Unresolved requirement: Import-Package: com.liferay.petra.io; version="[1.0.0,2.0.0)"
-> Export-Package: com.liferay.petra.io; bundle-symbolic-name="com.liferay.petra.io"; bundle-version="1.0.0"; version="1.0.0"
com.liferay.petra.io [16]
Unresolved requirement: Require-Capability osgi.extender; filter:="(osgi.extender=osgi.serviceloader.processor)"
```

The first part of the message states *could not resolve the bundles*. The rest of the message shows a string of unresolved requirements. Liferay's OSGi Runtime could not resolve one of the bundle's transitive requirements.

The bundle exception message follows this general pattern:

- Module A has an unresolved requirement (package or capability) `aaa.bbb`.
- Module B provides `aaa.bbb` but has an unresolved requirement `ccc.ddd`.
- Module C provides `ccc.ddd` but has an unresolved requirement `eee.fff`.
- etc.
- Module Z provides `www.xxx` but has an unresolved requirement `yyy.zzz`.

The pattern stops at the final requirement no module provides. The last module's dependencies are key to resolving the bundle exception. There are two possible causes:

1. A dependency that satisfies the final requirement might be missing from the build file.

2. A dependency that satisfies the final requirement might not be deployed.

Both cases require deploying a bundle that provides the missing requirement.

The example bundle exception concludes that module `com.liferay.petra.io` requires capability `osgi.extender; filter:="(osgi.extender=osgi.serviceloader.processor)"`. To resolve the requirement, make sure all of `com.liferay.petra.io`'s dependencies are deployed.

First, note the module's dependencies. Here is the `dependencies` section of the `com.liferay.petra.io` module's `build.gradle` file.

```
dependencies {
    provided group: "com.liferay", name: "com.liferay.petra.concurrent", version: "1.0.0"
    provided group: "com.liferay", name: "com.liferay.petra.memory", version: "1.0.0"
    provided group: "org.apache.aries.spifly", name: "org.apache.aries.spifly.dynamic.bundle", version: "1.0.8"
    provided group: "org.slf4j", name: "slf4j-api", version: "1.7.2"
    testCompile group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "default"
}
```

Then use Felix Gogo Shell's `lb command` to verify the dependencies in Liferay's OSGi Runtime:

```
lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|OSGi System Bundle (3.10.100.v20150529-1857)|3.10.100.v20150529-1857
    1|Active     |    1|com.liferay.messaging.client.command (1.0.0.201707261923)|1.0.0.201707261923
    2|Active     |    1|com.liferay.messaging.client.provider (1.0.0.201707261927)|1.0.0.201707261927
    3|Active     |    1|Apache Felix Configuration Admin Service (1.8.8)|1.8.8
    4|Active     |    1|Apache Felix Log Service (1.0.1)|1.0.1
    5|Active     |    1|Apache Felix Declarative Services (2.0.2)|2.0.2
    6|Active     |    1|Meta Type (1.4.100.v20150408-1437)|1.4.100.v20150408-1437
    7|Active     |    1|org.osgi:org.osgi.service.metatype (1.3.0.201505202024)|1.3.0.201505202024
    8|Active     |    1|Apache Felix Gogo Command (0.16.0)|0.16.0
    9|Active     |    1|Apache Felix Gogo Runtime (0.16.2)|0.16.2
   10|Active     |    1|Apache Felix Gogo Runtime (1.0.0)|1.0.0
...
```

Notice the dependency module `org.apache.aries.spifly.dynamic.bundle` is missing from the runtime bundle list. Examining the module's `MANIFEST.MF` file shows it provides the requirement capability `osgi.extender; filter:="(osgi.extender=osgi.serviceloader.processor)"`.

```
Provide-Capability: osgi.extender;osgi.extender="osgi.serviceloader.regi
 strar";version:Version="1.0",osgi.extender;osgi.extender="osgi.servicel
 oader.processor";version:Version="1.0"
```

Deploying this missing bundle `org.apache.aries.spifly.dynamic.bundle` resolves the example's bundle exception.

You can follow these similar steps to resolve your bundle exceptions.

Note: Bndtools's *Resolve* button can resolve bundle dependencies automatically. You specify the bundles your application requires and Bndtools adds transitive dependencies from your configured artifact repository.

Related Topics

Configuring Dependencies

Adding Third Party Libraries to a Module

Felix Gogo Shell

Resolving a Plugins's Dependencies

116.4 Resolving Bundle-SymbolicName Syntax Issues

An OSGi bundle's `Bundle-SymbolicName` and `Bundle-Version` manifest headers uniquely identify it. You can specify a bundle's `Bundle-SymbolicName` in these ways:

1. `Bundle-SymbolicName` header in a bundle's (module's) `bnd.bnd` file.

2. `Bundle-SymbolicName` header in a plugin WAR's `liferay-plugin-package.properties` file.

3. Plugin WAR file name, if the WAR's `liferay-plugin-package.properties` has no `Bundle-SymbolicName` header.

For plugin WARs, specifying the `Bundle-SymbolicName` in the `liferay-plugin-package.properties` file is preferred.

Important: `Bundle-SymbolicName` values must not contain spaces. On bundle deployment, Liferay's OSGi Runtime framework throws an `IllegalContextNameException` if its `Bundle-SymbolicName` has a space.

For example, if you deploy a plugin WAR that has no `Bundle-SymbolicName` header in its `liferay-plugin-package.properties`, the WAB Generator uses the WAR's name as the WAB's `Bundle-SymbolicName`. If the

WAR's name has a space in it (e.g., `space-program-theme v1.war`) an `IllegalContextNameException` occurs on deployment.

```
org.apache.catalina.core.ApplicationContext.log The context name 'space-program-theme v1' does not follow Bundle-SymbolicName syntax.
org.eclipse.equinox.http.servlet.internal.error.IllegalContextNameException: The context name 'space-program-theme v1' does not follow Bundle-
SymbolicName syntax.
```

To avoid using spaces and to follow naming best practices, you can use a reverse-domain name in your `Bundle-SymbolicName`.

Here's an example domain name and reverse domain name:

Module domain: `troubleshooting.liferay.com`

Module reverse-domain: `com.liferay.troubleshooting`

However you set your a `Bundle-SymbolicName`, refrain from using spaces.

Related Topics

Using the WAB Generator

116.5 Resolving ClassNotFoundException and NoClassDefFoundError in OSGi Bundles

Understanding a `ClassNotFoundException` or `NoClassDefFoundError` in non-OSGi environments is straightforward.

- `ClassNotFoundException`: thrown when looking up a class that isn't on the classpath or using an invalid name to look up a class that isn't on the runtime classpath.
- `NoClassDefFoundError`: occurs when a compiled class references another class that isn't on the runtime classpath.

In OSGi environments, however, there are additional cases where a `ClassNotFoundException` or `NoClassDefFoundError` can occur. Here are four:

1. The missing class belongs to a module dependency that's an OSGi module.
2. The missing class belongs to a module dependency that's *not* an OSGi module.
3. The missing class belongs to a global library, either at the Liferay DXP webapp scope or the application server scope.
4. The missing class belongs to a Java runtime package.

This tutorial explains how to handle each case.

Case 1: The Missing Class Belongs to an OSGi Module

In this case, there are two possible causes:

1. **The module doesn't import the class's package:** For a module (or WAB) to consume another module's exported class, the consuming module must import the exported package that contains the class. To do this, you add an `Import-Package` header in the consuming module's `bnd.bnd` file. If the consuming module tries to access the class without importing it, a `ClassNotFoundException` or `NoClassDefFoundError` occurs.

In the consuming module, make sure you import the correct package. First check the package name. If the package import is correct but you still get the exception or error, the class might no longer exist in the package.

2. **The class no longer exists in the imported package:** In OSGi runtime environments, modules can change and come and go. If you reference another module's class that its developer removed, a `NoClassDefFoundError` or `ClassNotFoundException` occurs. Semantic Versioning guards against this scenario: removing a class from an exported package constitutes a new major version for that package. Neglecting to increment the package's major version breaks dependent modules.

   For example, say a module that consumes the class `com.foo.Bar` specifies the package import `com.foo;version=[1.0.0, 2.0.0)`. The module uses `com.foo` versions from `1.0.0` up to (but not including) `2.0.0`. The first part of the version number (the `1` in `1.0.0`) represents the *major* version. The consuming module doesn't expect any *major* breaking changes, like a class removal. Removing `com.foo.Bar` from `com.foo` without incrementing the package to a new major version (e.g., `2.0.0`) causes a `ClassNotFoundException` or `NoClassDefFoundError` when other modules look up or reference that class.

   You have these options since the class no longer exists in the package:

   - Adapt to the new API. To learn how to do this, read the package's/module's Javadoc, release notes, and or formal documentation. You can also ask the author, or search forums.

   - Revert to the module version you used previously. Deployed module versions reside in `[Liferay_Home]/osgi/`. For details, see Backing up Liferay Installations.

   Do what you think is best to get your module working properly.

Now you know how to resolve common situations involving `ClassNotFoundException` or `NoClassDefFoundError`. For additional information on `NoClassDefFoundError`, see OSGi Enroute's article What is NoClassDef-FoundError?.

Case 2: The Missing Class Doesn't Belong to an OSGi Module

In this case, you have two options:

1. Convert the dependency into an OSGi module so it can export the missing class. Converting a non-OSGi `JAR` file dependency into an OSGi module that you can deploy alongside your application is the ideal solution, so it should be your first choice.

2. Embed the dependency in your module by embedding the dependency `JAR` file's packages as private packages in your module. If you want to embed a non-OSGi `JAR` file in your application, see the tutorial Adding Third Party Libraries to a Module.

Case 3: The Missing Class Belongs to a Global Library

In this case, you can configure Liferay DXP so the OSGi system module exports the missing class's package. Then your module can import it. You should **NOT**, however, undertake this lightly. If Liferay intended to make a global library available for use by developers, the system module would already export this library! Still, if you must access a global library that's not currently exported and can't think of **any** other solution, you can consider adding the required package for export by the system module. There are two ways to do this:

1. In your portal-ext.properties file, use the property module.framework.system.packages.extra to specify the packages to export.

2. If the package you need is from a Liferay DXP JAR, you might be able to add the module to the list of exported packages in [LIFERAY_HOME]/osgi/core/com.liferay.portal.bootstrap.jar's META-INF/system.packages.extra.bnd file. Try this option only if the first option doesn't work.

If the package you need is from a Liferay DXP module, (i.e., it's **NOT** from a global library), you can add the package to that module's bnd.bnd exports. You should **NOT**, however, undertake this lightly. The package would already be be exported if Liferay intended for it to be available.

Case 4: The Missing Class Belongs to a Java Runtime Package

rt.jar (the JRE library) has non-public packages. If your module imports one of them, configure Liferay DXP's system bundle to export the package to the module framework.

1. Add the current module.framework.system.packages.extra property setting to a [LIFERAY_HOME]/portal-ext.properties file. Your server's current setting is in the Liferay DXP web application's /WEB-INF/lib/portal-impl.jar/portal.properties file.

2. In your portal-ext.properties file, append the required Java runtime package to the end of the module.framework.system.packages.extra property's package list.

3. Restart your server.

The package requirement resolves.

Related Topics

Backing up Liferay Installations
Adding Third Party Libraries to a Module
Bundle Classloading Flow

116.6 Identifying Liferay Artifact Versions for Dependencies

When you're developing a Liferay DXP application, it's often necessary to use various Liferay DXP APIs or tools. For example, you might want to create a Service Builder application or use Liferay DXP's message bus or asset framework. How can you determine which versions of Liferay DXP artifacts (modules, apps, etc.) your application's modules need to specify as dependencies? To learn how to find and configure Liferay DXP dependencies, please refer to the following tutorial:

- Configuring Dependencies

Related Topics

Finding Extension Points

116.7 Connecting to JNDI Data Sources

Connecting to an application server's JNDI data sources from Liferay DXP's OSGi environment is almost the same as connecting to them from the Java EE environment. In an OSGi environment, the only difference is that you must use Liferay DXP's class loader to load the application server's JNDI classes. The following code demonstrates this.

```
Thread thread = Thread.currentThread();

// Get the thread's class loader. You'll reinstate it after using
// the data source you look up using JNDI

ClassLoader origLoader = thread.getContextClassLoader();

// Set Liferay's class loader on the thread

thread.setContextClassLoader(PortalClassLoaderUtil.getClassLoader());

try {

        // Look up the data source and connect to it

        InitialContext ctx = new InitialContext();
        DataSource datasource = (DataSource)
            ctx.lookup("java:comp/env/jdbc/TestDB");

        Connection connection = datasource.getConnection();
        Statement statement = connection.createStatement();

        // Execute SQL statements here ...

        connection.close();
}
catch (NamingException ne) {

    ne.printStackTrace();
}
catch (SQLException sqle) {

    sqle.printStackTrace();
}
finally {
       // Switch back to the original context class loader

       thread.setContextClassLoader(origLoader);
}
```

The example code sets Liferay DXP's classloader on the thread to access the JNDI API. After working with the data source, the code reinstates the thread's original classloader.

Here are the class imports for the code above:

```
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import com.liferay.portal.kernel.util.PortalClassLoaderUtil;
```

Your applications can use similar code to access a data source. Make sure to substitute `jdbc/TestDB` with your data source name.

Note: An OSGi bundle's attempt to connect to a JNDI data source without using Liferay DXP's classloader results in a `java.lang.ClassNotFoundException`. For example, here's an exception from attempting to use Apache Tomcat's JNDI API without using Liferay DXP's classloader:

```
javax.naming.NoInitialContextException: Cannot instantiate class:
org.apache.naming.java.javaURLContextFactory [Root exception is
java.lang.ClassNotFoundException:
org.apache.naming.java.javaURLContextFactory]
```

An easier way to work with databases is to connect to them using Service Builder.

Related Topics

Connecting Service Builder to External Data Sources

116.8 Adjusting Module Logging

Liferay DXP uses Log4j logging services. Here are the ways to configure logging for module classes and class hierarchies.

- Liferay DXP's UI
- Configure Log4j for multiple modules in a `[anyModule]/src/main/resources/META-INF/module-log4j.xml` file.
- Configure Log4j for a specific module in a `[Liferay Home]/osgi/log4j/[symbolicNameOfBundle]-log4j-ext.xml` file.
- Configure Log4j for an OSGi fragment host module in a `/META-INF/module-log4j-ext.xml` file

Here's an example Log4j XML configuration:

```
<?xml version="1.0"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <category name="org.foo">
        <priority value="DEBUG" />
    </category>
</log4j:configuration>
```

Use `category` elements to specify each class or class hierarchy to log messages for. Set the `name` attribute to that class name or root package. The example category sets logging for the class hierarchy starting at package `org.foo`. Log messages at or above the `DEBUG` log level are printed for classes in `org.foo` and classes in packages starting with `org.foo`.

Set each category's `priority` element to the log level (priority) you want.

- ALL
- DEBUG
- ERROR
- FATAL
- INFO
- OFF
- TRACE
- WARN

The log messages are printed to Liferay log files in `[Liferay_Home]/logs`.

You can see examples of module logging in several Liferay sample projects. For example, the action-command-portlet, document-action, and service-builder/jdbc samples (among others) leverage module logging.

Note: If the log level configuration isn't appearing (e.g., you set the log level to `ERROR` but you're still getting `WARN` messages), make sure the log configuration file name prefix matches the module's symbolic name. If you have Bnd installed, output from command `bnd print [path-to-bundle]` includes the module's symbolic name (Here are instructions for installing Bnd for the command line).

Note: A Log4j configuration's appenders control log file location, naming, and rotation. To override advanced Log4j options such as Liferay DXP's log appenders, use an Ext plugin.

That's it for module log configuration. You're all set to print the information you want.

116.9 Implementing Logging

7.0 uses the Log4j logging framework, but a different one may eventually replace it. It's a best practice to use Simple Logging Facade for Java (SLF4J) to log messages in your modules and traditional plugins. SLF4J is already integrated into Liferay DXP, so you can focus on logging messages.

Here's how to use SLF4J to log messages in a class:

1. Add a private static SLF4J `Logger` field.

   ```
   private static Logger _logger;
   ```

2. Instantiate the logger.

   ```
   _logger = LoggerFactory.getLogger(this.getClass().getName());
   ```

3. Throughout your class, log messages where noteworthy things happen.

   For example,

   ```
   _logger.debug("...");
   _logger.warn("...");
   _logger.error("...");
   ...
   ```

   Use `Logger` methods appropriate for each message:

   - `debug`: Event and application information helpful for debugging.
   - `error`: Normal errors. This is the least verbose message level.
   - `info`: High level events.
   - `trace`: Provides more information than debug. This is the most verbose message level.
   - `warn`: Information that might, but does not necessarily, indicate a problem.

Log verbosity should correlate with the log level set for the class or package. Make sure you provide additional information at log levels expected to be more verbose, such as `info` and `debug`.

You're all set to add logging to your modules and traditional plugins.

116.10 Declaring Optional Import Package Requirements

When developing Liferay DXP modules, you can declare *optional* dependencies. An optional dependency is one your module can use if it's available, but can still function without it.

Important: Try to avoid optional dependencies. The best module designs rely on normal dependencies. If an optional dependency seems desirable, your module may be trying to provide distinct types of functionality. In such a situation, it's best to split it into multiple modules that provide smaller, more focused functionality.

If you decide that your module requires an optional dependency, follow these steps to add it:

1. Declare the package that your module optionally requires as an optional dependency in your module's `bnd.bnd` file:

   ```
   Import-Package: com.liferay.demo.foo;resolution:="optional"
   ```

   Note that you can use either an optional or dynamic import. The differences are explained in this blog post.

2. Create a component to use the optional package:

   ```
   import com.liferay.demo.foo.Foo; // A class from the optional package

   @Component(
       enabled = false // instruct declarative services to ignore this component by default
   )
   public class OptionalPackageConsumer implements Foo {...}
   ```

3. Create a second component to act as a controller of the first component. The second component checks the classloader for the optional class on the classpath. It handles both cases appropriately. If it's not there, this means you must catch any `ClassNotFoundException`. For example:

   ```
   @Component
   public class OptionalPackageConsumerStarter {
       @Activate
       void activate(ComponentContext componentContext) {
           try {
               Class.forName(com.liferay.demo.foo.Foo.class.getName());

               componentContext.enableComponent(OptionalPackageConsumer.class.getName());
           }
           catch (Throwable t) {
               _log.warn("Could not find {}", t.getMessage()); // Could use _log.info instead
           }
       }
   }
   ```

If the classloader check in the controller component is successful, the client component is enabled. This check is automatically performed whenever there are any wiring changes to the module containing these components (Declarative Services components are always restarted when there are wiring changes).

As above, if you install the module when the optional dependency is missing from Liferay DXP's OSGi runtime, your controller component catches a `ClassNotFoundException` and logs a warning or info message (or takes whatever other action you implement to handle this case). If you install the optional dependency, refreshing your module triggers the OSGi bundle lifecycle events that trigger your controller's `activate` method and the check for the optional dependency. Since the dependency exists, your client component uses it.

Note that you can refresh a bundle from Liferay DXP's Gogo shell with this command:

```
equinox:refresh [bundle ID]
```

For more information about optional dependencies, see OSGi Enroute's documentation.

Related Topics

Configuring Dependencies

116.11 Why Aren't my Module's JavaScript and CSS Changes Showing?

To determine why JavaScript and CSS updates to your module aren't having an effect in your browser, perform these checks:

1. If you're developing a portlet module, check that your portlet class has the correct properties specified in its `@Component` annotation:

    - Make sure the resources referred to by the properties of your portlet class's `@Component` annotation exist in the correct location in your module project.
    - Make sure that you're using a portlet CSS wrapper class to prevent potential CSS ID and class name conflicts with other applications on the page.

    For example, consider this sample portlet class:

    ```
    @Component(
        immediate = true,
        property = {
            "com.liferay.portlet.css-class-wrapper=example-portlet",
            "com.liferay.portlet.display-category=category.sample",
            "com.liferay.portlet.instanceable=true",
            "com.liferay.portlet.header-portlet-css=/css/main.css",
            "com.liferay.portlet.header-portlet-javascript=/css/main.js",
            "javax.portlet.display-name=Example Portlet",
            "javax.portlet.init-param.template-path=/",
            "javax.portlet.init-param.view-template=/view.jsp",
            "javax.portlet.name=" + ExamplePortletKeys.TicTacToe,
            "javax.portlet.resource-bundle=content.Language",
            "javax.portlet.security-role-ref=power-user,user"
        },
        service = Portlet.class
    )
    public class ExamplePortlet extends MVCPortlet {

    }
    ```

    As described in the first item above, the portlet's CSS file is specified by the property `com.liferay.portlet.header-portlet-css`. Paths specified as values of this property are relative to the module's `src/main/resources/META-INF/resources` folder. So if you specify a value of `css/main.css`, the actual path to the CSS file in the module is `src/main/resources/META-INF/resources/css/main.css`. The path to your portlet's JavaScript file is specified by the property `com.liferay.portlet.header-portlet-javascript`. Values for this property work the same as the values for the CSS property.

Also note that the property `com.liferay.portlet.css-class-wrapper` specifies the CSS class wrapper `example-portlet`. Thus, you should use subclasses of `example-portlet` in your portlet's actual CSS file. For example, in `main.css` you'd do this to change the background to green:

```
.example-portlet {
    .greenBackground {
        background-color: green;
    }

    ... (further properties)

}
```

In other words, to avoid CSS class and ID name conflicts, all the CSS properties you specify must be subclasses of the class specified via the `com.liferay.portlet.css-class-wrapper` property. Liferay DXP wraps your portlet's HTML content with a `<div>`. The class specified by `com.liferay.portlet.css-class-wrapper` (`example-portlet`, in this example) has been applied to this `<div>`.

2. Check that caching isn't preventing JS and CSS updates to your module from appearing in your browser:

    - Clear your browser's cache.
    - During development, enable developer mode to turn off Liferay DXP's resource caching. Click here to learn how to enable Liferay DXP's developer mode.

Related Topics

Using Developer Mode with Themes

116.12 Why Aren't JSP overrides I Made Using Fragments Showing?

The fragment module must specify the exact version of the host module. A Liferay DXP upgrade might have changed some JSPs in the host module, prompting a version update. If this occurs, check that your JSP customizations are compatible with the updated host JSPs and then update your fragment module's targeted version to match the host module.

For example, this `bnd.bnd` file from a fragment module uses `Fragment-Host` to specify the host module and host module version:

```
Bundle-Name: custom-login-jsp
Bundle-SymbolicName: custom.login.jsp
Bundle-Version: 1.0.0
Fragment-Host: com.liferay.login.web;bundle-version="1.1.18"
```

For information on finding the versions of your deployed modules, click here.
For more information on overriding JSPs, click here.

Related Topics

Overriding JSPs
Configuring Dependencies

116.13 Detecting Unresolved OSGi Components

Liferay DXP includes Gogo shell commands that come in handy when trying to diagnose a problem due to an unresolved OSGi component. The specific tools to use depend on the component framework of the unresolved component. Most Liferay DXP components are developed using Declarative Services (DS), also known as SCR (Service Component Runtime). An exception to this is Liferay DXP's Service Builder services, which are provided as Dependency Manager (DM) components. Both Declarative Services and Dependency Manager are Apache Felix projects.

The troubleshooting instructions are divided into these sections:

- Declarative Services Components
  - Declarative Services Unsatisfied Component Scanner
  - ds:unsatisfied Command
- Service Builder Components
  - Unavailable Component Scanner
  - ServiceProxyFactory

Declarative Services Components

Start with DS, since most Liferay DXP components, apart from Service Builder components, are DS components. Suppose one of your bundle's components has an unsatisfied service reference. How can you detect this? Two ways:

- You can enable a Declarative Services Unsatisfied Component Scanner to report unsatisfied references automatically or
- You can use the Gogo shell command `ds:unsatisfied` to check for them manually.

Declarative Services Unsatisfied Component Scanner

Note: The Declarative Services Unsatisfied Component Scanner appears in DXP Digital Enterprise Fix 7.0 Pack 31 and Liferay CE Portal 7.0 GA5.

Here are the steps for enabling the unsatisfied component scanner:

1. Create a file `com.liferay.portal.osgi.debug.declarative.service.internal.configuration.UnsatisfiedComponentScann`
2. Add the following file content:

   ```
   unsatisfiedComponentScanningInterval=5
   ```

3. Copy the file into `[LIFERAY_HOME]/osgi/configs`.

The scanner detects and logs unsatisfied service component references. The log message includes the class that contains the reference, the bundle's ID, and the referenced component type.

Here's an example scanner message:

```
11:18:28,881 WARN  [Declarative Service Unsatisfied Component Scanner][UnsatisfiedComponentScanner:91]
Bundle {id: 631, name: com.liferay.blogs.web, version: 2.0.0}
    Declarative Service {id: 3333, name: com.liferay.blogs.web.internal.portlet.action.EditEntryMVCRenderCommand, unsatisfied references:
        {name: ItemSelectorHelper, target: null}
    }
```

The message above warns that service component `com.liferay.blogs.web.internal.portlet.action.EditEntryMVCRenderCo` has an unsatisfied reference to a component of type `ItemSelectorHelper`. The referencing component's ID (SCR ID) is 3333 and the component belongs to bundle 631.

ds:unsatisfied Command

Another way to detect unsatisfied component references is to invoke the Gogo shell command `ds:unsatisfied`.

- `ds:unsatisfied` shows all unsatisfied services
- `ds:unsatisfied [BUNDLE_ID]` shows the bundle's unsatisfied services

Note: The Gogo shell command `ds:unsatisfied` appears in DXP Digital Enterprise Fix 7.0 Pack 31 and Liferay CE Portal 7.0 GA5.

To view more detailed information about the component with the unsatisfied reference, use the command `scr:info [component ID]`. For example, the following command does this for the component ID `1701`:

```
g! scr:info 1701
*** Bundle: org.foo.bar.command (507)
Component Description:
    Name: org.foo.bar.command
    Implementation Class: org.foo.bar.command.FooBarCommand
    Default State: enabled
    Activation: delayed
    Configuration Policy: optional
    Activate Method: activate
    Deactivate Method: deactivate
    Modified Method: -
    Configuration Pid: [org.foo.bar.command]
    Services:
        org.foo.bar.command.DuckQuackCommand
    Service Scope: singleton
    Reference: Duck
        Interface Name: org.foo.bar.api.Foo
        Cardinality: 1..1
        Policy: static
        Policy option: reluctant
        Reference Scope: bundle
    Component Description Properties:
        osgi.command.function = foo
        osgi.command.scope = bar
    Component Configuration:
        ComponentId: 1701
        State: unsatisfied reference
        UnsatisfiedReference: Foo
        Target: null
        (no target services)
        Component Configuration Properties:
        component.id = 1701
        component.name = org.foo.bar.command
        osgi.command.function = foo
        osgi.command.scope = bar
```

In the `Component Configuration` section, `UnsatisfiedReference` lists the unsatisfied reference's type. This bundle's component isn't working because it's missing a `Foo` service. Now you can focus on why `Foo` is unavailable. The solution may be as simple as starting or deploying a bundle that provides the `Foo` service.

Service Builder Components

Liferay DXP's Service Builder modules are implemented using Spring. Liferay DXP uses the Apache Felix Dependency Manager to manage Service Builder modules' OSGi components via the Portal Spring Extender module.

When developing a Liferay Service Builder application, you might encounter a situation where your application has an unresolved Spring-related OSGi component. This could occur, for example, if you update your application's database schema but forget to trigger an upgrade (for information on creating database upgrade processes for your Liferay DXP applications, see the tutorial Creating an Upgrade Process for Your App).

These features detect unresolved Service Builder related components.

- Unavailable Component Scanner
- ServiceProxyFactory

Unavailable Component Scanner

The Liferay Foundation application suite's Unavailable Component Scanner reports missing components in modules that use Service Builder. Here are the steps for enabling the scanner:

1. Create configuration file `com.liferay.portal.osgi.debug.spring.extender.internal.configuration.UnavailableCompon`
2. In the configuration file, set the time interval (in seconds) between scans:

   ```
   unavailableComponentScanningInterval=5
   ```

3. Copy the file into folder `[LIFERAY_HOME]/osgi/configs`.

The scanner reports Spring extender dependency manager component status on the set interval. If all components are registered, the scanner sends a confirmation message.

```
11:10:53,817 INFO  [Spring Extender Unavailable Component Scanner][UnavailableComponentScanner:166] All Spring extender dependency manager components are re
```

If a component is unavailable, it reports an error like this one:

```
11:13:08,851 WARN  [Spring Extender Unavailable Component Scanner][UnavailableComponentScanner:173] Found unavailable component in bundle com.liferay.screen
Component ComponentImpl[null com.liferay.portal.spring.extender.internal.context.ModuleApplicationContextRegistrator@1541eee] is unavailable due to missing
```

Component unavailability, such as what's reported above, can occur when declarative services components and Service Builder components are published and used in the same module. We recommend you publish DS components and Service Builder components in separate modules.

Note: The Spring Extender Unavailable Component Scanner appears in DXP Digital Enterprise Fix 7.0 Pack 24 and Liferay CE Portal 7.0 GA5.

ServiceProxyFactory

Liferay DXP's logs report unresolved Service Builder components too. For example, Liferay DXP logs an error when a Service Proxy Factory can't create a new instance of a Service Builder based entity because a service is unresolved.

Note: The Service Proxy Factory timeout logs appear in DXP Digital Enterprise Fix 7.0 Pack 32 and Liferay CE Portal 7.0 GA5.

The following code demonstrates using a `ServiceProxyFactory` class to create a new entity instance:

```
private static volatile MessageBus _messageBus =
    ServiceProxyFactory.newServiceTrackedInstance(
        MessageBus.class, MessageBusUtil.class, "_messageBus", true);
```

This message alerts you to the unavailable service:

```
11:07:35,139 ERROR [localhost-startStop-1][ServiceProxyFactory:265] Service "com.liferay.portal.kernel.messaging.sender.SingleDestinationMessageSenderFactor
```

Based on the message above, there's no bundle providing the service `com.liferay.portal.kernel.messaging.sender.SingleI`

To check your Service Builder modules for unresolved Spring components, you can use the Dependency Manager's `dm` Gogo shell command, which is explained here:

- Dependency Manager - Leveraging the shell

For example, to get diagnostic information about Service Builder components, use the `dependencymanager:dm` command. This command lists all Service Builder components, their required services, and whether each required service is available.

Related Topics

Calling Non-OSGi Code that Uses OSGi Services

Felix Gogo Shell

OSGi Basics For Liferay Development

116.14 Using Files to Configure Module Components

Liferay DXP uses Felix File Install to monitor file system folders for new/updated configuration files, and the Felix OSGi implementation of Configuration Admin to let you use files to configure module service components.

To learn how to work with configuration files, first review Understanding System Configuration Files.

Configuration File Formats

There are two different configuration file formats:

- `.cfg`: An older, simple format that only supports string values as properties.
- `.config`: A format that supports strings, type information, and other non-string values in its properties.

Although Liferay DXP supports both formats, we recommend `.config` files for their flexibility. Since `.cfg` file lacks type information, if you want to store anything but a String, you'll need a properties utility class that casts the Strings to their proper types (and you must carefully document properties that aren't Strings). Since `.config` files can include type information, using them eliminates this need. The articles below provide a detailed explanation of these file formats, including their syntax:

- Understanding System Configuration Files
- Configuration file (`.config`) syntax
- Properties file(`.cfg`) syntax

Naming Configuration Files

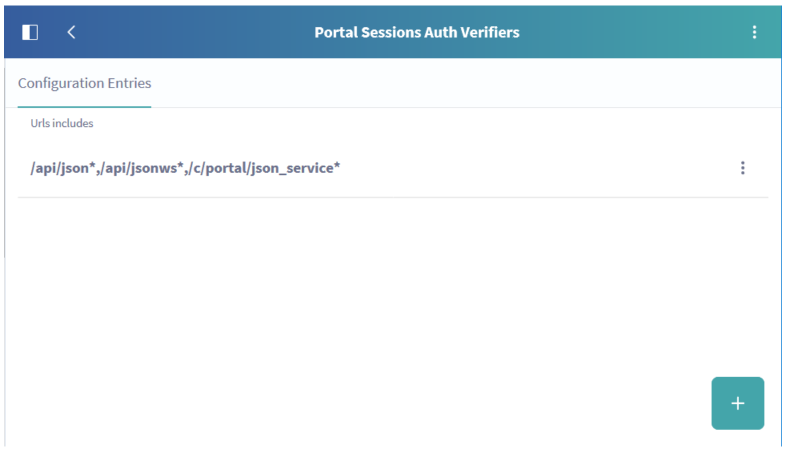
Before you create a configuration file, you should determine whether multiple instances of the component can be created, or if the component is intended to be a singleton. Follow these steps to make that determination:

1. Deploy the component's module if you haven't done so already.

2. In Liferay DXP's UI, go to *Control Panel* → *Configuration* → *System Settings*.

3. Find the component's settings by searching or browsing for the component.

4. If the component's settings page has a section called *Configuration Entries*, you can create multiple instances of the component configured however you like. Otherwise, you should treat the component as a singleton.



/api/json*,/api/jsonws*,/c/portal/json_service*

Figure 116.1: You can create multiple instances of components whose System Settings page has a *Configuration Entries* section.

All configuration file names must start with the component's PID (PID stands for *persistent identity*) and end with `.config` or `.cfg`.

For example, this class uses Declarative Services to define a component:

```
package com;
@Component
class Foo {}
```

The component's PID is `com.Foo`. All the component's configuration files must start with the PID `com.Foo`.

For each non-singleton component instance you want to create or update with a configuration, you must use a uniquely named configuration file that starts with the component's PID and ends with `.config` or `.cfg`. Creating configurations for multiple component instances requires that the configuration files use different *subnames*. A subname is the part of a configuration file name after the PID and before the suffix `.config` or `.cfg`. Here's the configuration file name pattern for non-singleton components:

- `[PID]-[subname1].config`
- `[PID]-[subname2].config`
- etc.

For example, you could configure two different instances of the component `com.Foo` by using configuration files with these names:

- `com.Foo-one.config`
- `com.Foo-two.config`

Each configuration file creates and/or updates an instance of the component that matches the PID. The subname is arbitrary–it doesn't have to match a specific component instance. This means you can use whatever subname you like. For example, these configuration files are just as valid as the two above:

- `com.Foo-puppies.config`
- `com.Foo-kitties.config`

Using the subname `default`, however, is Liferay DXP's convention for configuring a component's first instance. The file name pattern is therefore:

```
[PID]-default.config
```

A singleton component's configuration file must also start with `[PID]` and end with `.config` or `.cfg`. Here's the common pattern used for singleton component configuration file names:

```
[PID].config
```

When you're done creating a configuration file, you can deploy it.

Resolving Configuration File Deployment Failures

The following `IOException` hints that the configuration file has a syntax issue:

```
Failed to install artifact: [path to .config or .cfg file]
java.io.IOException: Unexpected token 78; expected: 61 (line=0, pos=107)
```

To resolve this, fix the configuration file's syntax.

Great! Now you know how to configure module components using configuration files.

Related Articles

Understanding System Configuration Files

116.15 Calling Non-OSGi Code that Uses OSGi Services

Liferay DXP's static utility functions (e.g., `UserServiceUtil`, `CompanyServiceUtil`, `GroupServiceUtil`, etc.) are examples of non-OSGi code that use OSGi services.

Note that it's safer to track and use Liferay DXP's OSGi services directly with Liferay DXP's Registry API than to invoke Liferay DXP's similar static utility functions. For example, you can't call an OSGi service unless all its dependencies are satisfied: the container won't enable the service. If you invoke Liferay DXP's static utility functions, you might invoke them prematurely (e.g., before OSGi bundle activation and application startup events). You could work around this by identifying all of the implied OSGi service dependencies and making sure they are satisfied before invocation, but then you're not only duplicating the container's more robust functionality for this, you're also creating a bigger surface for bugs. Avoid this mess by using Liferay DXP's Registry API to track the services you want. This way, you let OSGi make sure a service's dependencies are satisfied before invoking that service. For example, use Liferay DXP's OSGi service `UserService` instead of `UserServiceUtil`, which in turn uses the OSGi service `UserService`. Click here to see an example of this.

Remember that you can check the state of Liferay DXP's services in the Gogo shell. If you're running Liferay DXP locally, use the command `telnet localhost 11311` to connect to the Gogo shell. Once connected, the `scr:list` command shows all Declarative Services components, including inactive ones from unsatisfied dependencies. To find unsatisfied dependencies for Service Builder services, use the Dependency Manager's `dependencymanager:dm wtf` command. Note that these commands only show components that haven't been activated because of unsatisfied dependencies–they don't show pure service trackers that are waiting for a service because of unsatisfied dependencies.

Related Topics

Using OSGi Services from EXT Plugins

Detecting Unresolved OSGi Components

Felix Gogo Shell

OSGi Basics For Liferay Development

116.16 Patching DXP Source Code

Auto mechanics, enthusiasts, and prospective owners ask about cars, "What's under the hood?" Here are common reasons for asking that question:

- Concern about an issue
- Curiosity about the car's capability and inner-workings
- Desire to improve or customize the car

You might have similar reasons for asking "What's under the *DXP's* hood?" And since you get access to DXP Digital Enterprise (DXP)'s source code, you can attach a debugger see it in action! Setting up the code locally is your ticket to exploring DXP, investigating issues, and making improvements and customizations.

Here's how:

1. Download DXP, the DXP source code, and patches

2. Prepare DXP

3. Patch the DXP source code

Step 1: Download DXP, the DXP source code, and patches

1. Download a DXP bundle (or DXP JARs) and the DXP source code for the version you're using from the customer portal.

2. Download fix packs and their source code from here. Fix pack ZIP files that end in `-src.zip` contain a fix pack and source code.

Next you'll install and configure DXP. DXP's patching tool lets you install fix packs and fix pack source code. If you have a patched DXP installation already and want to use it, skip to the section *Patching the DXP Source Code*.

Step 2: Prepare DXP

Preparing DXP locally involves installing, configuring, and patching DXP.

Install and Configure DXP

Here's how to install and configure DXP:

1. Install and Deploy DXP locally.

2. Start DXP.

3. Configure DXP to use your database.

4. Stop DXP.

It's time apply the DXP patches you want.

Patch DXP

Here's how to patch DXP:

1. Copy all the patch ZIP files you want to `[LIFERAY_HOME]/patching-tool/patches`. The `-src.zip` fix pack files are best to use because they contain both the fix pack binaries and source code.

2. Open a terminal window to `[LIFERAY_HOME]/patching-tool`.

3. Run the command `patching-tool.sh auto-discovery` to generate the default patching profile called `default.properties`. Make sure the profile's properties refer to your DXP installation. See the patching tool documentation for more details.

   Here's an example profile:

   ```
   patching.mode=binary
   war.path=../tomcat-8.0.32/webapps/ROOT/
   global.lib.path=../tomcat-8.0.32/lib/ext/
   liferay.home=../
   ```

4. To list all the patch files available in `[LIFERAY_HOME]/patching-tool/patches`, execute the following command:

   ```
   patching-tool.sh info
   ```

5. Execute this command to install the patches:

   ```
   patching-tool.sh install
   ```

   The patching tool documentation describes additional steps that might apply to your situation, such as creating database indexes.

It's time to prepare the DXP source code and patch source code.

Step 3: Patch the DXP Source Code

Unzip the DXP source code to where you want to work with it.

Next you'll create a patching tool profile for your DXP source code.

Create a Patching Tool Profile for the Source Code

Here's how to create a profile that refers to your source code.

1. Execute the following command to create a profile. Replace `[profile]` with a name for your profile.

   ```
   patching-tool.sh auto-discovery [profile]
   ```

2. In the profile properties file generated in the previous step, set the `patching.mode` property to `source` and set the `source.path` property to your source code path:

   ```
       patching.mode=source
       source.path=[DXP source code path]
   ```

It's time to apply the DXP patches you downloaded earlier.

Patch the Source Code

DXP's patching tool is safe and easy to use. Beyond installing patches, it has these functions:

- List a patch's code changes
- List the issues (LPS/LPE tickets) a patch fixes
- Revert a patch

See the following patching tool documentation for more details:

- Comparing Patch Levels
- Removing or Reverting Patches

In addition to using the patching tool to manage DXP source code, you can optionally manage it in a version control system such as Git.

Here are commands for setting up the DXP source code in Git:

```
cd [path to source code root folder]
git init
git add .
git commit -a
```

Here's are the command descriptions:

- `init` creates a Git repository for the current folder (i.e., the root folder) and all its contents.
- `add` stages the root folder and its contents.
- `commit` checks in the staged files.

You can commit any code changes (e.g., DXP patches) to your Git repository.

The patching tool installs all patches and patch source code from the ZIP files it finds in `[LIFERAY_HOME]/patching-tool/patches`. All your patches must be in the `patches` folder for the patching tool to apply them.

1. Copy all the patch source ZIP files to `[LIFERAY_HOME]/patching-tool/patches` if you haven't already copied them there.

2. Execute the `info` command to make sure it lists your patches. If a patch isn't listed, copy its ZIP file into the `patches` folder. Replace `[profile]` with your DXP source code profile name:

```
patching-tool.sh [profile] info
```

3. Apply the patches by executing the `install` command on your profile:

```
patching-tool.sh [profile] install
```

Your DXP installation and source code is patched and ready to debug!

Attach your favorite debugger to your DXP instance and start the server. See your debugger's documentation for configuration details.

Congratulations! You're free to explore DXP inside and out!

Related Topics (id=related-topics)

Troubleshooting

Liferay @ide@

116.17 Liferay DXP Failed to Initialize Because the Database Wasn't Ready

If you start your database server and application server at the same time, Liferay DXP might try connecting to the data source before the database is ready. By default, Liferay DXP doesn't retry connecting to the database; it just fails. Now Liferay DXP provides a way to avoid this situation: database connection retries.

1. Create a `portal-ext.properties` file.

2. Set the property `retry.jdbc.on.startup.max.retries` equal to the number of times to retry connecting to the data source.

3. Set property `retry.jdbc.on.startup.delay` equal to the number of seconds to wait before retrying connection.

If at first the connection doesn't succeed, Liferay DXP uses the retry settings to try again.

Related Topics

Connecting to JNDI Data Sources

116.18 Using OSGi Services from EXT Plugins

Using OSGi services from an Ext plugin is done the same way that Liferay DXP's core uses OSGi services: via the `com.liferay.registry` API provided by the `registry-api` bundle. All usages of this API in Liferay DXP's core can serve as examples. Here's a very simple example:

```
Registry registry = RegistryUtil.getRegistry();
UserService userService = registry.getService(UserService.class);
If (userService ≠ null) {
    User user = userService.getCurrentUser();
    System.out.println("Current user is " + user.getFirstName() + StringPool.BLANK +
        user.getLastName());
}
```

Remember that OSGi services can come and go at any time. Liferay DXP services, including `UserService`, aren't an exception to this rule. Although it's unlikely that `UserService` becomes unavailable, you must still account for that possibility. Liferay DXP's registry API provides `ServiceReference` and `ServiceTracker`, which you can use to simplify dealing with OSGi services. If you're familiar with OSGi development, you've heard of these classes because the OSGi framework provides them. Liferay DXP's versions of these classes wrap the OSGi ones, so you can use them the same way.

Here's a smarter version of the above example. Using service trackers takes away much of the pain of having to deal with services that can appear and disappear dynamically:

```
Registry registry = RegistryUtil.getRegistry();
ServiceTracker<UserService, UserService> tracker = registry.trackServices(UserService.class);
tracker.open();
UserService userService = tracker.getService();
if (userService ≠ null) {
    User user = userService.getCurrentUser();
    System.out.println("Current user is " + user.getFirstName() + StringPool.SPACE +
        user.getLastName());
}
tracker.close();
```

Remember to open your service trackers before use and close them after use. If you must use Liferay DXP's OSGi services in a servlet, for example, it's a good idea to open your service trackers in `Servlet.init()` and close them in `Servlet.destroy()`.

Related Topics

Calling Non-OSGi Code that Uses OSGi Services

OSGi Basics For Liferay Development

116.19 Sort Order Changed with a Different Database

If you've been using Liferay DXP, but are switching it to use a different database type, consult your database vendor documentation to understand your old and new database's default query result order. The default order is either case-sensitive or case-insensitive. This affects entity sort order in Liferay DXP.

Here are some examples of ascending alphabetical sort order.

Case-sensitive:

```
111
222
AAA
BBB
aaa
bbb
```

Case-insensitive:

```
111
222
AAA
aaa
BBB
bbb
```

Your new database's default query result order might differ from your current database's order. Consult your vendor's documentation to configure the order the way you want.

CHAPTER 117

DATA UPGRADES

Your module goes through various stages of development, because you're constantly trying to improve it. You add new features, remove features, enhance features, reorganize the code, and do whatever you can to respond to what your users want and make your module the best it can be. To transition users to new and improved versions of your module, you need to take them through the process of upgrading.

Liferay provides a robust data upgrade framework for you to use. Here you'll learn how to create a data upgrade process.

117.1 Creating Data Upgrade Processes for Modules

Some changes you make to a module involve modifying the database. These changes bring with them the need for an upgrade process to move your module's database from one version to the next. Liferay has an upgrade framework you can use to make this easier to do. It's a feature-rich framework that makes upgrades safe: the system records the current state of the schema so that if the upgrade fails, the process can revert the module back to its previous version.

Note: Upgrade processes for traditional Liferay plugins (WAR files) work the same way they did for Liferay Portal 6.x.

Liferay DXP's upgrade framework executes your module's upgrades automatically when the new version starts for the first time. You implement concrete data schema changes in upgrade step classes and then register them with the upgrade framework using an upgrade step registrator. In this tutorial, you'll learn how to do all these things to create an upgrade process for your module.

Here's what's involved:

- **Specifying the schema version**
- **Declaring dependencies**
- **Writing upgrade steps**
- **Writing the registrator**
- **Waiting for upgrade completion**

It's time to get started.

Specifying the Schema Version

In your module's bnd.bnd file, specify a Liferay-Require-SchemaVersion header with the new schema version value. Here's an example schema version header for a module whose new schema is version 1.1.0:

```
    Liferay-Require-SchemaVersion: 1.1.0
```

Important: If no Liferay-Require-SchemaVersion header is specified, Liferay DXP considers the Bundle-Version header value to be the database schema version.

Next, you'll specify your upgrade's dependencies.

Declaring Dependencies

In your module's dependency management file (e.g., Maven POM, Gradle build file, or Ivy ivy.xml file), add a dependency on the com.liferay.portal.upgrade module.

In a build.gradle file, the dependency would look like this:

```
compile group: "com.liferay", name: "com.liferay.portal.upgrade", version: "2.0.0"
```

If there are other modules your upgrade process requires, specify them as dependencies.

You've configured your module project for the upgrade. It's time to create upgrade steps to update the database from the current schema version to the new one.

Writing Upgrade Steps

An upgrade step is a class that adapts module data to the module's target database schema. It can execute SQL commands and DDL files to upgrade the data. As a developer, you can encapsulate upgrade logic in multiple upgrade step classes per schema version.

The upgrade class extends the UpgradeProcess base class, which implements the UpgradeStep interface. Each upgrade step must override the UpgradeProcess class's method doUpgrade with instructions for modifying the database.

Since UpgradeProcess extends the BaseDBProcess class, you can use its runSQL and runSQLTemplate* methods to execute your SQL commands and SQL DDL, respectively.

If you want to create, modify, or drop tables or indexes by executing DDL sentences from an SQL file, make sure to use ANSI SQL only. Doing this assures the commands work on different databases.

If you need to use non-ANSI SQL, it's best to write it in the UpgradeProcess class's runSQL or alter methods, along with tokens that allow porting the sentences to different databases.

For example, consider the journal-service module's UpgradeSchema upgrade step class:

```
package com.liferay.journal.internal.upgrade.v0_0_4;

import com.liferay.journal.internal.upgrade.v0_0_4.util.JournalArticleTable;
import com.liferay.journal.internal.upgrade.v0_0_4.util.JournalFeedTable;
import com.liferay.portal.kernel.upgrade.UpgradeMVCCVersion;
import com.liferay.portal.kernel.upgrade.UpgradeProcess;
import com.liferay.portal.kernel.util.StringUtil;

/**
 * @author Eduardo Garcia
 */
public class UpgradeSchema extends UpgradeProcess {
    @Override
    protected void doUpgrade() throws Exception {
        String template = StringUtil.read(
            UpgradeSchema.class.getResourceAsStream("dependencies/update.sql"));

        runSQLTemplateString(template, false, false);

        upgrade(UpgradeMVCCVersion.class);

        alter(
            JournalArticleTable.class,
            new AlterColumnName(
                "structureId", "DDMStructureKey VARCHAR(75) null"),
            new AlterColumnName(
                "templateId", "DDMTemplateKey VARCHAR(75) null"),
            new AlterColumnType("description", "TEXT null"));

        alter(
            JournalFeedTable.class,
            new AlterColumnName("structureId", "DDMStructureKey TEXT null"),
            new AlterColumnName("templateId", "DDMTemplateKey TEXT null"),
            new AlterColumnName(
                "rendererTemplateId", "DDMRendererTemplateKey TEXT null"),
            new AlterColumnType("targetPortletId", "VARCHAR(200) null"));
    }

}
```

The above example class `UpgradeSchema` uses the `runSQLTemplateString` method to execute ANSI SQL DDL from an SQL file. To modify column names and column types, it uses the `alter` method and `UpgradeProcess`'s `UpgradeProcess.AlterColumnName` and `UpgradeProcess.AlterColumnType` inner classes as token classes.

Here's a simpler example upgrade step from the `com.liferay.calendar.service` module. It uses the `alter` method to modify a column type in the calendar booking table:

```
public class UpgradeCalendarBooking extends UpgradeProcess {

        @Override
        protected void doUpgrade() throws Exception {
                alter(
                        CalendarBookingTable.class,
                        new AlterColumnType("description", "TEXT null"));
        }

}
```

You can implement upgrade steps just like these for your module schemas.

How you name and organize upgrade steps is up to you. Liferay's upgrade classes are organized using a package structure similar to this one:

- *some.package.structure*
    - `upgrade`
        - `v1_1_0`
            - `UpgradeFoo.java` ← Upgrade Step
        - `v2_0_0`
            - `UpgradeFoo.java` ← Upgrade Step

- UpgradeBar.java ← Upgrade Step

* MyCustomModuleUpgrade.java ← Registrator

The example upgrade structure shown above is for a module that has two database schema versions: `1.1.0` and `2.0.0`. They're represented by packages `v1_1_0` and `v2_0_0`. Each version package contains upgrade step classes that update the database. The example upgrade steps focus on fictitious data elements `Foo` and `Bar`. The registrator class (`MyCustomModuleUpgrade`, in this example) is responsible for registering the applicable upgrade steps for each schema version.

Here are some organizational tips:

- Put all upgrade classes in a sub-package called `upgrade`.
- Group together similar database updates (ones that operate on a data element or related data elements) in the same upgrade step class.
- Create upgrade steps in sub-packages named after each data schema version.

Before continuing with upgrade step registrators, if your application was modularized from a former traditional Liferay plugin application (application WAR) and it uses Service Builder, it requires a Bundle Activator to register itself in Liferay DXP's `Release_` table. If this is the case for your application, create and register a Bundle Activator and then return here to write your upgrade step registrator.

Writing the Upgrade Step Registrator

A module's upgrade step registrator notifies Liferay's upgrade framework of all the upgrade steps to update the module data for each schema version. It specifies the module's entire upgrade process. The upgrade framework executes the upgrade steps to update the current module data to the latest schema.

For example, the upgrade step registrator class `MyCustomModuleUpgrade` (below) registers upgrade steps incrementally for each schema version (past and present):

```
package com.liferay.mycustommodule.upgrade;

import com.liferay.portal.upgrade.registry.UpgradeStepRegistrator;

import org.osgi.service.component.annotations.Component;

@Component(immediate = true, service = UpgradeStepRegistrator.class)
public class MyCustomModuleUpgrade implements UpgradeStepRegistrator {

    @Override
    public void register(Registry registry) {
        registry.register(
            "com.liferay.mycustommodule", "0.0.0", "2.0.0",
            new DummyUpgradeStep());

        registry.register(
            "com.liferay.mycustommodule", "1.0.0", "1.1.0",
            new com.liferay.mycustommodule.upgrade.v1_1_0.UpgradeFoo());

        registry.register(
            "com.liferay.mycustommodule", "1.1.0", "2.0.0",
            new com.liferay.mycustommodule.upgrade.v2_0_0.UpgradeFoo(),
            new com.liferay.mycustommodule.upgrade.v2_0_0.UpgradeBar());
    }

}
```

The registrator's `register` method informs the upgrade framework about each new schema and associated upgrade steps to adapt data to it. Each schema upgrade is represented by a registration. A registration is an abstraction for all the changes you need to apply to the database from one schema version to the next one.

The following diagram illustrates the relationship between the registrator and the upgrade steps.

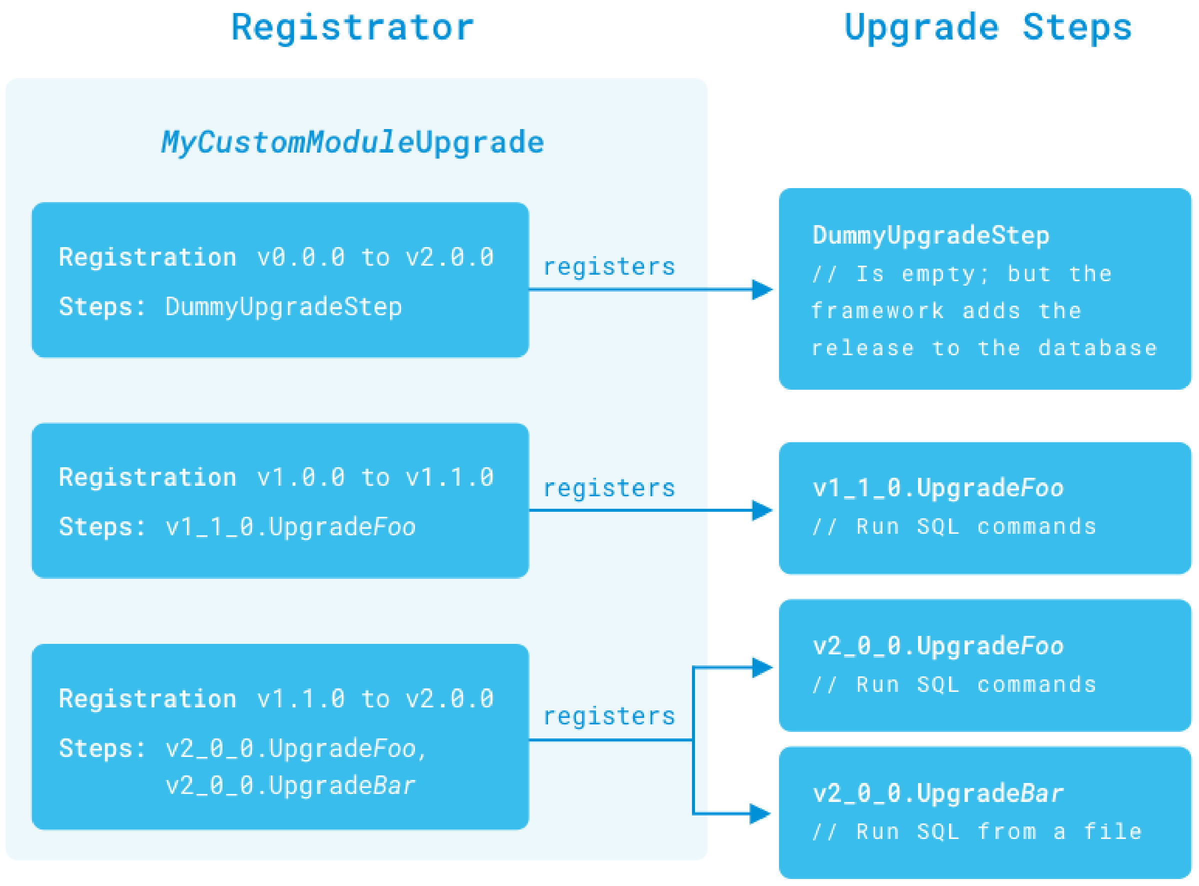


Figure 117.1: In a registrator class, the developer specifies a registration for each schema version upgrade. The upgrade steps handle the database updates.

The previous example `MyCustomModuleUpgrade` registrator class listing shows how this works.

The registrator class declares itself to be an OSGi Component of service type `UpgradeStepRegistrator.class`. The `@Component` annotation registers the class to the OSGi framework as the module's upgrade step registrator. The attribute `immediate = true` tells the OSGi framework to activate this module immediately after it's installed.

The registrator implements the `UpgradeStepRegistrator` interface, which is in the `com.liferay.portal.upgrade` module. The interface declares a `register` method that the registrator must override. In that method, the registrator implements all the module's upgrade registrations.

Upgrade registrations are defined by the following values:

- **Module's bundle symbolic name**
- **Schema version to upgrade from** (as a `String`)
- **Schema version to upgrade to** (as a `String`)

- **List of upgrade steps**

The example registrator `MyCustomModuleUpgrade` registers three upgrades:

- `0.0.0` to `2.0.0`
- `1.0.0` to `1.1.0`
- `1.1.0` to `2.0.0`

The `MyCustomModuleUpgrade` registrator's first registration is applied by the upgrade framework if the module has not been installed previously. Its list of upgrade steps contains only one: `new DummyUpgradeStep()`.

```
registry.register(
    "com.liferay.document.library.web", "0.0.0", "2.0.0",
    new DummyUpgradeStep());
```

The `DummyUpgradeStep` class provides an empty upgrade step. The `MyCustomModuleUpgrade` registrator defines this registration so that the upgrade framework records the module's latest schema version (i.e., `2.0.0`) in Liferay DXP's `Release_` table.

Important: Modules that use Service Builder *should not* define a registration for their initial database schema version, as Service Builder already records their schema versions to Liferay DXP's `Release_` table. Modules that don't use Service Builder, however, *should* define a registration for their initial schema.

The `MyCustomUpgrade` registrator's next registration (from schema version `1.0.0` to `1.1.0`) includes one upgrade step.

```
registry.register(
    "com.liferay.mycustommodule", "1.0.0", "1.1.0",
          new com.liferay.mycustommodule.upgrade.v1_1_0.UpgradeFoo());
```

The upgrade step's fully qualified class name is required because classes named `UpgradeFoo` are in package `com.liferay.mycustommodule.upgrade.v1_1_0`and `com.liferay.mycustommodule.upgrade.v2_0_0`.

The registrator's final registration (from schema version `1.1.0` to `2.0.0`) contains two upgrade steps.

```
registry.register(
    "com.liferay.mycustommodule", "1.1.0", "2.0.0",
    new com.liferay.mycustommodule.upgrade.v2_0_0.UpgradeFoo(),
    new UpgradeBar());
```

Both upgrade steps, `UpgradeFoo` and `UpgradeBar`, reside in the module's `com.liferay.mycustommodule.upgrade.v2_0_0` package. The fully qualified class name `com.liferay.mycustommodule.upgrade.v2_0_0.UpgradeFoo` is used for the `UpgradeFoo` class, while the simple class name `UpgradeBar` is fine for the second upgrade step.

A registration's upgrade step list can consist of as many upgrade steps as needed.

Important: If your upgrade step uses an OSGi service, your upgrade must wait for that service's availability. To specify that your upgrade is to be executed only after that service is available, add an OSGi reference to that service.

For example, the `WikiServiceUpgrade` registrator class references the `SettingsFactory` class. The upgrade step class `UpgradePortletSettings` upgrade step uses it. Here's the `WikiServiceUpgrade` class:

```
@Component(immediate = true, service = UpgradeStepRegistrator.class)
public class WikiServiceUpgrade implements UpgradeStepRegistrator {

    @Override
    public void register(Registry registry) {
        registry.register(
            "com.liferay.wiki.service", "0.0.1", "0.0.2", new UpgradeSchema());
        registry.register(
            "com.liferay.wiki.service", "0.0.2", "0.0.3",
            new UpgradeKernelPackage(), new UpgradePortletId());

        registry.register(
            "com.liferay.wiki.service", "0.0.3", "1.0.0",
            new UpgradeCompanyId(), new UpgradeLastPublishDate(),
            new UpgradePortletPreferences(),
            new UpgradePortletSettings(_settingsFactory),
            new UpgradeWikiPageResource());
    }

    @Reference(unbind = "-")
    protected void setSettingsFactory(SettingsFactory settingsFactory) {
        _settingsFactory = settingsFactory;
    }

    private SettingsFactory _settingsFactory;

}
```

In the third registration in the listing above, the `UpgradePortletSettings` upgrade step uses the `SettingsFactory` service. The `setSettingsFactory` method's `@Reference` annotation declares that the registrator class depends on and must wait for the `SettingsFactory` service to be available in the run time environment. The annotation's attribute setting `unbind = "-"` indicates that the registrator class has no method for unbinding the service.

Next, you must make sure the module's upgrade is executed before making its services available.

Waiting for Upgrade Completion

Before module services that access the database are used, the database should be upgraded to the latest database schema.

As a convenience, configuring the Bnd header `Liferay-Require-SchemaVersion` to the latest schema version is all that's required to assure the database is upgraded for Service Builder services.

For all other services, the developer can assure database upgrade by specifying an `@Reference` annotation that targets the containing module and its latest schema version.

Here are the target's required attributes:

- `release.bundle.symbolic.name`: module's bundle symbolic name
- `release.schema.version`: module's current schema version

For example, the `com.liferay.comment.page.comments.web` module's `PageCommentsPortlet` class assures upgrading to schema version `1.0.0` by defining the following reference:

```
@Reference(
    target = "(&(release.bundle.symbolic.name=com.liferay.comment.page.comments.web)(release.schema.version=1.0.0))",
    unbind = "-"
)
protected void setRelease(Release release) {
}
```

Dependencies between OSGi services can reduce the number of service classes in which upgrade reference annotations are needed. For example, there's no need to add an upgrade reference in a dependent service, if the dependency already refers to the upgrade.

Note: Data verifications using the class `VerifyProcess` are deprecated. Verifications should be tied schema versions. Upgrade processes are associated with schema versions but `VerifyProcess` instances are not.

Now you know how to create data upgrades for all your modules. You specify the new data schema version in the `bnd.bnd` file, add a reference to your module and to the schema version to assure upgrade execution if the module doesn't use Service Builder, and add a dependency on the `com.liferay.portal.upgrade` module. For the second part of the process, you create upgrade step classes to update the database schema and register the upgrade steps in a registrator class. That's all there is to it!

Related Topics

Upgrade Processes for Former Service Builder Plugins

Upgrading Plugins to Liferay 7

Configuration

Migrating Data Upgrade Processes to the New Framework for Modules

117.2 Upgrade Processes for Former Service Builder Plugins

If you modularized a traditional Liferay plugin application that implements Service Builder services, your new modular application must register itself in the `Release_` table. This is required regardless of whether release records already exist for previous versions of the app. A Bundle Activator is the recommended way to add a release record for the first modular version of your converted application. Here you'll see an example Bundle Activator and learn how to create and activate a Bundle Activator for your application.

Important: This tutorial only applies to modular applications that use Service Builder and were modularized from traditional Liferay plugin applications. It does not apply to you if your application does not use Service Builder or has never been a traditional Liferay plugin application (a WAR application).

Bundle Activator class code is dense but straightforward. Referring to an example Bundle Activator can be helpful. Here's the Liferay Knowledge Base application's Bundle Activator:

```
public class KnowledgeBaseServiceBundleActivator implements BundleActivator {

    @Override
    public void start(BundleContext bundleContext) throws Exception {
        Filter filter = bundleContext.createFilter(
            StringBundler.concat(
                "(&(objectClass=", ModuleServiceLifecycle.class.getName(), ")",
                ModuleServiceLifecycle.DATABASE_INITIALIZED, ")"));

        _serviceTracker = new ServiceTracker<Object, Object>(
            bundleContext, filter, null) {

            @Override
            public Object addingService(
                ServiceReference<Object> serviceReference) {

                try {
                    BaseUpgradeServiceModuleRelease
                        upgradeServiceModuleRelease =
                            new BaseUpgradeServiceModuleRelease() {

                                @Override
                                protected String getNamespace() {
                                    return "KB";
                                }
                                @Override
                                protected String getNewBundleSymbolicName() {
                                    return "com.liferay.knowledge.base.service";
                                }

                                @Override
                                protected String getOldBundleSymbolicName() {
                                    return "knowledge-base-portlet";
                                }

                            };

                    upgradeServiceModuleRelease.upgrade();

                    return null;
                }
                catch (UpgradeException ue) {
                    throw new RuntimeException(ue);
                }
            }

        };

        _serviceTracker.open();
    }

    @Override
    public void stop(BundleContext bundleContext) throws Exception {
        _serviceTracker.close();
    }

    private ServiceTracker<Object, Object> _serviceTracker;

}
```

The following steps explain how to create a Bundle Activator, like the example above.

1. Create a class that implements the interface `org.osgi.framework.BundleActivator`.

2. Add a service tracker field:

   `private ServiceTracker<Object, Object> _serviceTracker;`

3. Override BundleActivator's `stop` method to close the service tracker:

   ```
   @Override
   public void stop(BundleContext bundleContext) throws Exception {
       _serviceTracker.close();
   }
   ```

4. Override BundleActivator's `start` method to instantiate a service tracker that creates a filter to listens for the app's database initialization event and initializes the service tracker to use that filter. You'll add the service tracker initialization code in the next steps. At the end of the `start` method, open the service tracker.

   ```
   @Override
   public void start(BundleContext bundleContext) throws Exception {
       Filter filter = bundleContext.createFilter(
           StringBundler.concat(
               "(&(objectClass=", ModuleServiceLifecycle.class.getName(), ")",
               ModuleServiceLifecycle.DATABASE_INITIALIZED, ")"));
   ```

```
    _serviceTracker = new ServiceTracker<Object, Object>(
        bundleContext, filter, null) {
        // See the next step for this code ...
    };

    _serviceTracker.open();
}
```

5. In the service tracker initialization block `{ // See the next step for this code ... }` from the previous step, add an `addingService` method that instantiates a `BaseUpgradeServiceModuleRelease` for describing your app. The example `BaseUpgradeServiceModuleRelease` instance below describes Liferay's Knowledge Base app:

```
@Override
public Object addingService(
    ServiceReference<Object> serviceReference) {

    try {
        BaseUpgradeServiceModuleRelease
                upgradeServiceModuleRelease =
            new BaseUpgradeServiceModuleRelease() {

                @Override
                protected String getNamespace() {
                    return "KB";
                }

                @Override
                protected String getNewBundleSymbolicName() {
                    return "com.liferay.knowledge.base.service";
                }

                @Override
                protected String getOldBundleSymbolicName() {
                    return "knowledge-base-portlet";
                }

            };

        upgradeServiceModuleRelease.upgrade();

        return null;
    }
    catch (UpgradeException ue) {
        throw new RuntimeException(ue);
    }
}
```

The `BaseUpgradeServiceModuleRelease` implements the following methods:

- `getNamespace`: Returns the namespace value as specified in the former plugin's `service.xml` file. This value is also in the `buildNamespace` field in the plugin's `ServiceComponent` table record.
- `getOldBundleSymbolicName`: Returns the former plugin's name.
- `getNewBundleSymbolicName`: Returns the module's symbolic name. In the module's `bnd.bnd` file, it's the `Bundle-SymbolicName` value.
- `upgrade`: Invokes the app's upgrade processes.

6. In the module's `bnd.bnd` file, reference the Bundle Activator class you created. Here's the example's Bundle Activator reference:

```
Bundle-Activator: com.liferay.knowledge.base.internal.activator.KnowledgeBaseServiceBundleActivator
```

The Bundle Activator uses one of the following values to initialize the `schemaVersion` field in the application's `Release_` table record:

- Current `buildNumber`: if there is an existing Release_ table record for the previous plugin.
- `0.0.1`: if there is no existing Release_ table record.

You've set your service module's data upgrade process.

Related Topics

Creating Data Upgrade Processes for Modules

Upgrading Plugins to Liferay 7

117.3 Upgrading Data Schemas in Development

As you develop modules, you might need to iterate through several database schema changes. Before you release new module versions with your finalized schema changes, you must create a formal data upgrade process. Until then, you can use the Build Auto Upgrade feature to test schema changes on the fly.

Note: In Liferay Portal 6.x Service Builder portlets, the `build.auto.upgrade` property in `service.properties` applies Liferay Service schema changes upon rebuilding services and redeploying the portlets. As of 7.0, this property is deprecated.

Liferay Digital Enterprise 7.0 Fix Pack 28, and Liferay CE Portal 7.0.4 GA5 reintroduce Build Auto Upgrade in a new global property `schema.module.build.auto.upgrade` in the file `[Liferay_Home]/portal-developer.properties`.

Setting the global property `schema.module.build.auto.upgrade` to `true` applies module schema changes for redeployed modules whose service build numbers have incremented. The `build.number` property in the module's `service.properties` file indicates the service build number. Build Auto Upgrade executes schema changes without massaging existing data. It leaves data empty for created columns, drops data from deleted and renamed columns, and orphans data from deleted and renamed tables.

Although Build Auto Upgrade updates databases quickly and automatically, it doesn't guarantee a proper data upgrade–you implement that via data upgrade processes. Build Auto Upgrade is for development purposes only.

WARNING: DO NOT USE the Build Auto Upgrade feature in production. Liferay DXP DOES NOT support Build Auto Upgrade in production. Build Auto Upgrade is for development purposes only. Enabling it in production can result in data loss and improper data upgrade. In production environments, leave the property `schema.module.build.auto.upgrade` in `portal-developer.properties` set to `false`.

By default, `schema.module.build.auto.upgrade` is set to `false`. On any module's first deployment, the module's tables are generated regardless of the `schema.module.build.auto.upgrade` value.

The following table summarizes Build Auto Upgrade's handling of schema changes:

Schema Change	Result
Add column	Create a new empty column.

Schema Change	Result
Rename column	Drop the existing column and delete all its data. Create a new empty column.
Delete column	Drop the existing column and delete all its data.
Create or rename a table in Liferay DXP's built-in data source.	Orphan the existing table and all its data. Create the new table.

Great! Now you know how to use the Build Auto Upgrade developer feature.

Related Topics

Creating Data Upgrade Process for Modules

Chapter 118

Back-end Frameworks

What are back-end frameworks? Are they important? If so, why aren't they up-front and center in the docs? Good questions.

Back-end frameworks are analogous to supporting actors and actresses in show business. They fill out the stories in films we know and love. As actors bring richness and life to their films, Liferay's powerful back-end frameworks bring essential services and deliver terrific performances of their own. Here are some of the frameworks:

- Device Recognition
- Message Bus

These frameworks and more deliver smashing performances and are stars in their own right.

118.1 Device Recognition API

As you know, Internet traffic has risen exponentially over the past decade, and this trend shows no sign of stopping. In addition, the bulk of Internet traffic now comes from mobile devices. The mobile boom presents new obstacles and challenges for content management. How will content adapt to different devices with different capabilities? How can your grandma's gnarly tablet and your cousin's awesome new smart phone request the same information from your portal?

The Device API detects the capabilities of any device making a request to your portal. It can also determine what mobile device or operating system was used to make a request, and then follows rules to make Liferay DXP render pages based on the device. To use these features, you must first install a device detection database that can detect which mobile devices are accessing the portal. Liferay DXP provides such a database in the Liferay Mobile Device Detection (LMDD) app from the Liferay Marketplace. Click here for instructions on using Liferay Marketplace to find and install apps.

Important: On Windows, Liferay Mobile Device Detection Enterprise must be run on a 64-bit JVM. On all operating systems, Liferay Mobile Device Detection Enterprise requires a JVM minimum memory setting of at least 2 gb.

You can create your own plugin that makes use of the device database. This tutorial shows you some of the methods in the Device API that you can use to retrieve device attributes and capabilities. Now go ahead and get started!

Getting Started with the Device API

One important thing that you'll want to get using the Device API is the `Device` object. You can obtain it from the `themeDisplay` object like this:

```
Device device = themeDisplay.getDevice();
```

You can view the `Device` API. Using some of the methods from the Javadocs, here's an example that obtains a device's dimensions:

```
Dimensions dimensions = device.getScreenSize();
float height = dimensions.getHeight();
float width = dimensions.getWidth();
```

Now your code can get the `Device` object and the dimensions of a device. Of course, this is just a simple example. You can acquire many other device attributes that help you take care of the pesky problems that arise when sending content to different devices. Refer to the Device Javadocs mentioned above for assistance. Next, you'll learn about retrieving some other device capabilities.

Getting Device Properties

With the Device API, you can detect the *properties* of a device making a request to your portal and then render content accordingly. Properties refer to things that the requesting device can do. For example, you can determine the device's operating system, browser, form factor, and much more. Properties can be retrieved with the `getCapability` and `getCapabilities` methods of the Device API.

Most of the properties of the requesting device can be detected, depending on the device detection implementation you're using. For example, you can obtain the brand name of the device with this code:

```
String brand = device.getCapability("OEM");
```

You can grab many other device properties, including `HardwareModel`, `HardwareName`, `ReleaseYear`, and `ReleaseMonth`. You can also get boolean values like `IsMobile`, `IsTablet`, and many more.

Keep in mind the Device API is an API. The underlying implementation of the Device API may change. You can also implement your own. Thus, the device property names are specific and proprietary to the underlying Device API implementation.

Now that you know about the Device API, you can use it to make sure that your grandma's gnarly tablet and your cousin's awesome new smart phone can make requests to your portal and receive identical content. This will make everyone happy!

Related Topics

Using the Mobile SDK

Service Builder and Services

Chapter 119

Message Bus

If you ever need to do some data processing outside the scope of the web's request/response, look no further than the Message Bus. It's conceptually similar to Java Messaging Service (JMS) Topics, but sacrifices transactional, reliable delivery capabilities, making it much lighter-weight. Liferay DXP uses Message Bus all over the place:

- Auditing
- Search engine integration
- Email subscriptions
- Monitoring
- Document Library processing
- Background tasks
- Cluster-wide request execution
- Clustered cache replication

You can use it too! Here are some of Message Bus's most important features:

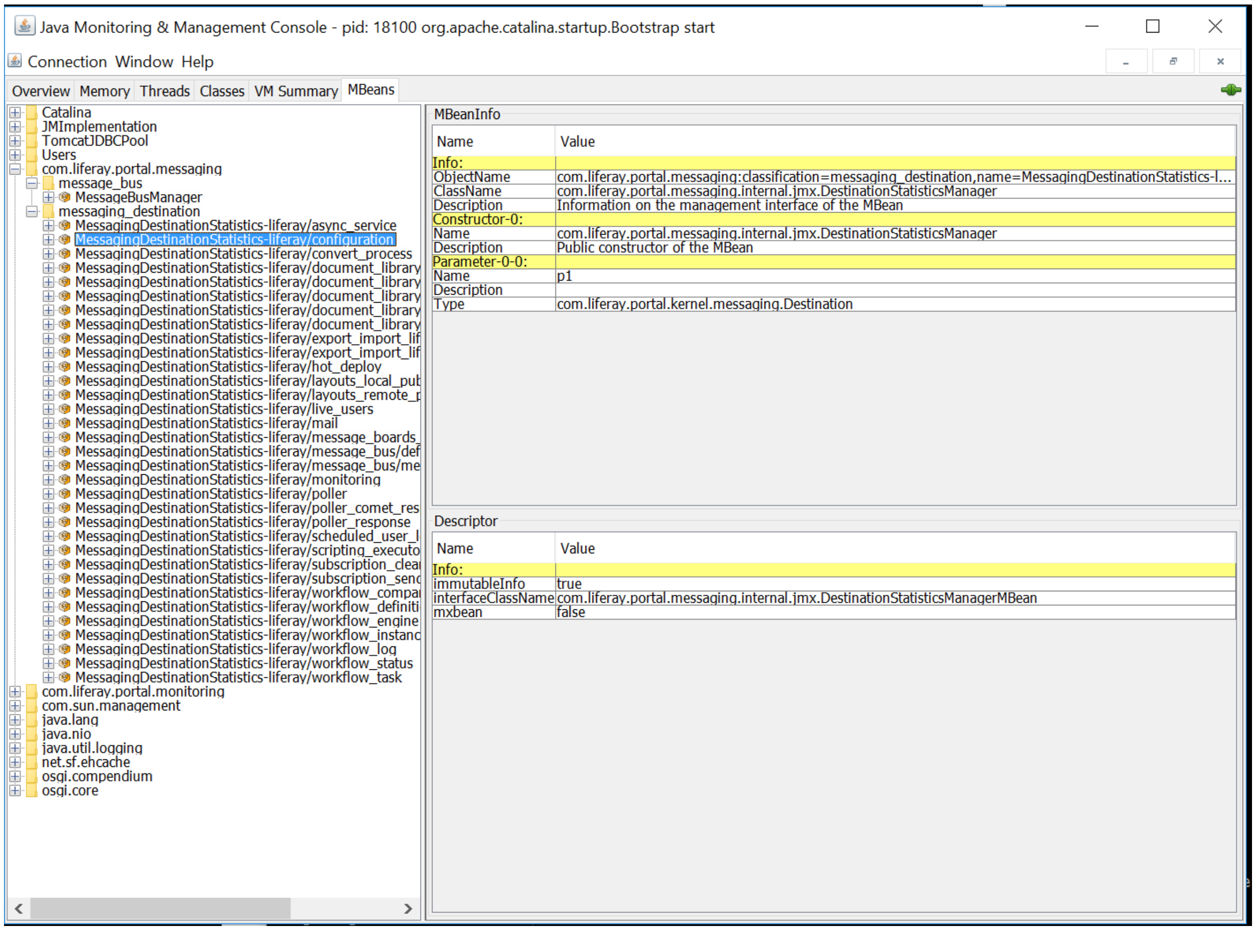
- publish/subscribe messaging
- request queuing and throttling
- flow control
- multi-thread message processing

There are also tools, such as the Java SE's JConsole, that can monitor Message Bus activities.

The Message Bus topics are covered in these tutorials:

- Messaging Destinations
- Message Listeners
- Sending Messages

Since all messages are sent to and received at destinations, messaging destinations is worth exploring first.


Java Monitoring & Management Console - pid: 18100 org.apache.catalina.startup.Bootstrap start
Connection Window Help
Overview Memory Threads Classes VM Summary MBeans

```
Catalina
JMImplementation
TomcatJDBCPool
Users
com.liferay.portal.messaging
  message_bus
    MessageBusManager
  messaging_destination
    MessagingDestinationStatistics-liferay/async_service
    MessagingDestinationStatistics-liferay/configuration
    MessagingDestinationStatistics-liferay/convert_process
    MessagingDestinationStatistics-liferay/document_library
    MessagingDestinationStatistics-liferay/document_library
    MessagingDestinationStatistics-liferay/document_library
    MessagingDestinationStatistics-liferay/document_library
    MessagingDestinationStatistics-liferay/document_library
    MessagingDestinationStatistics-liferay/export_import_lif
    MessagingDestinationStatistics-liferay/export_import_lif
    MessagingDestinationStatistics-liferay/hot_deploy
    MessagingDestinationStatistics-liferay/layouts_local_pub
    MessagingDestinationStatistics-liferay/layouts_remote_p
    MessagingDestinationStatistics-liferay/live_users
    MessagingDestinationStatistics-liferay/mail
    MessagingDestinationStatistics-liferay/message_boards_
    MessagingDestinationStatistics-liferay/message_bus/def
    MessagingDestinationStatistics-liferay/message_bus/me
    MessagingDestinationStatistics-liferay/monitoring
    MessagingDestinationStatistics-liferay/poller
    MessagingDestinationStatistics-liferay/poller_comet_res
    MessagingDestinationStatistics-liferay/poller_response
    MessagingDestinationStatistics-liferay/scheduled_user_l
    MessagingDestinationStatistics-liferay/scripting_executo
    MessagingDestinationStatistics-liferay/subscription_clea
    MessagingDestinationStatistics-liferay/subscription_send
    MessagingDestinationStatistics-liferay/workflow_compa
    MessagingDestinationStatistics-liferay/workflow_definiti
    MessagingDestinationStatistics-liferay/workflow_engine
    MessagingDestinationStatistics-liferay/workflow_instanc
    MessagingDestinationStatistics-liferay/workflow_log
    MessagingDestinationStatistics-liferay/workflow_status
    MessagingDestinationStatistics-liferay/workflow_task
com.liferay.portal.monitoring
com.sun.management
java.lang
java.nio
java.util.logging
net.sf.ehcache
osgi.compendium
osgi.core
```

MBeanInfo

Name	Value
Info:	
ObjectName	com.liferay.portal.messaging:classification=messaging_destination,name=MessagingDestinationStatistics-l...
ClassName	com.liferay.portal.messaging.internal.jmx.DestinationStatisticsManager
Description	Information on the management interface of the MBean
Constructor-0:	
Name	com.liferay.portal.messaging.internal.jmx.DestinationStatisticsManager
Description	Public constructor of the MBean
Parameter-0-0:	
Name	p1
Description	
Type	com.liferay.portal.kernel.messaging.Destination

Descriptor

Name	Value
Info:	
immutableInfo	true
interfaceClassName	com.liferay.portal.messaging.internal.jmx.DestinationStatisticsManagerMBean
mxbean	false

Figure 119.1: JConsole shows statistics on Message Bus messages sent, messages pending, and more.

119.1 Messaging Destinations

In Message Bus, you send messages to *destinations*. A destination is a named logical (not physical) location. Sender classes send messages to destinations, while listener classes wait to receive messages at the destinations. In this way, the sender and recipient don't need to know each other–they're loosely coupled. Here are the messaging destination topics this tutorial covers:

- Destination configuration
- Creating a destination
- Messaging event listeners

It's time to configure a destination.

Destination Configuration

Each destination has a name and type and can have several other attributes. The destination type determines whether there's a message queue, the kinds of threads involved with a destination, and the message delivery behavior to expect at the destination.

Here are the primary destination types:

- **Parallel Destination**

  - Messages sent here are queued.
  - Multiple worker threads from a thread pool deliver each message to a registered message listener. There's one worker thread per message per message listener.

- **Serial Destination**

  - Messages sent here are queued.
  - Worker threads from a thread pool deliver the messages to each registered message listener, one worker thread per message.

- **Synchronous Destination**

  - Messages sent here are directly delivered to message listeners.
  - The thread sending the message here delivers the message to all message listeners also.

Liferay has preconfigured destinations for various purposes. The `DestinationNames` class defines `String` constants for each of them. For example, `DestinationNames.HOT_DEPLOY` (value is `"liferay/hot_deploy"`) is for deployment event messages. Since destinations are tuned for specific purposes, don't modify them.

Destinations are based on `DestinationConfiguration` instances. The configuration specifies the destination type, name, and these destination- related attributes:

Maximum Queue Size: limits the number of queued messages for the destination.

Rejected Execution Handler: A `com.liferay.portal.kernel.concurrent.RejectedExecutionHandler` instance can take action (e.g., log warnings) regarding rejected messages when the destination queue is full.

Workers Core Size: initial number of worker threads for processing messages.

Workers Max Size: limits the number of worker threads for processing messages.

The `DestinationConfiguration` class provides these static methods for creating the various types of configurations.

- `createParallelDestinationConfiguration(String destinationName)`
- `createSerialDestinationConfiguration(String destinationName)`
- `createSynchronousDestinationConfiguration(String destinationName)`

You can also use the `DestinationConfiguration` constructor to create a configuration for any destination type, even your own.

Creating a Destination

Message Bus destinations are based on destination configurations and registered as OSGi services. Message Bus detects the destination services and manages their associated destinations.

Here are the general steps for creating a destination. The example configurator class that follows demonstrates these steps.

1. Create a destination configuration using one of `DestinationConfiguration`'s static `create*` methods or its constructor. Set any attributes that apply to the destinations you'll create with it.

2. Create a destination by invoking the `DestinationFactory` method `createDestination(DestinationConfiguration)`, passing in the destination configuration you created in the previous step.

3. Register the destination as an OSGi service by invoking the `BundleContext` method `registerService`, passing in the following parameters.

    - Destination class `Destination.class`
    - Your `Destination` object
    - A `Dictionary` of properties defining the destination, including the `destination.name`

4. Manage the destination object and service registration resources using a collection, such as a `Map<String, ServiceRegistration<Destination>>`. Keeping references to these resources is helpful for when you're ready to unregister and destroy them. The `deactivate` method in example below demonstrates this.

Here's an example messaging configurator component that creates and registers a parallel destination and manages its resources:

```
@Component (
    immediate = true,
    service = MyMessagingConfigurator .class
)
public class MyMessagingConfigurator {

    @Activate
    protected void activate(BundleContext bundleContext) {

        _bundleContext = bundleContext;

        // Create a DestinationConfiguration for parallel destinations.

        DestinationConfiguration destinationConfiguration =
            new DestinationConfiguration(
                DestinationConfiguration.DESTINATION_TYPE_PARALLEL,
                    "myDestinationName");

        // Set the DestinationConfiguration's max queue size and
        // rejected execution handler.

        destinationConfiguration.setMaximumQueueSize(_MAXIMUM_QUEUE_SIZE);

        RejectedExecutionHandler rejectedExecutionHandler =
            new CallerRunsPolicy() {

                @Override
                public void rejectedExecution(
                    Runnable runnable, ThreadPoolExecutor threadPoolExecutor) {
```

```
                if (_log.isWarnEnabled()) {
                    _log.warn(
                        "The current thread will handle the request " +
                            "because the graph walker's task queue is at " +
                                "its maximum capacity");
                }

                super.rejectedExecution(runnable, threadPoolExecutor);
            }

        };

     destinationConfiguration.setRejectedExecutionHandler(
          rejectedExecutionHandler);

     // Create the destination

     Destination destination = _destinationFactory.createDestination(
          kaleoGraphWalkerDestinationConfiguration);

      // Add the destination to the OSGi service registry

      Dictionary<String, Object> properties = new HashMapDictionary<>();

      properties.put("destination.name", destination.getName());

      ServiceRegistration<Destination> serviceRegistration =
          _bundleContext.registerService(
              Destination.class, destination, properties);

      // Track references to the destination service registrations

      _serviceRegistrations.put(destination.getName(),
          serviceRegistration);
    }

    @Deactivate
    protected void deactivate() {

        // Unregister and destroy destinations this component unregistered

        for (ServiceRegistration<Destination> serviceRegistration :
        _serviceRegistrations.values()) {

            Destination destination = _bundleContext.getService(
                serviceRegistration.getReference());

            serviceRegistration.unregister();

            destination.destroy();

        }

        _serviceRegistrations.clear();

     }

    @Reference
    private DestinationFactory _destinationFactory;

    private final Map<String, ServiceRegistration<Destination>>
        _serviceRegistrations = new HashMap<>();
}
```

On activation, the example configurator above does these things:

1. Creates a `DestinationConfiguration` for parallel destinations.

2. Sets the `DestinationConfiguration`'s max queue size and a rejected execution handler.

3. Uses the `DestinationFactory` (the one bound to the `_destinationFactory` field) to create the destination.

4. Adds the destination to the OSGi service registry

5. Adds the destination service registration to a map for managing them.

Once the destination is registered, Message Bus detects its service and manages the destination. On the example configurator's deactivation, its `deactivate` method unregisters the destination services and destroys the destinations.

As an added bonus to creating destinations, you can create classes that listen for new destinations and new message listeners. You might want to create such listeners to keep up to log the deployment of new message bus endpoints.

Messaging Event Listeners

There are Message Bus framework interfaces that let you listen for new destinations and message listeners.

Listening for new Destinations

The Message Bus notifies Message Bus Event Listeners when destinations are added and removed. To register these listeners, publish a `MessageBusEventListener` instance to the OSGi service registry (e.g., via an `@Component` annotation).

```
@Component(
    immediate = true,
    service = MessageBusEventListener.class
)
public class MyMessageBusEventListener implements MessageBusEventListener {

    void destinationAdded(Destination destination) {
        ...
    }

    void destinationDestroyed(Destination destination) {
        ...
    }
}
```

Listening for new message listeners is easy too.

Listening for new Message Listeners

The Message Bus notifies `DestinationEventListener` instances when message listeners are either registered or unregistered to destinations. To register a listener to a destination, publish a `DestinationEventListener` service to the OSGi service registry, making sure to specify the destination's `destination.name` property.

```
@Component(
    immediate = true,
    property = {"destination.name=myCustom/Destination"},
    service = DestinationEventListener.class
)
public class MyDestinationEventListener implements DestinationEventListener {

    void messageListenerRegistered(String destinationName,
                                   MessageListener messageListener) {
        ...
    }

    void messageListenerUnregistered(String destinationName,
                                MessageListener messageListener) {
        ...
    }
}
```

And that's how you listen for new destinations and message listeners.

Now you understand the different destination types, how to create and register destinations, and how to manage destination resources. Once you deploy your destination, registered message listeners receive messages sent to it.

Related Topics

Message Listeners

Sending Messages

119.2 Message Listeners

If you're interested in messages sent to a destination, you need to "listen" for them. That is, you must create and register a message listener for the destination.

To create a message listener, implement the `MessageListener` interface and override its `receive(Message)` method to process messages your way.

```
public void receive(Message message) {
    // Process messages your way
}
```

Here are the ways to register your listener with Message Bus:

- **Automatic Registration as a Component**: Publish the listener to the OSGi registry as a Declarative Services Component that specifies a destination. Message Bus automatically wires the listener to the destination.

- **Registering via MessageBus**: Obtain a reference to the Message Bus and use it directly to register the listener to a destination.

- **Registering directly to a Destination**: Obtain a reference to a specific destination and use it directly to register the listener with that destination.

Note: The `DestinationNames` class defines `String` constants for Liferay's preconfigured destinations.

The Declarative Services component module provides the easiest way to register a message listener.

Automatic Registration as a Component

You can specify a message listener in the Declarative Services (DS) `@Component` annotation:

```
@Component (
    immediate = true,
    property = {"destination.name=myCustom/Destination"},
    service = MessageListener.class
)
public class MyMessageListener implements MessageListener {
   ...

   public void receive(Message message) {
       // Handle the message
   }
}
```

The Message Bus listens for `MessageListener` service components like this one to publish themselves to the OSGi service registry. The attribute `immediate = true` tells the OSGi framework to activate the component as soon as its dependencies resolve. Message Bus wires each registered listener to the destination its `destination.name` property specifies. If the destination is not yet registered, Message Bus queues the listener until the destination registers.

Registration as a component is the preferred way to register message listeners to destinations.

Registering via MessageBus

You can use the `MessageBus` instance directly to register message listeners to destinations. You might want to do this if, for example, you want to create some special proxy wrappers. Here's a registrator that demonstrates registering a listener this way:

```
@Component (
    immediate = true,
    service = MyMessageListenerRegistrator.class
)
public class MyMessageListenerRegistrator {
    ...

    @Activate
    protected void activate() {

        _messageListener = new MessageListener() {

            public void receive(Message message) {
                // Handle the message
            }
        };

        _messageBus.registerMessageListener("myDestinationName",
            _messageListener);
    }

    @Deactivate
    protected void deactivate() {
        _messageBus.unregisterMessageListener("myDestinationName",
            _messageListener);
    }

    @Reference
    private MessageBus _messageBus;

    private MessageListener _messageListener;
}
```

The `_messageBus` field's `@Reference` annotation binds it to the `MessageBus` instance. The `activate` method creates the listener and uses the Message Bus to register the listener to a destination named `"myDestination"`. When this registrator component is destroyed, the `deactivate` method unregisters the listener.

Registering directly to the Destination

You can use a `Destination` instance to register a listener to that destination. You might want to do this if, for example, you want to create some special proxy wrappers. Here's a registrator that demonstrates registering a listener this way:

```
@Component (
    immediate = true,
    service = MyMessageListenerRegistrator.class
)
public class MyMessageListenerRegistrator {
   ...

    @Activate
    protected void activate() {

        _messageListener = new MessageListener() {

            public void receive(Message message) {
                // Handle the message
            }
        };

        _destination.register(_messageListener);
    }

    @Deactivate
    protected void deactivate() {

        _destination.unregister(_messageListener);
    }

    @Reference(target = "(destination.name=someDestination)")
    private Destination _destination;

    private MessageListener _messageListener;
}
```

The `_destination` field's `@Reference` annotation binds it to a destination named `"someDestination"`. The `activate` method creates the listener and registers it to the destination. When this registrator component is destroyed, the `deactivate` method unregisters the listener.

Now you know how to create and register message listeners for receiving messages sent to the destinations.

Related Topics

Messaging Destinations

Sending Messages

119.3 Sending Messages

Message Bus lets you send messages to destinations that have any number of listening classes. As a message sender you don't need to know the message recipients. Instead, you focus on creating message content (payload) and sending messages to destinations.

You can also send messages in a synchronous or asynchronous manner. The synchronous option waits for a response that the message was received or that it timed out. The asynchronous option gives you the "fire and forget" behavior; send the message and continue processing without waiting for a response.

Here are the message sending topics:

- Creating a message
- Sending a message (the way you want)
- Sending messages across a cluster

Start by creating a message.

Creating a Message

Here's how to create a message:

1. Call the `Message` constructor.

   `Message message = new Message();`

2. Populate the message with a `String` or `Object` payload

   - String payload: `message.setPayload("Message Bus is great!")`

   - Object payload: `message.put("firstName", "Joe")`

3. To receive responses at a particular location, set both of these attributes

   - Response destination name: `setResponseDestinationName(String)`

   - Response ID: `setResponseId(String)`

Your new message is ready to send.

Sending a Message

Here are the ways to send a message:

- Directly using the `MessageBus`
- Asynchronously using a `SingleDestinationMessageSender`
- Using a `SynchronousMessageSender`

First, let's consider using Message Bus directly.

Directly Using the Message Bus

This method involves obtaining a `MessageBus` instance and invoking it to send messages. Here's an example of directly using Message Bus to send a message.

```
@Component(
    immediate = true,
    service = SomeServiceImpl.class
)
public class SomeServiceImpl {
    ...

    public void sendSomeMessage() {

        Message message = new Message();
        message.put("myId", 12345);
        message.put("someAttribute", "abcdef");
        _messageBus.sendMessage("myDestinationName", message);
    }

    @Reference
    private MessageBus _messageBus;
}
```

To send messages asynchronously, consider using `SingleDestinationMessageSender`.

Using SingleDestinationMessageSender

The `SingleDestinationMessageSender` class wraps the Message Bus to send messages asynchronously. This class demonstrates using a `SingleDestinationMessageSender`:

```
@Component(
    immediate = true,
    service = SomeServiceImpl.class
)
public class SomeServiceImpl {
    ...

    public void sendSomeMessage() {

        Message message = new Message();
        message.put("myId", 12345);
        message.put("someValue", "abcdef");

        SingleDestinationMessageSender messageSender =
           _messageSenderFactory.createSingleDestinationMessageSender("myDestinationName");

        messageSender.send(message);
    }

    @Reference
    private SingleDestinationMessageSenderFactory _messageSenderFactory;
}
```

The `_messageSenderFactory` field's `@Reference` wires it to a `SingleDestinationMessageSenderFactory` instance. The method `sendSomeMessage` creates a message, uses the `_messageSenderFactory` to create a `SingleDestinationMessageSender` for the specified destination, and sends the message through the sender.

Using a SynchronousMessageSender

A `SynchronousMessageSender` instance sends a message to the Message Bus and blocks until receiving a response or the response times out. A `SynchronousMessageSender` has these operating modes:

- `DEFAULT`: Delivers the message in a separate thread and also provides timeouts, in case the message is not delivered properly.
- `DIRECT`: Delivers the message in the same thread of execution and blocks until it receives a response.

Here's an example of using `SynchronousMessageSender` in `DEFAULT` mode.

```
@Component(
    immediate = true,
    service = SomeServiceImpl.class
)
public class SomeServiceImpl {
    ...
    public void sendSomeMessage() {

        Message message = new Message();
        message.put("myId", 12345);
        message.put("someAttribute", "abcdef");

        SingleDestinationSynchronousMessageSender messageSender =
            _messageSenderFactory.createSingleDestinationSynchronousMessageSender(
                "myDestinationName", SynchronousMessageSender.Mode.DEFAULT);

        messageSender.send(message);

    }

    @Reference
    private SingleDestinationMessageSenderFactory _messageSenderFactory;
}
```

And those are the ways to send messages. Next, if you're in a cluster and want messages sent to a destination across all nodes, you must register a bridge message listener to that destination.

Sending Messages Across the Cluster

To ensure a message sent to a destination is received by all cluster nodes, you must register a `ClusterBridgeMessageListener` at that destination. This bridges the local destination to the cluster.

Here's a message listener registrator that bridges a destination for distributing messages to all the cluster nodes.

```
@Component(
    immediate = true,
    service = MyMessageListenerRegistrator.class
)
public class MyMessageListenerRegistrator {
    ...

    @Activate
    protected void activate() {

        _clusterBridgeMessageListener = new ClusterBridgeMessageListener();
        _clusterBridgeMessageListener.setPriority(Priority.LEVEL_5)
        _destination.register(_clusterBridgeMessageListener);
    }

    @Deactivate
    protected void deactivate() {

        _destination.unregister(_clusterBridgeMessageListener );
    }

    @Reference(target = "(destination.name=liferay/live_users)")
    private Destination _destination;

    private MessageListener _clusterBridgeMessageListener;
}
```

The destination named `"liferay/live_users"` is bound to the `_destination` field. The `activate` method creates a `ClusterBridgeMessageListener`, sets its priority queue, and registers it to the destination. Messages sent to the destination are distributed across the cluster's JVMs.

The `com.liferay.portal.kernel.cluster.Priority` class has ten levels (`Level_1` through `Level_10`, with `Level 10` being the most important). Each level is a priority queue for sending messages through the cluster. This is similar in concept to thread priorities: `Thread.MIN_PRIORITY`, `Thread.MAX_PRIORITY`, and `Thread.NORM_PRIORITY`.

That concludes the tour on sending messages. You've learned how to create messages, send messages synchronously and asynchronously, and send messages to a destination in a clustered environment.

Related Topics

Messaging Destinations

Message Listeners

Chapter 120

Audience Targeting

Liferay's Audience Targeting application lets you monitor your users. You can organize them into user segments, target specific content to those user segments, and create campaigns to expose user segments to a certain set of assets. Visit the Targeting Content to your Audience section for more information on Audience Targeting and how to use it.

Although the Audience Targeting app can be configured to monitor your audience out of the box, it is also designed as a framework to be extended by developers.

There are a set of extensions that can be easily hooked by creating other hot-deployable plugins.

These extension points include

- Rule Types
- Report Types
- Report Metrics

Audience Targeting extensions are created using OSGi modules. There are convenient Blade CLI templates for creating these projects, but you can create the modules any way you want. To use the templates, see the Blade CLI tutorials.

In this section's tutorials, you'll learn how to create these extension points for your Audience Targeting application.

120.1 Accessing the Content Targeting API

The Audience Targeting application can be used to show relevant content to users based on profiles. You might want to take the next step and use the Content Targeting API. For instance, you could list user segments in your own application or update a campaign when someone creates a calendar event. Using the Content Targeting API, you can unleash the power of Audience Targeting to the realms outside of Liferay's default applications.

In this tutorial, you'll learn how to give your application access to the Content Targeting API. Then you can view some examples of how to use the Java and JSON APIs that are available.

Exposing the Content Targeting API

Configuring your app to have access to the Content Targeting API requires only one line of code. This line of code is a dependency that should be added to your Gradle project. Follow the instructions below to add the Content Targeting API dependency.

1. Open the `build.gradle` file in your app's project folder.
2. Find the `dependencies` declaration and add the following line within that declaration:

   ```
   provided group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.api", version: "4.0.0"
   ```

   The complete declaration should look like this:

   ```
   dependencies {
       ...
       provided group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.api", version: "4.0.0"
   }
   ```

Your app now has access to the Content Targeting API and can take advantage of everything Audience Targeting has to offer. In the next section, you'll learn how to use the Content Targeting API by studying a few examples.

Using the Content Targeting Java API

There are two ways to call the Content Targeting API: through the Java API or through the JSON API.

Suppose you'd like to display a list of existing user segments in your portlet. First you need to obtain an implementation of the `UserSegmentLocalService` provided by Audience Targeting. You can do this by adding the following code to your Portlet class (e.g., the class that extends the `MVCPortlet` class):

```
@Reference(unbind = "-")
protected void setUserSegmentLocalService(
    UserSegmentLocalService userSegmentLocalService) {

    _userSegmentLocalService = userSegmentLocalService;
}

private UserSegmentLocalService _userSegmentLocalService;
```

When an implementation of the `UserSegmentLocalService` is available (i.e., the Audience Targeting app has been installed) the `_userSegmentLocalService` field is populated. Otherwise, the portlet won't be available till this dependency is resolved.

It's a good practice to access Audience Targeting services this way instead of using util classes (e.g., `UserSegmentLocalServiceUtil.java`). You can take advantage of dependency management, and you also won't be tied to a specific implementation of the service.

The next step is to use the service to obtain a list of existing user segments and make it available to your view layer as a request attribute. To do this, add logic to your portlet class that obtains user segments and exposes them as a request attribute, like this:

```
ThemeDisplay themeDisplay = (ThemeDisplay)renderRequest.getAttribute(
    WebKeys.THEME_DISPLAY);

List<UserSegment> userSegments = null;

try {
```

```
    userSegments = _userSegmentLocalService.getUserSegments(
        themeDisplay.getScopeGroupId());
}
catch (Exception e) {
    _log.error(e, e);
}

renderRequest.setAttribute("userSegments", userSegments);

private static final Log _log = LogFactoryUtil.getLog(MyPortlet.class)
```

Notice that the `userSegments` list is populated by calling `UserSegmentLocalService`'s `getUserSegments` method. This service is part of the Content Targeting API.

To finish off this example, some logic needs to be added to your portlet's `view.jsp`:

```
<h2>User Segments</h2>

<ul>

<%
List<UserSegment> userSegments = (List<UserSegment>)request.getAttribute("userSegments");

for (UserSegment userSegment : userSegments) {
%>

    <li><%= userSegment.getName(locale) %></li>

<%
}
%>

</ul>
```

This logic uses the `UserSegment` object to list the existing user segments. That's it! By importing the `UserSegment` and `UserSegmentLocalService` classes into your files, you have direct access to your portal's user segments through the Content Targeting Java API.

Using the Content Targeting JSON API

Suppose you'd like to show a list of existing campaigns in your portlet using the JSON API. You could do this by opening your portlet's `view.jsp` file and using the following code:

```
<h2>Campaigns</h2>

<ul id="<portlet:namespace/>campaigns">
</ul>

<aui:script use="aui-base">
    var campaignsList = A.one('#<portlet:namespace/>campaigns');

    Liferay.Service(
        '/ct.campaign/get-campaigns',
        {
        groupId: '<%= scopeGroupId %>'
        },
        function(response) {
            if (response.length) {
                A.Array.each(response, function(item) {
                    campaignsList.append('<li>' + item.name + '</li>');
                });
            }
        }
    );
</aui:script>
```

Notice that the Content Targeting API is called to retrieve the existing campaigns:

```
...
Liferay.Service(
    '/ct.campaign/get-campaigns',
    {
...
```

Then, each campaign is listed in the `campaignsList` and displayed in your portlet for users to see.

If you'd like to view all the available methods (with examples) exposed in the JSON API by Audience Targeting, you can visit the `/api/jsonws` URL (e.g., `localhost:8080/api/jsonws`). As you can see, accessing the Content Targeting JSON API is just as easy as accessing the related Java API.

You've learned how easy it is to expose the Content Targeting API and use it in your application to unleash its power!

Related Topics

Creating New Audience Rule Types

Targeting Content to Your Audience

Running Service Builder and Understanding the Generated Code

120.2 Creating New Audience Targeting Rule Types

In the Audience Targeting application, a User Segment is defined as a group of users that match a set of rules. Out of the box, Liferay provides several types of rules that are based on characteristics such as age range, gender, location, and so on. You combine these rules to create User Segments. For example, if you want to target probable buyers of a shoe that has a particular style, you might create a User Segment composed of Females over 40 who live in urban areas.

The Audience Targeting app ships with many rules you can use make up User Segments, but it's also extensible. This means that if there isn't a rule that already fits your case, you can create it yourself!

Creating a rule type involves targeting what you want to evaluate. Suppose you own an Outdoor Sporting Goods store. On your website, you'd like to promote goods that are appropriate for the current weather. If a user is from Los Angeles and it's raining the day he or she visits your website, you could show that user new umbrellas. If it's sunny, however, you could show the user sunglasses instead. For this example, your evaluation entity would be weather based on the user's location. To make this work, you'll need to do two things:

1. Retrieve the user's location so you can obtain that location's weather.

2. Let administrators set the value that should be compared with the user's current weather, using a UI component like a selection list of weather options.

With this design, an administrator can set *rainy* as the value for the rule, and the rule could be added to a user segment targeted for rain-related goods. When users visit your site, their user segment assignments come from matching the weather in their current locations with the rule's preset weather value (rainy). On a match, you show rain-related content; otherwise, the user is part of a different User Segment and sees that segment's content, like a promotion for sunglasses.

Now that you have an idea of how to plan your custom rule's development, you'll begin creating one yourself!

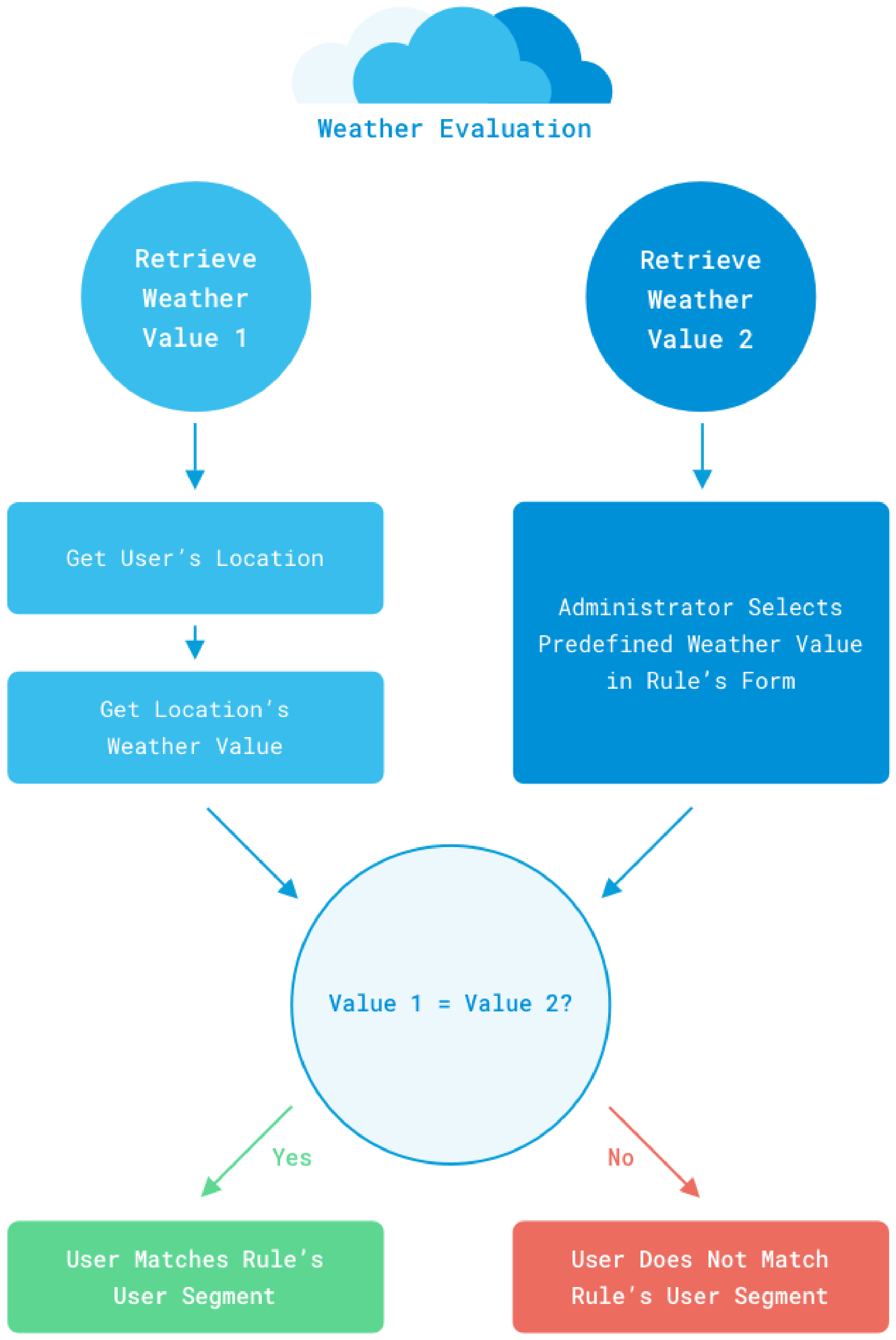

Weather Evaluation
Retrieve Weather Value 1
Retrieve Weather Value 2
Get User’s Location
Get Location’s Weather Value
Administrator Selects Predefined Weather Value in Rule’s Form
Value 1 = Value 2?
Yes
No
User Matches Rule’s User Segment
User Does Not Match Rule’s User Segment

Figure 120.1: This diagram breaks down the evaluation process for the weather rule.

Creating a Custom Rule Type

Adding a new type of rule to the Audience Targeting application is easy. First, you must create a module and ensure it has the necessary Content Targeting API dependencies.

1. Create a module project for deploying a rule. A Blade CLI content-targeting-rule template is available to help you get started quickly. It sets the default configuration for you, and it contains boilerplate code so you can skip the file creation steps and get started right away.

2. Make sure your module specifies the dependencies necessary for an Audience Targeting rule. For example, you should specify the Content Targeting API and necessary Liferay packages. For example, this is the example `build.gradle` file used from a Gradle based rule:

```
dependencies {
    compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.analytics.api", version: "3.0.0"
    compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.anonymous.users.api", version: "2.0.2"
    compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.api", version: "4.0.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.3.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", version: "2.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

   You can learn more about exposing the Content Targeting API in the Accessing the Content Targeting API tutorial. Once you've created your module and specified its dependencies, you'll need to define your rule's behavior. How your rule behaves is controlled by a Java class file that you create.

3. Create a unique package name in the module's `src` directory, and create a new Java class in that package. To follow naming conventions, your class name should begin with the rule name you're creating, and end with *Rule* (e.g., `WeatherRule.java`). Your Java class should implement the `com.liferay.content.targeting.api.model.Rule` interface.

   It is required to implement the Rule interface, but there are `Rule` extension classes that provide helpful utilities that you can extend. For example, your rule can extend the BaseJSPRule class to support generating your rule's UI using JSPs. This tutorial demonstrates implementing the UI using a JSP, and assumes the `Rule` interface is implemented by extending the `BaseJSPRule` class. For more information on choosing a UI for your rule, see the Selecting a UI Technology section.

4. Directly above the class's declaration, insert the following code:

```
@Component(immediate = true, service = Rule.class)
```

   This annotation declares the implementation class of the Component and specifies to immediately start the module once deployed to Liferay DXP.

Now that your Java class is set up, you'll need to define how your rule works by implementing the Rule interface's methods. You'll begin implementing these methods next.

Note: If you're planning on developing a social rule type that classifies users based on their social network profile, it's important to remember that the specific social network's SSO (Single Sign On) must be enabled and configured properly. Visit the Social Rules section for more details.

The first thing you'll define in your weather rule is the view/save lifecycle.

Defining a Rule's View/Save Lifecycle

This section covers how to define a rule's view/save lifecycle. This is when a user applies a rule to a user segment using the User Segment Editor.

In this section, you'll begin defining the weather rule's Java class. This assumes that you followed the instructions above, creating the `WeatherRule` class and extending BaseJSPRule. If you used the `content-targeting-rule` Blade CLI template, your project is already extending `BaseJSPRule` and has a default `view.jsp` file already created.

1. Add the activation and deactivation methods to your class.

   ```
   @Activate
   @Override
   public void activate() {
       super.activate();
   }

   @Deactivate
   @Override
   public void deActivate() {
       super.deActivate();
   }
   ```

   These methods call the super class BaseRule to implement necessary logging and processing for when your rule starts and stops. Make sure to include the @Activate and @Deactivate annotations, which are required.

2. Define the category for the Rule when displayed in the User Segment Editor.

   ```
   @Override
   public String getRuleCategoryKey() {
       return SessionAttributesRuleCategory.KEY;
   }
   ```

   This code puts the weather rule in the Session Attributes category. To put your rule into the appropriate category, use the `getRuleCategoryKey` method to return the category class's key. Available category classes include BehaviourRuleCategory, SessionAttributesRuleCategory, SocialRuleCategory, and UserAttributesRoleCategory.

3. Add the following method:

   ```
   @Override
   protected void populateContext(
       RuleInstance ruleInstance, Map<String, Object> context,
       Map<String, String> values) {

       String weather = "";

       if (!values.isEmpty()) {
           weather = GetterUtil.getString(values.get("weather"));
       }
       else if (ruleInstance != null) {
           weather = ruleInstance.getTypeSettings();
       }

       context.put("weather", weather);
   }
   ```

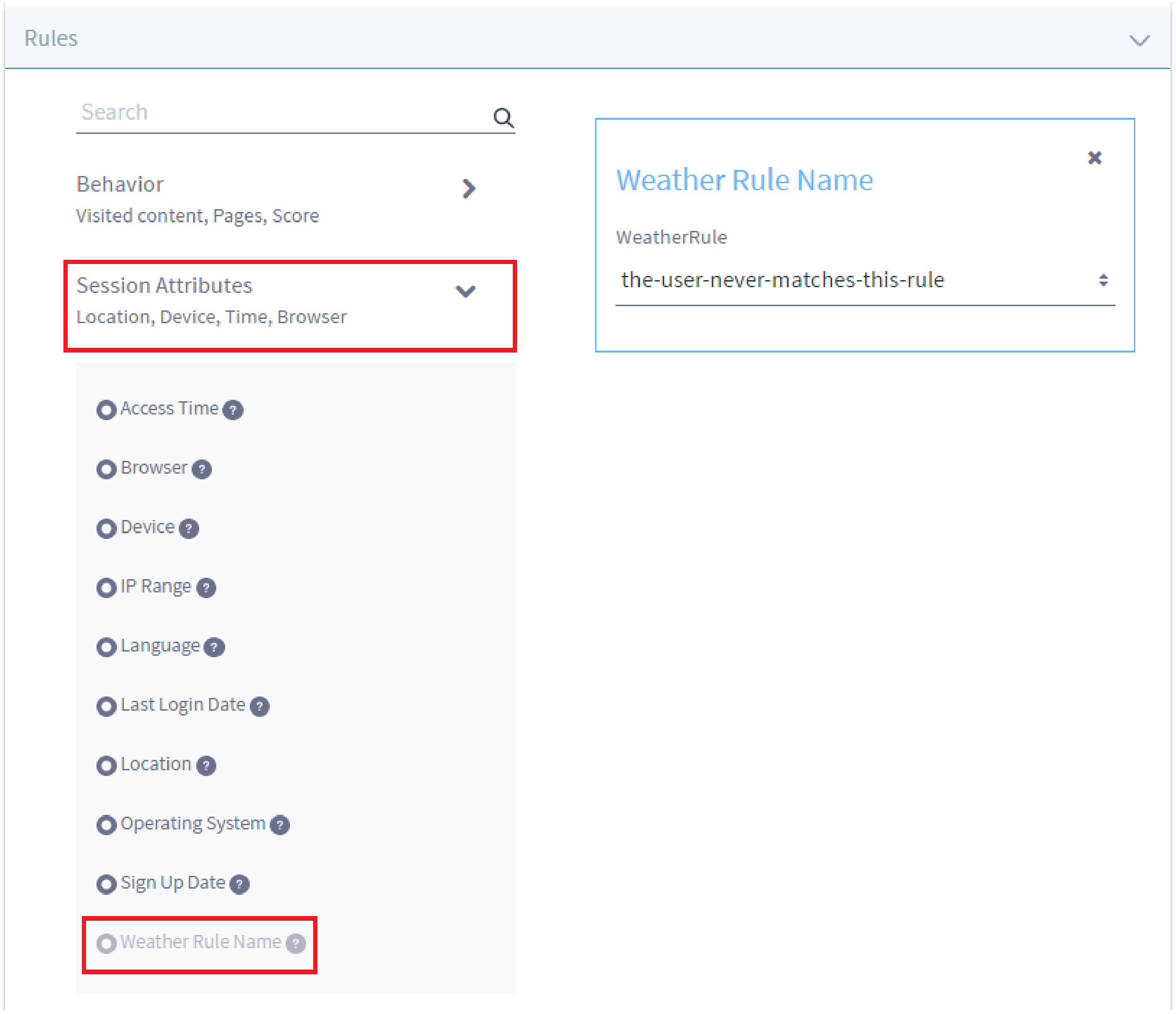


Figure 120.2: This example Weather rule was modified to reside in the Session Attributes category.

To understand what this method accomplishes, you'll need to examine the rule's configuration lifecycle.

When the user opens the User Segment Editor, the render phase begins for the rule. The `getFormHTML(...)` method retrieves the HTML to display. You don't have to worry about implementing this method because it's already implemented in the BaseJSPRule class you're extending. The `getFormHTML` method calls the `populateContext(...)` method.

You'll notice the `populateContext` method is not available in the Rule interface. This is because it's not needed in all cases. It's available by extending the BaseJSPRule class, and you'll need to add more logic to it for the weather rule.

The goal of the `populateContext` method is to generate a map with all the parameters your JSP view needs to render the rule's HTML. This map is stored in the `context` variable, which is pre-populated with basic values in the Portlet logic, and then each rule contributes its specific parameters to it. The `populateContext` method above populates a `weather` context variable with the `weather` values from the `values` map parameter, which is then passed to the JSP.

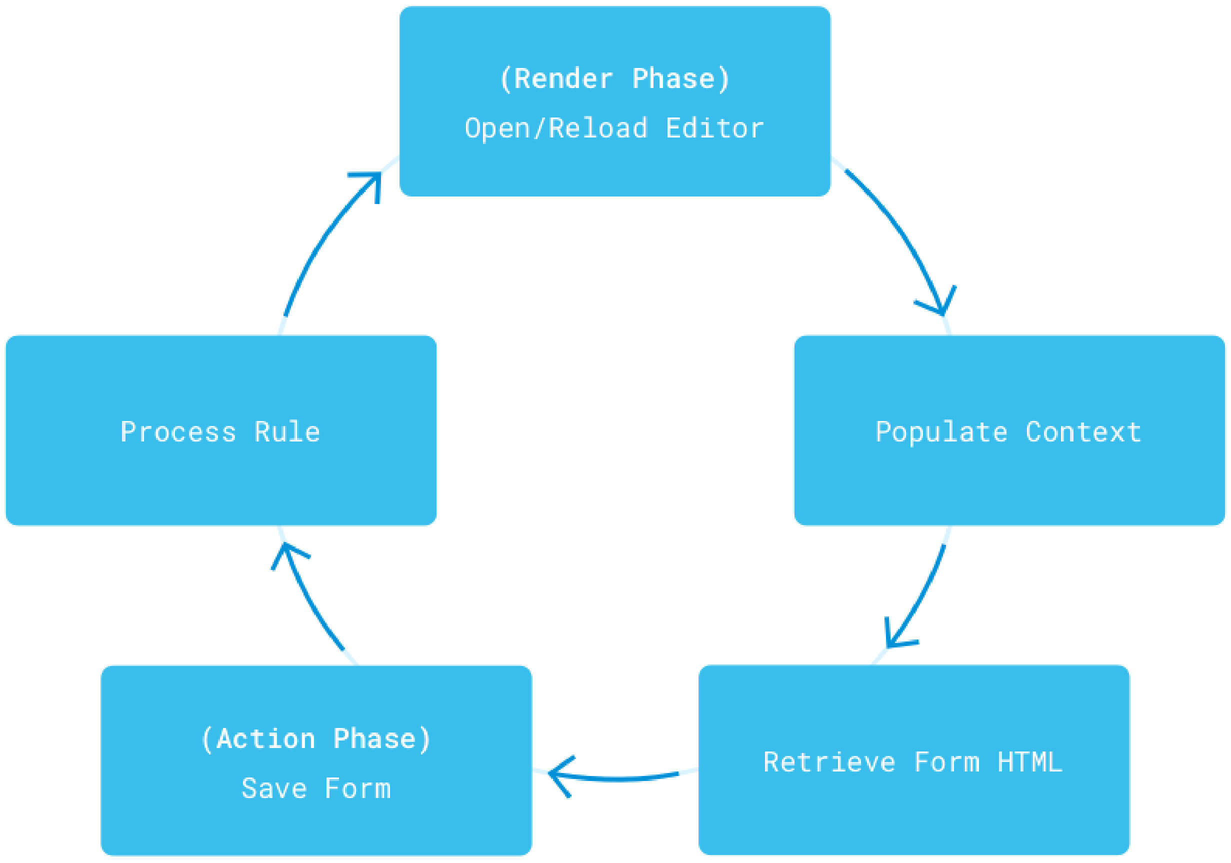


Figure 120.3: An Audience Targeting rule must be configured by the user and processed before it can become part of a User Segment.

For the weather rule, the `populateContext` method accounts for three use cases:

a. The rule was added but has no set values yet. In this case, the default values defined by the developer are injected (e.g., `weather=""`).

b. The rule was added and a value is set, but the request failed to complete (e.g., due to an error). In this case, the `values` parameter of the `populateContext` method contains the values that were intended to be saved, and they are injected so that they are displayed in the rule's view together with the error message.

c. The rule was added and a value was successfully set. In this case, the `values` parameter is empty, and you have to obtain the values from storage that the form should display and inject them in the context so they're displayed in the rule's HTML. The weather rule uses the `typeSettings` field of the rule instance, but complex rules could use services to store values.

You can think of the `populateContext` method as the intermediary between your JSP and your backend code. You can see how to create the weather rule's UI using a JSP by seeing the Defining the Rule's UI section. Once the HTML is successfully retrieved and the user has set the weather value and clicked *Save*, the action phase begins.

4. Add the following method:

   ```
   @Override
   public String processRule(
       PortletRequest portletRequest, PortletResponse portletResponse,
       String id, Map<String, String> values) {

       return values.get("weather");
   }
   ```

   The `processRule(...)` method is invoked when the action phase is initiated. The `values` parameter only contains the value(s) the user added in the form. The logic you could add to a `processRule` method is outlined below.

   a. Obtain the value(s) from the `values` parameter.

   b. (Optional) Validate the data consistency and possible errors. If anything is wrong, throw an InvalidRuleException and prohibit the values from being stored. In the weather rule scenario, when the rule is reloaded after an exception is thrown in the form, case 3b from the previous step occurs.

   c. Return the value to be stored in the rule instance's `typeSettings` field. The `typeSettings` field is managed by the framework in the Rule Instance table. If your rule has its own storage mechanism, then you should call your services in the `processRule` method.

   Once the rule processing ends, the form is reloaded and the lifecycle restarts again. The value(s) selected in the rule are stored and are ready to be accessed once user segment evaluation begins. There are a couple more methods you'll need to add to the `WeatherRule` class before defining the rule's evaluation.

5. Define a way to retrieve the rule's localized summary. In many instances, you can do this by combining keys in the rule's resource bundle with the information stored for the rule. For the weather rule, you can return the rule's type settings, which contains the selected weather condition.

   ```
   @Override
   public String getSummary(RuleInstance ruleInstance, Locale locale) {
       return ruleInstance.getTypeSettings();
   }
   ```

6. Set the servlet context for your rule.

   ```
   @Override
   @Reference(
       target = "(osgi.web.symbolicname=weather)",
       unbind = "-"
   )
   public void setServletContext(ServletContext servletContext) {
       super.setServletContext(servletContext);
   }
   ```

   This is only required for rules extending the BaseJSPRule class. The servlet context must be set for the rule to render its own JSP files. The `setServletContext` method is invoked automatically when the rule module is installed and resolved in Liferay. Make sure the `osgi.web.symbolicname` in the `target` property of the `@Reference` annotation is set to the same value as the `Bundle-SymbolicName` defined in the `bnd.bnd` file of the module.

Next, you'll learn how to evaluate a rule that is configured and saved to a user segment.

Evaluating a Rule

Imagine an administrator has successfully configured and saved your custom rule to his or her user segment. Now what? Your rule needs to fulfill its purpose, which is to evaluate the preset weather value compared to a user's weather value visiting the site. If the user's value matches the preset value (along with the segment's other rules), that user is added to the user segment.

1. You must implement the `evaluate(...)` rule to begin the evaluation process. This method is part of the user segmentation lifecycle. When a page is loaded, Liferay invokes the `evaluate` method of the rule to determine if the current user belongs to the user segment. For the weather rule, add this `evaluate` method:

   ```
   @Override
   public boolean evaluate(
           HttpServletRequest request, RuleInstance ruleInstance,
           AnonymousUser anonymousUser)
       throws Exception {

       String userWeather = getUserWeather(anonymousUser);

       String weather = ruleInstance.getTypeSettings();

       if (Validator.equals(userWeather, weather)) {
           return true;
       }

       return false;
   }
   ```

   You acquire the user's weather by calling the `getUserWeather` method, which you'll define later. Then you get the preset weather value by accessing the rule instance's `typeSettings` parameter. Finally, you compare the two values. If they match, return `true`; otherwise return `false`. Remember that users are only added to User Segments when all the Rules in the User Segment return true.

2. Next, you need to retrieve the user's weather. As you learned earlier, you must access the user's location first. Add the logic below to do this.

   ```
   protected String getCityFromUserProfile(long contactId, long companyId)
       throws PortalException, SystemException {

       List<Address> addresses = AddressLocalServiceUtil.getAddresses(companyId, Contact.class.getName(), contactId);

       if (addresses.isEmpty()) {
           return null;
       }

       Address address = addresses.get(0);

       return address.getCity() + StringPool.COMMA + address.getCountry().getA2();
   }
   ```

   This method retrieves the location by accessing the user's profile information. You could also have used a geo-location service to find this by the user's IP address. Once you have the user's location, you can find the current weather for that location.

3. Add the following method to retrieve a user's weather forecast.

```
protected String getUserWeather(AnonymousUser anonymousUser)
    throws PortalException, SystemException {

    User user = anonymousUser.getUser();

    String city = getCityFromUserProfile(user.getContactId(), user.getCompanyId());

    Http.Options options = new Http.Options();

    String location = HttpUtil.addParameter(API_URL, "q", city);
    location = HttpUtil.addParameter(location, "format", "json");

    options.setLocation(location);

    int weatherCode = 0;

    try {
        String text = HttpUtil.URLtoString(options);

        JSONObject jsonObject = JSONFactoryUtil.createJSONObject(text);

        weatherCode = jsonObject.getJSONArray("weather").getJSONObject(0).getInt("id");
    }
    catch (Exception e) {
        _log.error(e);
    }

    return getWeatherFromCode(weatherCode);
}

private static Log _log = LogFactoryUtil.getLog(WeatherRule.class);
```

This method calls the `getCityFromUserProfile` method to acquire the user's location. Then it retrieves the weather code for that location from a weather service.

4. Set the `API_URL` field to the Open Weather Map's API URL:

```
private static final String API_URL = "http://api.openweathermap.org/data/2.5/weather";
```

   For the weather rule, you can access Open Weather Map's APIs to retrieve the weather code.

5. The last thing is to convert the weather code to a string you can evaluate (e.g., sunny). Add the following method to convert Open Weather Map's weather codes:

```
protected String getWeatherFromCode(int code) {
    if (code == 800 || code == 801) {
        return "sunny";
    }
    else if (code > 801 && code < 805) {
        return "clouds";
    }
    else if (code >= 600 && code < 622) {
        return "snow";
    }
    else if (code >= 500 && code < 532) {
        return "rain";
    }

    return null;
}
```

   All possible weather codes are here.

Excellent! You've implemented the evaluate method and added the necessary logic in your -Rule class to acquire a user's local weather. The weather rule's behavior is defined and complete. The last thing you need to do is create a JSP template.

Defining the Rule's UI

The Java code you've added to this point has assumed that a preset weather value is available for comparing during the evaluation process. To let administrators set that value, you must define a UI so your rule can be configured during the view/save lifecycle. Create a `view.jsp` file in your rule's module (e.g., `/src/main/resources/META-INF/resources/view.jsp`) and add the following logic:

```
<%
Map<String, Object> context = (Map<String, Object>)request.getAttribute("context");

String weather = (String)context.get("weather");
%>

<aui:fieldset>
    <aui:select name="weather" value="<%= weather %>">
        <aui:option label="sunny" value="sunny" />
        <aui:option label="clouds" value="clouds" />
        <aui:option label="snow" value="snow" />
        <aui:option label="rain" value="rain" />
    </aui:select>
</aui:fieldset>
```

The `weather` variable in the `context` map should be set for the weather rule. When the user selects an option, it's passed from the view template to the `populateContext` method.

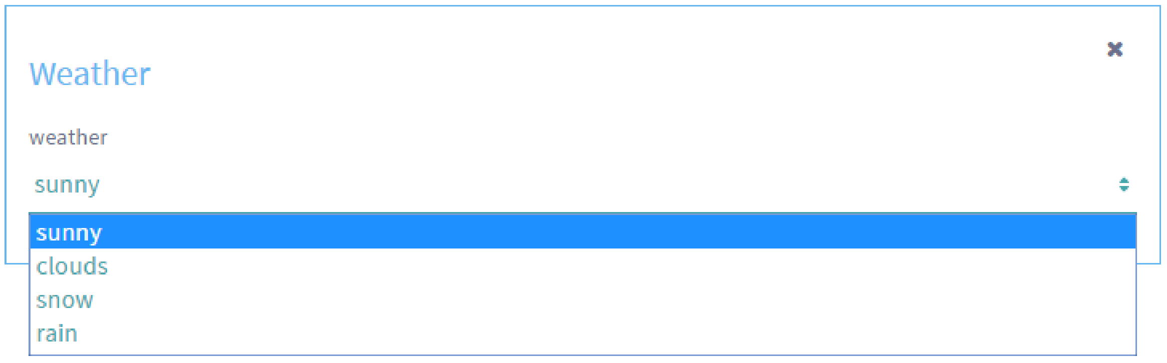


Figure 120.4: The weather rule uses a select drop-down box to set the weather value.

The weather rule uses JSP templates to display the rule's view. Audience Targeting, however, is compatible with any UI technology. Visit the Selecting a UI Technology section for details on how to use other UI technologies like FreeMarker.

Congratulations! You've created the weather rule and can now target users based on their weather conditions. You can view the finished version of the weather rule by downloading its ZIP file.

Now you've created and examined a fully functional rule and have the knowledge to create your own.

Related Topics

Best Practices for Rules
Creating Modules with Blade CLI
Internationalization
Service Builder Persistence

120.3 Tracking User Actions with Audience Targeting

In the Audience Targeting (AT) application, a campaign defines a set of content targeted to specific user segments during a time period. Campaign custom reports allow campaign administrators to learn how users behave in the context of a campaign by monitoring their interaction over different elements of the site. Out of the box, Liferay provides several metrics that are based on entity types that you can track, such as content, forms, links, pages, etc. You can use these metrics to create custom reports. For example, if you want track how many users watch a YouTube video that is published on your site, you might create a custom report with the YouTube Videos metric.

The AT app ships with many metrics you can apply to custom reports, but it's also extensible. This means that if the default metrics available do not fulfill your needs, you can create one yourself!

A metric's development strategy comes down to four choices:

- Entity to Track
- Tracking Mechanism
- Tracking Events
- Differentiation Method

Creating a metric involves targeting what you want to track in a custom report. Suppose you're the owner of a hardware store and you'd like to send emails to your customers notifying them of the store's weekly newsletter. You send the email every week, but you're in the dark about how many customers actually open and read the newsletter. For this example, your entity to track is a newsletter.

To track how many customers view the newsletter, you'll need to create a tracking mechanism. You can provide a custom tracking mechanism (e.g., a servlet) or you can use the ones provided by Audience Targeting. For a newsletter, you could use a transparent image as the tracking mechanism, which would have the *View* tracking event capability. Whenever the image is viewed, the Audience Targeting app computes and stores the information.

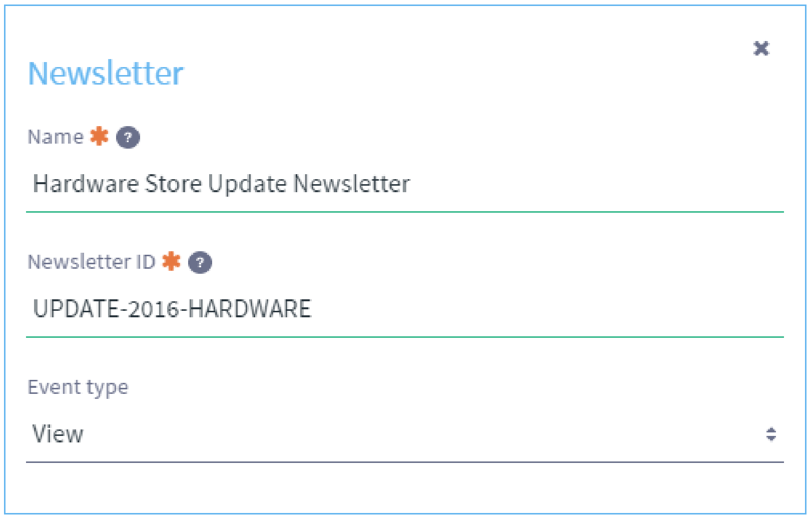
In many cases, a metric can have multiple tracking event options. For example, the YouTube Videos metric provides tracking event options like Buffering, Playing, Paused, Ended, etc. This lets you track different kinds of actions on an entity, providing a more accurate report on user interactions.

Finally, you must assign the metric to an entity. For a newsletter, you could provide a Newsletter ID field that the user could fill in to differentiate newsletters, if there's more than one.

To learn more about how metrics are used in the Audience Targeting application, visit the Defining Metrics section.

For this tutorial, you'll create a newsletter that can track who views it. This process involves defining the view/save lifecycle, which is when a user applies a metric to a report using the Report Editor. Then you'll define its tracking mechanism, tracking event(s), and differentiation method, similar to what was described above.

Now that you have an idea of how to plan your new metric, you'll begin creating one next!


Newsletter

Name *

Hardware Store Update Newsletter

Newsletter ID *

UPDATE-2016-HARDWARE

Event type

View

Figure 120.5: The sample Newsletter metric requires the newsletter name, ID, and event type.

Creating a Metric

Adding a new metric to the Audience Targeting application is easy. First, you must create a module and ensure it has the necessary Content Targeting API dependencies.

1. Create a module project for deploying a metric. A Blade CLI content-targeting-tracking-action template is available to help you get started quickly. It sets the default configuration for you, and it contains boilerplate code so you can skip the file creation steps and get started right away.

2. Make sure your module specifies the dependencies necessary for an Audience Targeting metric. For example, you should specify the Content Targeting API and necessary Liferay packages. For example, this is the example `build.gradle` file used from a Gradle based metric:

```
dependencies {
    compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.analytics.api", version: "3.0.0"
    compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.anonymous.users.api", version: "2.0.2"
    compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.api", version: "4.0.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.3.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", version: "2.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "2.0"
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
}
```

   You can learn more about exposing the Content Targeting API in the Accessing the Content Targeting API tutorial. Once you've created your module and specified its dependencies, you'll need to define your metric's behavior. How your metric behaves is controlled by a Java class file that you create.

3. Create a unique package name in the module's src directory, and create a new Java class in that package. To follow naming conventions, your class name should begin with the metric's name you're creating and end with *TrackingAction* (e.g., `NewsletterTrackingAction.java`). Your Java class should implement the com.liferay.content.targeting.api.model.TrackingAction‘ interface.

   You must implement the TrackingAction interface, but there are `TrackingAction` extension classes that provide helpful utilities that you can extend. For example, your metric can extend the BaseJSP-TrackingAction class to support generating your metric's UI using JSPs. This tutorial demonstrates implementing the UI using a JSP and assumes the TrackingAction interface is implemented by extending the `BaseJSPTrackingAction` class. For more information on choosing a UI for your metric, see the Selecting a UI Technology section.

4. Directly above the class's declaration, insert the following annotation:

   ```
   @Component(immediate = true, service = TrackingAction.class)
   ```

   This declares the Component's implementation class and configures it to start immediately once deployed to Liferay DXP.

Now that your Java class is set up, you'll need to define how your metric works by implementing the TrackingAction interface's methods. You'll begin implementing these methods next.

The first thing you'll define in your newsletter metric is the view/save lifecycle.

Defining a Metric's View/Save Lifecycle

This section covers how to define a metric's view/save lifecycle. This is when a user applies a metric to a report using the Report Editor.

In this section, you'll begin defining the newsletter metric's Java class. This assumes that you followed the instructions above, creating the `NewsletterTrackingAction` class and extending BaseJSPTrackingAction. If you used the `content-targeting-tracking-action` Blade CLI template, your project is already extending `BaseJSPTrackingAction` and a default `view.jsp` file is already created.

1. Add the activation and deactivation methods to your class.

   ```
   @Activate
   @Override
   public void activate() {
       super.activate();
   }

   @Deactivate
   @Override
   public void deActivate() {
       super.deActivate();
   }
   ```

   These methods call the super class BaseTrackingAction to implement necessary logging and processing for when your metric starts and stops. Make sure to include the @Activate and @Deactivate annotations, which are required.

2. Add the following method:

   ```
    @Override
    protected void populateContext(
        TrackingActionInstance trackingActionInstance,
        Map<String, Object> context, Map<String, String> values) {

        String alias = StringPool.BLANK;
        String elementId = StringPool.BLANK;
        String eventType = StringPool.BLANK;

        if (!values.isEmpty()) {
            alias = values.get("alias");
            elementId = values.get("elementId");
   ```

```
        eventType = values.get("eventType");
    }
    else if (trackingActionInstance != null) {
        alias = trackingActionInstance.getAlias();
        elementId = trackingActionInstance.getElementId();
        eventType = trackingActionInstance.getEventType();
    }

    context.put("alias", alias);
    context.put("elementId", elementId);
    context.put("eventType", eventType);
    context.put("eventTypes", getEventTypes());
}
```

To understand what this method accomplishes, you should understand the metric's configuration lifecycle.

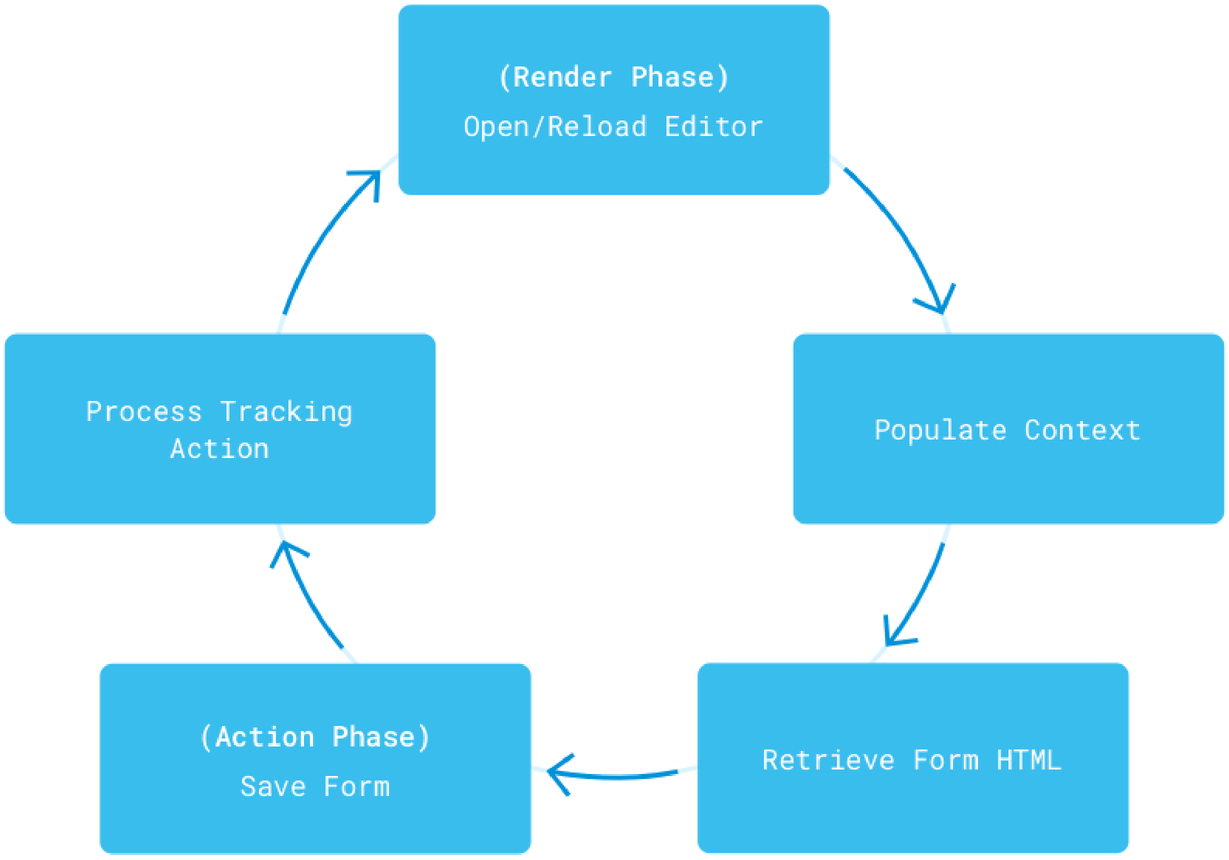


Figure 120.6: An Audience Targeting metric must be configured by the user and processed before it can become part of a Report.

When the user opens the Report Editor, the render phase begins for the metric. The `getFormHTML(...)` method retrieves the HTML to display. You don't have to worry about implementing this method because it's already implemented in the BaseJSPTrackingAction class you're extending. The `getFormHTML` method calls the `populateContext(...)` method.

You'll notice the `populateContext` method is not available in the TrackingAction interface. This is because it's not needed in all cases. It's available by extending the `BaseJSPTrackingAction` class, and you'll need to add more logic to it for the newsletter metric.

The goal of the `populateContext` method is to generate a map with all the parameters your JSP view needs to render the metric's HTML. This map is stored in the `context` variable, which is pre-populated with basic values in the Portlet logic, and then each metric contributes its specific parameters to it. The `populateContext` method above populates the `alias`, `elementId`, `eventType`, and `eventTypes` context variables with the adjacent values from the `values` map parameter, which is then passed to the JSP.

For the newsletter metric, the `populateContext` method accounts for three use cases:

a. The metric was added but has no set values yet. In this case, the default values defined by the developer are injected (e.g., `alias=""`).

b. The metric was added and a value is set, but the request failed to complete (e.g., due to an error). In this case, the `values` parameter of the `populateContext` method contains the values that were intended to be saved, and they are injected so that they are displayed in the metric's view together with the error message.

c. The metric was added and a value was successfully set. In this case, the `values` parameter is empty, and you have to obtain the values from storage that the form should display and inject them in the context so they're displayed in the metric's HTML. The newsletter metric stores values in the metric's instance, but complex metrics could use services to store values.

You can think of the `populateContext` method as the intermediary between your JSP and your backend code. You can see how to create the newsletter metric's UI using a JSP by skipping to the Defining the Metric's UI section. Once the HTML is successfully retrieved and the user has set the newsletter's values and clicked *Save*, the action phase begins.

3. Once the action phase begins, AT processes the tracking action (metric). The `processTrackingAction(...)` method takes the values from the metric's UI form and stores them in the corresponding fields of the `trackingActionInstance`. Since the BaseTrackingAction class provides a default implementation of this method that returns `null`, the `NewsletterTrackingAction` class does not need to implement it.

   If you need to process any custom fields in your metric, you should override this method. If you want your custom values to be stored in the `typeSettings` field of the `trackingActionInstance`, return their value instead of `null`.

```
**Note:** For more complex cases, you can create your own services to store
your metric's information to a database. You should invoke your services'
update logic within the `processTrackingAction` method. For more information
on creating services, see the
[Service Builder](/docs/7-0/tutorials/-/knowledge_base/t/service-builder)
tutorials.
```

```
Once the metric processing ends, the form is reloaded and the lifecycle
restarts again. The value(s) specified in the metric are stored and are
ready to be accessed once the report generation begins. Next, you must set
the event types that the newsletter metric should evaluate.
```

4. Add the following method and private field:

   ```
   @Override
   public List<String> getEventTypes() {
       return ListUtil.fromArray(_EVENT_TYPES);
   }

   private static final String[] _EVENT_TYPES = {"view"};
   ```

   This specifies that your newsletter metric only tracks who views the newsletter.

5. Define a way to retrieve the metric's localized summary. In many instances, you can do this by combining keys in the metric's resource bundle with the information stored for the metric. For the newsletter metric, you can provide information about the ID of the newsletter being tracked, which is stored in the `alias` field of the `trackingActionInstance` object.

   ```
   @Override
   public String getSummary(
       String summary = LanguageUtil.format(
           locale, "tracking-newsletter-x",
           new Object[] {trackingActionInstance.getAlias()});

       return summary;
   }
   ```

6. Set the servlet context for your metric.

   ```
   @Override
   @Reference(
       target = "(osgi.web.symbolicname=newsletter)",
       unbind = "-"
   )
   public void setServletContext(ServletContext servletContext) {
       super.setServletContext(servletContext);
   }
   ```

   This is only required for metrics extending the `BaseJSPTrackingAction` class. The servlet context must be set for the metric to render its own JSP files. The `setServletContext` method is invoked automatically when the metric module is installed and resolved in Liferay. Make sure the `osgi.web.symbolicname` in the `target` property of the `@Reference` annotation is set to the same value as the `Bundle-SymbolicName` defined in the `bnd.bnd` file of the module.

Next, you'll define a tracking mechanism for your metric to use.

Using a Tracking Mechanism

Imagine an administrator has successfully configured and saved your custom metric to his or her report. Now what? Your metric needs to fulfill its purpose, which is to track the `view` event type for the defined newsletter. To do this, you must define a tracking mechanism. For your newsletter, you'll use a transparent image as the tracking mechanism, which would have the *View* tracking event capability. Whenever the image is viewed, the newsletter metric computes and stores the information.

For the newsletter metric, you'll use a tracking mechanism provided by the Audience Targeting app.

1. You must set the analytics processor that the Content Targeting API provides for tracking events. Add the following method and private field:

```
@Reference
protected void setAnalyticsProcessor(AnalyticsProcessor analyticsProcessor) {
    _analyticsProcessor = analyticsProcessor;
}

private AnalyticsProcessor _analyticsProcessor;
```

The analytics processor is a module of the Audience Targeting Analytics system. It contains a servlet to track analytics from Liferay pages (views, clicks, etc.) and an API to leverage this tracking mechanism. In the `setAnalyticsProcesoor(...)` method, you're obtaining a reference of the current analytics processor to build the URL used to generate a transparent image. All you have to do is insert the generated URL into your newsletter's HTML, and the transparent image tracks who reads it. Everything is processed by the default Audience Targeting Analytics system automatically.

Now that you've obtained a reference of the analytics processor, you need to add logic for generating the appropriate tracking URL.

2. Replace the `populateContext` method with the updated method:

```
@Override
protected void populateContext(
    TrackingActionInstance trackingActionInstance,
    Map<String, Object> context, Map<String, String> values) {

    String alias = StringPool.BLANK;
    String elementId = StringPool.BLANK;
    String eventType = StringPool.BLANK;
    String trackImageHTML = StringPool.BLANK;

    if (!values.isEmpty()) {
        alias = values.get("alias");
        elementId = values.get("elementId");
        eventType = values.get("eventType");
    }
    else if (trackingActionInstance ≠ null) {
        alias = trackingActionInstance.getAlias();
        elementId = trackingActionInstance.getElementId();
        eventType = trackingActionInstance.getEventType();

        String trackImageURL = _analyticsProcessor.getTrackingURL(
            trackingActionInstance.getCompanyId(), 0, 0, "", 0,
            Campaign.class.getName(),
            new long[] {trackingActionInstance.getCampaignId()},
            trackingActionInstance.getElementId(), "view", "");

        trackImageHTML = "<img alt=\"\" src=\"" + trackImageURL + "\" />";
    }

    context.put("alias", alias);
    context.put("elementId", elementId);
    context.put("eventType", eventType);
    context.put("eventTypes", getEventTypes());
    context.put("trackImageHTML", trackImageHTML);
}
```

This updated method creates a new variable named `trackImageHTML`, retrieves a tracking URL using the analytics processor, and then populates the `trackImageHTML` context variable. When creating a new metric, the transparent image's URL field is not present in the metric's form. When the metric is initially saved, however, the URL is generated using the analytics processor and is available for copying.

Excellent! You've obtained the analytics processor and can create the transparent image tracking mechanism. The newsletter metric's behavior is defined and complete. The last thing you need to do is create a JSP template.

Defining the Metric's UI

The Java code you've added to this point has assumed that there are three configurable fields for your newsletter metric:

- *Name:* used in reports that count the number of times a metric has been triggered. This is also known as the newsletter's alias.
- *Newsletter ID:* used to differentiate between newsletters.
- *Event Type:* used to differentiate several actions on the same newsletter, such as opening the newsletter or clicking on a link.

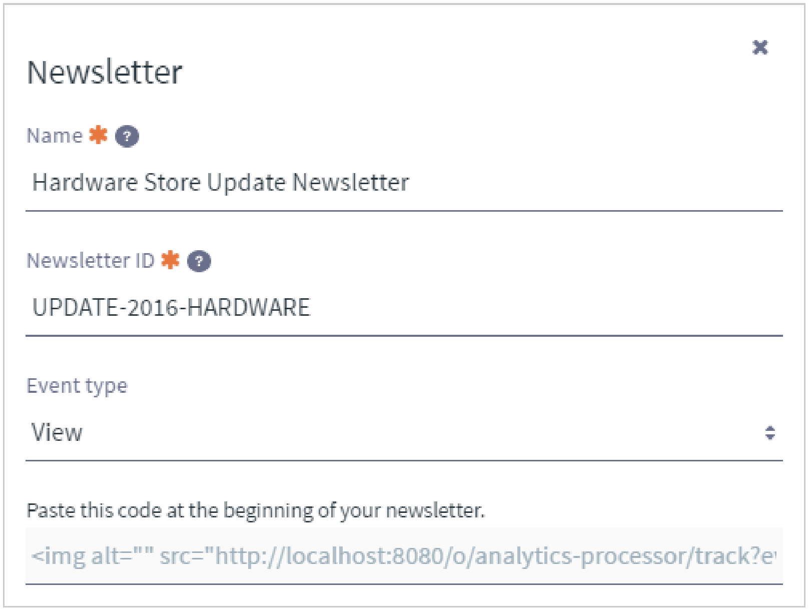
To let administrators set these values, you must define a UI so your metric can be configured during the view/save lifecycle. Remember that you must also define a field to display the generated transparent image's URL. Create a `view.jsp` file in your metric's module (e.g., `/src/main/resources/META-INF/resources/view.jsp`) and add the following logic:

```
<%
Map<String, Object> context = (Map<String, Object>)request.getAttribute("context");

String alias = (String)context.get("alias");
String elementId = (String)context.get("elementId");
String eventType = (String)context.get("eventType");
List<String> eventTypes = (List<String>)context.get("eventTypes");
String trackImageHTML = (String)context.get("trackImageHTML");
%>

<aui:input helpMessage="name-help" label="name" name='<%= ContentTargetingUtil.GUID_REPLACEMENT + "alias" %>' type="text" value="<%= alias %>">
    <aui:validator name="required" />
</aui:input>

<aui:input helpMessage="enter-the-id-of-the-newsletter-to-be-tracked" label="newsletter-id" name='<%= ContentTargetingUtil.GUID_REPLACEMENT + "elementId" %>
    <aui:validator name="required" />
</aui:input>

<c:if test="<%= ListUtil.isNotEmpty(eventTypes) %>">
    <aui:select label="event-type" name='<%= ContentTargetingUtil.GUID_REPLACEMENT + "eventType" %>'>

        <%
        for (String curEventType : eventTypes) {
        %>

            <aui:option label="<%= curEventType %>" selected="<%= curEventType.equals(eventType) %>" value="<%= curEventType %>" />

        <%
        }
        %>

    </aui:select>
</c:if>

<c:if test="<%= !Validator.isBlank(trackImageHTML) %>">
    <span class="h5">
        <liferay-ui:message key="paste-this-code-at-the-beginning-of-your-newsletter" />
    </span>
    <label for='<%= renderResponse.getNamespace() + ContentTargetingUtil.GUID_REPLACEMENT + "trackImageHTML" %>' key="paste-this-code-
at-the-beginning-of-your-newsletter" /></label>
```

```
    <liferay-ui:input-resource id='<%= renderResponse.getNamespace() + ContentTargetingUtil.GUID_REPLACEMENT + "trackImageHTML" %>' url="<%= trackImageHTML
</c:if>
```

First you instantiate the context variable and its attributes you configured in your Java class's `populateContext` method. Then you specify the appropriate fields Name, Newsletter ID, and Event Type. Finally, you present the generated transparent image URL.

Notice that the input field names in the JSP are prefixed with `ContentTargetingUtil.GUID_REPLACEMENT`. This prefix is required for multi-instantiable metrics, which are metrics that return true in the `isInstantiable` method of their `-TrackingAction` class and can be added more than once to the Metrics form.


Newsletter

Name: Hardware Store Update Newsletter

Newsletter ID: UPDATE-2016-HARDWARE

Event type: View

Paste this code at the beginning of your newsletter.

<img alt="" src="http://localhost:8080/o/analytics-processor/track?e

Figure 120.7: Once you've saved the metric, you can copy the generated transparent image URL into your newsletter's HTML to track who views it.

Congratulations! You've created the newsletter metric and can now track whether users viewed a newsletter. You can test if the metric is working by copying the generated tracking image HTML into an email HTML editor, sending it, and opening it as if it were an actual newsletter. Then open the custom report containing the newsletter metric and select *Update Report*. A chart and table with the newsletter's view count is shown.

You can view the finished version of the newsletter metric by downloading its ZIP file.

Now you've created and examined a fully functional metric and have the knowledge to create your own.

Related Topics

Creating Modules with Blade CLI

Defining Metrics

Audience Targeting Metrics

120.4 Best Practices for Metrics

In this tutorial, you'll learn about best practices to keep in mind when creating Audience Targeting Metrics. Before going through some best practices, you should understand the four components you can specify for a metric:

- *Metric Behavior*
- *Tracking Mechanism*
- *UI for Configuration (optional)*
- *Language Keys (optional)*

You discuss metric behavior and its UI configuration in great detail in the Tracking User Actions with Audience Targeting tutorial. To learn more about language keys and how to create, use, and generate them, visit the Internationalization tutorials.

Audience Targeting gives you the option to choose whatever frontend technology you like. In the next section, you'll learn how to use your preferred technology for displaying content in Audience Targeting metrics.

Selecting a UI Technology

Since 7.0, JSP is the preferred technology for Audience Targeting extension views. FreeMarker views, however, are still supported through their respective base classes (e.g., `BaseFreemarkerTrackingAction`). If you're interested in using a technology besides JSP or FreeMarker to implement your UI, you can add a method `getFormHTML` to your `-TrackingAction` class. Here's an example of implementing the `getFormHTML` method:

```
@Override
public String getFormHTML(
    TrackingActionInstance trackingActionInstance,
    Map<String, Object> context, Map<String, String> values) {

    String content = "";

    try {
        populateContext(trackingActionInstance, context, values);

        content = ContentTargetingContextUtil.includeJSP(
            _servletContext, getFormTemplatePath(), context);
    }
    catch (Exception e) {
        _log.error(
            "Error while processing form template " +
                getFormTemplatePath(),
            e);
    }

    return content;
}
```

The `getFormHTML` is used to retrieve the HTML created by the technology you choose, and to return it as a string that is viewable from your metric's form. If you plan, therefore, on using an alternative to JSP or FreeMarker, you must override this method by creating and modifying it in your `-TrackingAction` class.

Related Topics

Tracking User Actions with Audience Targeting

Internationalization

Service Builder Persistence

120.5 Best Practices for Rules

In this tutorial, you'll learn about best practices to keep in mind when creating Audience Targeting Rules. Before going through some best practices, you should understand the three components you can specify for a rule:

- *Rule Behavior*
- *UI for Configuration (optional)*
- *Language Keys (optional)*

You discuss rule behavior and its UI configuration in great detail in the Creating New Audience Targeting Rule Types tutorial. To learn more about language keys and how to create, use, and generate them, visit the Internationalization tutorials.

Audience Targeting gives you the option to choose whatever frontend technology you like. In the next section, you'll learn how to use your preferred technology for displaying content in Audience Targeting rules.

Selecting a UI Technology

Since 7.0, JSP is the preferred technology for Audience Targeting extension views. FreeMarker views, however, are still supported through their respective base classes (e.g., `BaseFreemarkerRule`). If you're interested in using a technology besides JSP or FreeMarker to implement your UI, you can add a method `getFormHTML` to your `-Rule` class. Here's an example of implementing the `getFormHTML` method:

```
@Override
public String getFormHTML(
    RuleInstance ruleInstance, Map<String, Object> context,
    Map<String, String> values) {

    String content = "";

    try {
        populateContext(ruleInstance, context, values);

        content = ContentTargetingContextUtil.parseTemplate(
            getClass(), getFormTemplatePath(), context);
    }
    catch (Exception e) {
        _log.error(
            "Error while processing template " + getFormTemplatePath(), e);
    }

    return content;
}
```

The `getFormHTML` is used to retrieve the HTML created by the technology you choose, and to return it as a string that is viewable from your rule's form. If you plan, therefore, on using an alternative to JSP or FreeMarker, you must override this method by creating and modifying it in your `-Rule` class.

Other Best Practices

Here are some things to consider as you implement and deploy Audience Targeting rules:

- As an alternative to storing complex information in the `typeSettings` field, which is managed by the framework in the Rule Instance table, you may want to consider persisting to a database by using Service Builder, which is supported for Rule plugins.

- If you deploy your rule into a production environment, you may want to consider adding your values to the cache (e.g., weather in different locations), since obtaining the same value on every request is very inefficient and could result in slowing down your portal. For example, when the `evaluate` method is called, you could obtain the current user ID, current user's weather forecast, and the time at which the user first visited the page. Then you could evaluate the rule only when the cached time is over three hours old. This would prevent the rule from evaluating every time the user visited the page. This is best done using services.

- You can override the `BaseJSPRule.deleteData` method in your `-Rule`, so that it deletes any data associated with the rule that is currently being deleted.

- If your rule handles data or references to data that can be staged (e.g., a reference to a page or web content article), you may need to override the `BaseRule.exportData` and `BaseRule.importData` methods, to manage the content properly.

Related Topics

Creating New Audience Targeting Rule Types
    Internationalization
    Service Builder Persistence

Chapter 121

Customizable Web Applications

One of the great strengths of Liferay DXP is the sheer number of out of the box applications. This gives it great flexibility in what it can do, which is why Liferay DXP is used to run many different kinds of websites around the world. For example, if you want users to add Blog posts, you can configure the Blogs portlet to handle those requests.

But really, that's technology from the last decade. What if you could define a particular function that users might want to perform and let Liferay DXP choose an available installed app to perform that function? That way, if users want to Blog, and you've installed your own custom-developed app for blogging instead of Liferay's, the Liferay DXP instance can just use yours instead?

With Liferay DXP 7, you can do just that. You can request an app based on an entity and action type. Processing the entity type and action, Liferay uses an available portlet that can handle the request. This increases the flexibility and modularity of using portlets in Liferay DXP.

121.1 Providing Portlets to Manage Requests

In this tutorial, you'll learn how to declare an entity type and action for a desired function, and you'll create a module that finds the correct application (portlet) to use based on those given parameters.

Specifying a Desired Portlet Behavior

To find the portlet you need for your particular request, you'll use the *Portlet Providers* framework. The first thing you'll need to do is call the PortletProviderUtil class and request the framework find a portlet suitable for the current task. You can request the portlet ID or portlet URL, depending on what you prefer. Here's an example declaration:

```
String portletId = PortletProviderUtil.getPortletId(
    "com.liferay.portlet.trash.model.TrashEntry", PortletProvider.Action.VIEW);
```

This declaration expects two parameters: the class name of the entity type you want the portlet to handle and the type of action. The above code requests a portlet ID for a portlet that can view Recycle Bin entries.

There are five different kinds of actions supported by the Portlet Providers framework: `ADD`, `BROWSE`, `EDIT`, `PREVIEW`, and `VIEW`. Find the portlet ID or portlet URL (depending on your needs), and specify the entity type and action you want the portlet to handle.

Note: The `getPortletURL` methods require an additional `HttpServletRequest` or `PortletRequest` parameter, depending on which you use. Make sure to account for this additional parameter when using the `getPortletURL` method.

You've successfully requested the portlet ID/portlet URL of a portlet that matches your entity and action type. The portal, however, is not yet configured to handle this request. You'll need to create a module that can find the correct portlet to handle the request.

1. Create an OSGi module.

2. Create a unique package name in the module's `src` directory and create a new Java class in that package. To follow naming conventions, name the class based on the element type and action type, followed by *PortletProvider* (e.g., `LanguageEntryViewPortletProvider`). The class should extend the `BasePortletProvider` class and implement the appropriate portlet provider interface based on the action type you chose your portlet to handle (e.g., ViewPortletProvider, BrowsePortletProvider, etc.).

3. Directly above the class's declaration, insert the following annotation:

   ```
   @Component(
       immediate = true,
       property = {"model.class.name=CLASS_NAME"},
       service = INTERFACE.class
   )
   ```

   The `property` element should match the element type you specified in your `getPortletID`/`getPortletURL` declaration (e.g., `com.liferay.portal.kernel.servlet.taglib.ui.LanguageEntry`). Also, your `service` element should match the interface you're implementing (e.g., `ViewPortletProvider.class`). You can view an example of a similar `@Component` annotation in the RolesSelectorEditPortletProvider class:

   ```
   @Component(
       immediate = true,
       property = {"model.class.name=com.liferay.portal.kernel.model.UserGroupRole"},
       service = EditPortletProvider.class
   )
   ```

4. In some cases, a default portlet is already in place to handle the entity and action type requested. To override the default portlet with a custom portlet, you can assign your portlet a higher service ranking. You can do this by setting the following property in your `@Component` declaration:

   ```
   property= {"service.ranking:Integer=10"}
   ```

   Make sure to replace the integer with a number that is ranked higher than the portlet being used by default.

5. Specify the methods you'd like to implement. Make sure to retrieve the portlet ID/portlet URL that should be provided when this service is called.

Lastly, generate the module's JAR file and deploy it to your portal instance. Now a portlet that can handle the entity and action type you specified is used when requesting a portlet ID/URL. You can now specify portlet usage without hardcoding a specific portlet!

Related Topics

Portlets
    Embedding Portlets in Themes
    Customizing Liferay Services

Part II

Developer Reference

Chapter 122

Development Reference

Here you'll find reference documentation for Liferay DXP, Liferay Screens, Liferay Faces, and technologies related to you as a third-party developer.

The different types of reference docs you'll find in this section of the Liferay Developer Network are as follows:

- Descriptions of Java and JavaScript APIs, CSS, tags and tag libraries, and XML DTDs
- Write ups on the latest Screenlets for Liferay Screens
- Breaking changes
- Cheat sheets and tips on
  - Plugin anatomy
  - Design patterns
  - Tools
  - Adapting to new APIs

Liferay's reference docs are at your fingertips.

122.1 Java APIs

Here you'll find Javadoc for Liferay DXP and Liferay DXP apps.

7.0 Java APIs

This table links you to the 7.0 API modules. Their root location is here. (Opens New Window) The reference doc JAR is available here. (Opens New Window)

Core:

com.liferay.portal.kernel (portal-kernel): (Opens New Window) for developing applications on Liferay DXP

com.liferay.util.bridges (util-bridges): (Opens New Window) for using various non-proprietary computing languages, frameworks, and utilities on Liferay DXP

com.liferay.util.java (util-java): (Opens New Window) for using various Java-related frameworks and utilities on Liferay DXP com.liferay.util.slf4j (util-slf4j): (Opens New Window) for using the Simple Logging Facade for Java (SLF4J)

com.liferay.portal.impl (portal-impl): (Opens New Window) refer to this only if you are an advanced Liferay developer that needs a deeper understanding of 7.0's implementation in order to contribute to it

Liferay DXP App Java APIs

This table links you to Liferay DXP application APIs. Their root location is here. (Opens New Window)

Collaboration (Opens New Window) (JAR) (Opens New Window)
com.liferay.blogs.api
com.liferay.blogs.item.selector.api
com.liferay.bookmarks.api
com.liferay.comment.api
com.liferay.document.library.api
com.liferay.document.library.repository.cmis.api
com.liferay.flags.api
com.liferay.invitation.invite.members.api
com.liferay.item.selector.api
com.liferay.item.selector.criteria.api
com.liferay.mentions.api
com.liferay.message.boards.api
com.liferay.microblogs.api
com.liferay.ratings.api
com.liferay.social.activity.api
com.liferay.social.privatemessaging.api
com.liferay.wiki.api
Forms & Workflow (Opens New Window) (JAR) (Opens New Window)
com.liferay.calendar.api
com.liferay.dynamic.data.lists.api
com.liferay.dynamic.data.mapping.api
com.liferay.polls.api
com.liferay.portal.reports.engine.api
com.liferay.portal.rules.engine.api
com.liferay.portal.workflow.kaleo.api
com.liferay.portal.workflow.kaleo.definition.api
com.liferay.portal.workflow.kaleo.runtime.api
Foundation (Opens New Window) (JAR) (Opens New Window)
com.liferay.contacts.api
com.liferay.frontend.image.editor.api
com.liferay.map.api
com.liferay.mobile.device.rules.api
com.liferay.password.policies.admin.api
com.liferay.portal.background.task.api

com.liferay.portal.lock.api
com.liferay.portal.scripting.api
com.liferay.portal.security.audit.api
com.liferay.portal.security.exportimport.api

com.liferay.portal.security.service.access.policy.api
com.liferay.portal.settings.api
com.liferay.roles.admin.api
com.liferay.user.groups.admin.api
com.liferay.users.admin.api
com.liferay.users.admin.demo.data.creator.api
com.liferay.xstream.configurator.api
Web Experience (Opens New Window) (JAR) (Opens New Window)
com.liferay.application.list.api
com.liferay.exportimport.api
com.liferay.journal.api
com.liferay.journal.item.selector.api
com.liferay.layout.item.selector.api
com.liferay.layout.prototype.api
com.liferay.layout.set.prototype.api
com.liferay.portlet.configuration.icon.locator.api
com.liferay.portlet.configuration.toolbar.contributor.locator.api
com.liferay.product.navigation.control.menu.api
com.liferay.site.api
com.liferay.site.item.selector.api
com.liferay.staging.api
For help finding API modules for specific common classes, see 7.0 API Modules.
For help finding module attributes and configuring dependencies, see Configuring Dependencies.

122.2 Taglibs

Here you'll find tag library documentation for the Liferay DXP, Liferay DXP apps, and Liferay Faces.

7.0 Taglibs

Util Taglibs (Opens New Window)
aui
liferay-portlet
portlet
liferay-security
liferay-theme
liferay-ui
liferay-util

Liferay DXP App Taglibs

Application List:
liferay-application-list (Opens New Window)
Assets:
liferay-asset (Opens New Window)
liferay-trash (Opens New Window)
Import, Export, & Staging:
liferay-staging (Opens New Window)

Item Selector:
liferay-item-selector (Opens New Window)
Product Navigation:
liferay-product-navigation (Opens New Window)
Sites:
liferay-layout (Opens New Window)
liferay-site-navigation (Opens New Window)
Social:
liferay-flags (Opens New Window)
For help finding module attributes and configuring dependencies, see Configuring Dependencies.

Faces Taglibs

Faces 3.2 Taglibs: the latest version of Liferay Faces JSF tag docs in View Declaration Language (VDL) format. VDL docs for all versions of Liferay Faces are available here.

JavaScript and CSS

Lexicon: The web implementation of Liferay's Lexicon Experience Language. Lexicon is a system for building applications in and outside of Liferay DXP, designed to be fluid and extensible, as well as provide a consistent and documented API.

Bootstrap: The base CSS library onto which Lexicon is built. Liferay DXP uses Bootstrap natively and all of its CSS classes and JavaScript features are available within portlets, templates, and themes.

AlloyUI: Liferay includes AlloyUI and all of its JavaScript APIs are available within portlets, templates and themes.

Descriptor Definitions

DTDs: Describes the XML files used in configuring Liferay DXP apps, 7.0 plugins, and @product-ver@.

Chapter 123

Liferay API Modules

The following table maps commonly used Liferay DXP components to their API modules and key classes. You configure dependencies on the component API modules to use them.

123.1 API Modules Table

Component	Classes	Module Symbolic Name (Artifact ID)
Application List	PanelApp	com.liferay.application.list.api
	PanelCategory	
	PanelEntry	
Background Tasks	BackgroundTask[Local]ServiceUtil	com.liferay.portal.background.task.api
Blogs	BlogsEntry[Local]ServiceUtil	com.liferay.blogs.api
Bookmarks	BookmarksEntry[Local]ServiceUtil	com.liferay.bookmarks.api
	BookmarksFolder[Local]ServiceUtil	
Calendar	Calendar[Local]ServiceUtil	com.liferay.calendar.api
	CalendarBooking[Local]ServiceUtil	
	CalendarImporter	
	CalendarNotificationTemplate[Local]ServiceUtil	
	CalendarResource[Local]ServiceUtil	
Comment	Comment	com.liferay.comment.api
	DiscussionComment	
Contacts	Entry	com.liferay.contacts.api
Document Library	DLFileEntry[Local]ServiceUtil	com.liferay.document.library.api
	DLContent[Local]ServiceUtil	
	DLFileEntryType[Local]ServiceUtil	
	DLFileVersion[Local]ServiceUtil	
	DLFolder[Local]ServiceUtil	
	DLSyncEvent[Local]ServiceUtil	

Component	Classes	Module Symbolic Name (Artifact ID)
Dynamic Data Lists	DDLRecord[Local]ServiceUtil	com.liferay.dynamic.data.lists.api
DDLRecordSet[Local]ServiceUtil		
DDLRecordVersion[Local]ServiceUtil		
Dynamic Data Mapping	DDMContent[Local]ServiceUtil	com.liferay.dynamic.data.mapping.api
DDMStructure[Local]ServiceUtil		
DDMStorageLink[Local]ServiceUtil		
DDMStructureLayout[Local]ServiceUtil		
DDMStructureLink[Local]ServiceUtil		
DDMStructureVersion[Local]ServiceUtil		
DDMTemplate[Local]ServiceUtil		
DDMTemplateLink[Local]ServiceUtil		
DDMTemplateVersion[Local]ServiceUtil		
Export / Import	ExportImportConfiguration	com.liferay.exportimport.api
StagingServiceUtil[Local]ServiceUtil		
Flags	FlagsEntryServiceUtil	com.liferay.flags.api
Invitation	MemberRequest[Local]ServiceUtil	com.liferay.invitation.invite.members.api
Item Selector	ItemSelector	com.liferay.item.selector.api
Item Selector Criteria	FileEntryItemSelectorReturnType	com.liferay.item.selector.criteria.api
UploadableFileReturnType		
URLItemSelectorReturnType		
UUIDItemSelectorReturnType		
Lock	Lock	com.liferay.portal.lock.api
Map	MapProvider	com.liferay.map.api
Marketplace	App	com.liferay.marketplace.api
Module		
Mentions	MentionsNotifier	com.liferay.mentions.api
MentionsUserFinder		
MentionsUtil		
Message Boards	MBMessage[Local]ServiceUtil	com.liferay.message.boards.api
MBCategory[Local]ServiceUtil		
MBThread[Local]ServiceUtil		
MBDiscussion[Local]ServiceUtil		
Microblogs	MicroblogsEntry[Local]ServiceUtil	com.liferay.microblogs.api
Mobile Device Rules	MDRAction[Local]ServiceUtil	com.liferay.mobile.device.rules.api
MDRRule[Local]ServiceUtil		
MDRRuleGroup[Local]ServiceUtil		
MDRRuleGroupInstance[Local]ServiceUtil		
Polls	PollsChoice[Local]ServiceUtil	com.liferay.polls.api
PollsQuestion[Local]ServiceUtil		
PollsVote[Local]ServiceUtil		
Portal Access Policy	SAPEntry[Local]ServiceUtil	com.liferay.portal.security.service.access.policy.api

Component	Classes	Module Symbolic Name (Artifact ID)
Portal Settings	PortalSettings	com.liferay.portal.settings.api
Portlet Configuration	PortletConfigurationIcon	com.liferay.portlet.configuration.icon.locator.api
	PortletToolbar	com.liferay.portlet.configuration.toolbar.contributor.locator.api
Private Messaging	UserThread[Local]ServiceUtil	com.liferay.social.privatemessaging.api
Product Navigation	ProductNavigationControlMenuCategory	com.liferay.product.navigation.control.menu.api
	ProductNavigationControlMenuEntry	
Ratings	RatingsEntry[Local]ServiceUtil	com.liferay.ratings.api
Reports Engine	RulesEngine	reports.engine.api
	RulesLanguage	
	Fact	
	Query	
Screens	ScreensAssetEntryServiceUtil	com.liferay.screens.api
	ScreensDDLRecordServiceUtil	
	ScreensJournalArticleServiceUtil	
Security Audit	AuditEvent	com.liferay.portal.security.audit.api
	AuditEventManager	
	AuditConfiguration	
Security Import / Export	UserExporter	com.liferay.portal.security.exportimport.api
	UserImporter	
	UserOperation	
Shopping Cart	ShoppingCart[Local]ServiceUtil	com.liferay.shopping.api
	ShoppingCategory[Local]ServiceUtil	
	ShoppingCoupon[Local]ServiceUtil	
	ShoppingItem[Local]ServiceUtil	
	ShoppingItemPrice[Local]ServiceUtil	
	ShoppingOrder[Local]ServiceUtil	
	ShoppingOrderItem[Local]ServiceUtil	
Site	GroupSearchProvider	com.liferay.site.api
Social Networking	MeetupsEntry[Local]ServiceUtil	com.liferay.social.networking.api
	MeetupsRegistration[Local]ServiceUtil	
	WallEntry[Local]ServiceUtil	
Staging	Staging[Local]ServiceUtil	com.liferay.staging.api
Web Content	JournalArticle[Local]ServiceUtil	com.liferay.journal.api
	JournalFolder[Local]ServiceUtil	
	JournalArticleImage[Local]ServiceUtil	
	JournalFeed[Local]ServiceUtil	
Wiki	WikiNode[Local]ServiceUtil	com.liferay.wiki.api
	WikiPage[Local]ServiceUtil	
XStream Configurator	XStreamConfigurator	com.liferay.xstream.configurator.api

For reference documentation on these APIs and others, see the the app reference docs at Liferay DXP and the Liferay DXP core reference docs at @platform-ref@/7.0-latest.

Related Articles

Configuring Dependencies
Development Reference
Liferay Upgrade Planner

Chapter 124

Portlet Descriptor to OSGi Service Property Map

This section describes the mapping of portlet XML descriptor values to OSGi service properties that can be used when publishing OSGi Portlets.

OSGi services can contain properties in their definitions. Using OSGi service properties makes dealing with configuration concerns simple and cohesive. These properties are typically represented as key-value pairs or, more generally, as a Map-like object.

Portlet spec property keys are prefixed by:

```
javax.portlet.
```

Liferay property keys are prefixed by:

```
com.liferay.portlet.
```

The mappings essentially flatten what is found in the XML descriptor, sticking relatively closely to the original naming in order to have a memorable relationship with those definitions.

JSR-168 & JSR-286 Descriptor Mappings

Note: XPath notation derived from the **Portlet XSD** 4 is used in this document for simplicity.

---

portlet.xml XPath | OSGi Portlet Service Property| /portlet-app/portlet/description|javax.portlet.description=<Stri
/portlet-app/portlet/portlet-name 6|javax.portlet.name=<String> 6| /portlet-app/portlet/display-
name|javax.portlet.display-name=<String>| /portlet-app/portlet/portlet-class|1| /portlet-app/portlet/init-
param/name|javax.portlet.init-param.<name>=<value>| /portlet-app/portlet/expiration-cache|javax.portlet.expiration-
cache=<int>| /portlet-app/portlet/cache-scope|not supported| /portlet-app/portlet/supports/mime-
type|javax.portlet.mime-type=<mime-type>| /portlet-app/portlet/supports/portlet-mode|javax.portlet.portlet-
mode=<mime-type>;<portlet-mode>[,<portlet-mode>]*| /portlet-app/portlet/supports/window-state|javax.portlet.window-
state=<mime-type>;<window-state>[,<window-state>]*| /portlet-app/portlet/supported-locale|not
supported| /portlet-app/portlet/resource-bundle|javax.portlet.resource-bundle=<String>| /portlet-
app/portlet/portlet-info/title|javax.portlet.info.title=<String>| /portlet-app/portlet/portlet-
info/short-title|javax.portlet.info.short-title=<String>| /portlet-app/portlet/portlet-info/keywords|javax.portlet.
/portlet-app/portlet/portlet-preferences|javax.portlet.preferences=<String>ORjavax.portlet.preferences=classpath:

/portlet-app/portlet/security-role-ref|javax.portlet.security-role-ref=<String>[,<String>]2| /portlet-app/portlet/supported-processing-event/name|javax.portlet.supported-processing-event=<String>ORjavax.por processing-event=<String>;<QName>2| /portlet-app/portlet/supported-publishing-event|javax.portlet.supported-publishing-event=<String>ORjavax.portlet.supported-publishing-event=<String>;<QName>2| /portlet-app/portlet/supported-public-render-parameter|javax.portlet.supported-public-render-parameter=<String>2| /portlet-app/portlet/container-runtime-option|not supported| /portlet-app/custom-portlet-mode|not supported| /portlet-app/custom-window-state|not supported| /portlet-app/user-attribute|not supported| /portlet-app/security-constraint|not supported| /portlet-app/resource-bundle|not supported| /portlet-app/filter/portlet-app/filter-mapping|3| /portlet-app/default-namespace|not supported| /portlet-app/event-definition|not supported| /portlet-app/filter/init-param/name|javax.portlet.init-param.<name>=<value>| /portlet-app/public-render-parameter|not supported| /portlet-app/listener|not supported?javax.portlet.PortletURLGenerationListener?| /portlet-app/container-runtime-option|not supported|

Liferay Descriptor Mappings

Liferay Display

---

liferay-display.xml XPath | OSGi Portlet Service Property| /display/category\[@name\]|com.liferay.portlet.display-category=<value>|

Liferay Portlet

Note: XPath notation derived from **Liferay Portlet** 5 is used in this document for simplicity.

---

liferay-portlet.xml XPath | OSGi Liferay Portlet Service Property| /liferay-portlet-app/portlet/portlet-name|not supported| /liferay-portlet-app/portlet/icon|com.liferay.portlet.icon=<String>| /liferay-portlet-app/portlet/virtual-path|com.liferay.portlet.virtual-path=<String>| /liferay-portlet-app/portlet/struts-path|com.liferay.portlet.struts-path=<String>| /liferay-portlet-app/portlet/parent-struts-path|com.liferay.portlet.parent-struts-path=<String>| /liferay-portlet-app/portlet/configuration-path|com.liferay.portlet.configuration-path=<String>| /liferay-portlet-app/portlet/configuration-action-class|3| /liferay-portlet-app/portlet/indexer-class|3| /liferay-portlet-app/portlet/open-search-class|3| /liferay-portlet-app/portlet/scheduler-entry|3| /liferay-portlet-app/portlet/portlet-url-class|3| /liferay-portlet-app/portlet/friendly-url-mapper-class|3| /liferay-portlet-app/portlet/friendly-url-mapping|com.liferay.portlet.friendly-url-mapping=<String>| /liferay-portlet-app/portlet/friendly-url-routes|com.liferay.portlet.friendly-url-routes=<String>| /liferay-portlet-app/portlet/url-encoder-class|3| /liferay-portlet-app/portlet/portlet-data-handler-class|3| /liferay-portlet-app/portlet/staged-model-data-handler-class|3| /liferay-portlet-app/portlet/template-handler|3| /liferay-portlet-app/portlet/portlet-layout-listener-class|3| /liferay-portlet-app/portlet/poller-processor-class|3| /liferay-portlet-app/portlet/pop-message-listener-class|3| /liferay-portlet-app/portlet/social-activity-interpreter-class|3| /liferay-portlet-app/portlet/social-request-interpreter-class|3| /liferay-portlet-app/portlet/social-interactions-configuration|3| /liferay-portlet-app/portlet/user-notification-definitions|not supported| /liferay-portlet-app/portlet/user-notification-handler-class|3| /liferay-portlet-app/portlet/webdav-storage-token|not supported| /liferay-portlet-app/portlet/webdav-storage-class|3| /liferay-portlet-app/portlet/xml-rpc-method-class|3| /liferay-portlet-app/portlet/control-panel-entry-category|com.liferay.portlet.control-

panel-entry-category=<String>| /liferay-portlet-app/portlet/control-panel-entry-weight|com.liferay.portlet.control-
panel-entry-weight=<double>| /liferay-portlet-app/portlet/control-panel-entry-class|3| /liferay-
portlet-app/portlet/asset-renderer-factory|3| /liferay-portlet-app/portlet/atom-collection-
adapter|3| /liferay-portlet-app/portlet/custom-attributes-display|3| /liferay-portlet-app/portlet/ddm-
display|3| /liferay-portlet-app/portlet/permission-propagator|3| /liferay-portlet-app/portlet/trash-
handler|3| /liferay-portlet-app/portlet/workflow-handler|3| /liferay-portlet-app/portlet/preferences-
company-wide|com.liferay.portlet.preferences-company-wide=<boolean>| /liferay-portlet-app/portlet/preferences-
unique-per-layout|com.liferay.portlet.preferences-unique-per-layout=<boolean>| /liferay-portlet-
app/portlet/preferences-owned-by-group|com.liferay.portlet.preferences-owned-by-group=<boolean>|
/liferay-portlet-app/portlet/use-default-template|com.liferay.portlet.use-default-template=<boolean>|
/liferay-portlet-app/portlet/show-portlet-access-denied|com.liferay.portlet.show-portlet-access-
denied=<boolean>| /liferay-portlet-app/portlet/show-portlet-inactive|com.liferay.portlet.show-
portlet-inactive=<boolean>| /liferay-portlet-app/portlet/action-url-redirect|com.liferay.portlet.action-
url-redirect=<boolean>| /liferay-portlet-app/portlet/restore-current-view|com.liferay.portlet.restore-
current-view=<boolean>| /liferay-portlet-app/portlet/maximize-edit|com.liferay.portlet.maximize-
edit=<boolean>| /liferay-portlet-app/portlet/maximize-help|com.liferay.portlet.maximize-help=<boolean>|
/liferay-portlet-app/portlet/pop-up-print|com.liferay.portlet.pop-up-print=<boolean>| /liferay-
portlet-app/portlet/layout-cacheable|com.liferay.portlet.layout-cacheable=<boolean>| /liferay-
portlet-app/portlet/instanceable|com.liferay.portlet.instanceable=<boolean>| /liferay-portlet-
app/portlet/remoteable|com.liferay.portlet.remoteable=<boolean>| /liferay-portlet-app/portlet/scopeable|com.liferay
/liferay-portlet-app/portlet/single-page-application|com.liferay.portlet.single-page-application=<boolean>|
/liferay-portlet-app/portlet/user-principal-strategy|com.liferay.portlet.user-principal-strategy=<String>|
/liferay-portlet-app/portlet/private-request-attributes|com.liferay.portlet.private-request-
attributes=<boolean>| /liferay-portlet-app/portlet/private-session-attributes|com.liferay.portlet.private-
session-attributes=<boolean>| /liferay-portlet-app/portlet/autopropagated-parameters|com.liferay.portlet.autopropag
parameters=<String>2| /liferay-portlet-app/portlet/requires-namespaced-parameters|com.liferay.portlet.requires-
namespaced-parameters=<boolean>| /liferay-portlet-app/portlet/action-timeout|com.liferay.portlet.action-
timeout=<int>| /liferay-portlet-app/portlet/render-timeout|com.liferay.portlet.render-timeout=<int>|
/liferay-portlet-app/portlet/render-weight|com.liferay.portlet.render-weight=<int>| /liferay-
portlet-app/portlet/ajaxable|com.liferay.portlet.ajaxable=<boolean>| /liferay-portlet-app/portlet/header-
portal-css|com.liferay.portlet.header-portal-css=<String>2| /liferay-portlet-app/portlet/header-
portlet-css|com.liferay.portlet.header-portlet-css=<String>2| /liferay-portlet-app/portlet/header-
portal-javascript|com.liferay.portlet.header-portal-javascript=<String>2| /liferay-portlet-
app/portlet/header-portlet-javascript|com.liferay.portlet.header-portlet-javascript=<String>2|
/liferay-portlet-app/portlet/footer-portal-css|com.liferay.portlet.footer-portal-css=<String>2|
/liferay-portlet-app/portlet/footer-portlet-css|com.liferay.portlet.footer-portlet-css=<String>2|
/liferay-portlet-app/portlet/footer-portal-javascript|com.liferay.portlet.footer-portal-javascript=<String>2|
/liferay-portlet-app/portlet/footer-portlet-javascript|com.liferay.portlet.footer-portlet-javascript=<String>2|
/liferay-portlet-app/portlet/css-class-wrapper|com.liferay.portlet.css-class-wrapper=<String>|
/liferay-portlet-app/portlet/facebook-integration|com.liferay.portlet.facebook-integration=<String>|
/liferay-portlet-app/portlet/add-default-resource|com.liferay.portlet.add-default-resource=<boolean>|
/liferay-portlet-app/portlet/system|com.liferay.portlet.system=<boolean>| /liferay-portlet-
app/portlet/active|com.liferay.portlet.active=<boolean>| /liferay-portlet-app/portlet/include|not
supported|

- [1] Portlets are registered as concrete objects.

- [2] Multiples of these properties may be used. This results in an array of values.
- [3] This type is registered as an OSGi service.
- [4] http://java.sun.com/xml/ns/portlet/portlet-app_2_0.xsd
- [5] http://www.liferay.com/dtd/liferay-portlet-app_7_0_0.dtd
- [6] Liferay DXP creates each portlet's ID based on the portlet's name (i.e., the `portlet-name` descriptor in `liferay-portlet.xml` or the `javax.portlet.name` OSGi service property). Dashes, periods, and spaces are allowed in the portlet name, but they and all other JavaScript unsafe characters are stripped from the name value that's used for the portlet ID. Therefore, make your portlet name unique in light of the characters that are removed. Otherwise, if you try to deploy a portlet whose ID is the same as a portlet that's already deployed, your portlet deployment fails and Liferay DXP logs a message like this:

  ```
  Portlet id [portletId] is already in use
  ```

Chapter 125

Classes Moved from portal-service.jar

To leverage the benefits of modularization in 7.0, many classes from former Liferay Portal 6 JAR file portal-service.jar have been moved into application and framework API modules. The table below provides details about these classes and the modules they've moved to. Package changes and each module's symbolic name (artifact ID) are listed, to facilitate configuring dependencies.

Classes Moved from portal-service to modules

This information was generated based on comparing classes in liferay-portal-src-6.2-ce-ga6 to classes in liferay-portal-src-7.0-ce-ga1.

Class

Package

Module Symbolic Name (Artifact ID)

ActionHandler

Old: com.liferay.portal.kernel.mobile.device.rulegroup.action New: com.liferay.mobile.device.rules.action

com.liferay.mobile.device.rules.api

ActionHandlerManager

Old: com.liferay.portal.kernel.mobile.device.rulegroup New: com.liferay.mobile.device.rules.action

com.liferay.mobile.device.rules.api

ActionHandlerManagerUtil

Old: com.liferay.portal.kernel.mobile.device.rulegroup New: com.liferay.mobile.device.rules.action

com.liferay.mobile.device.rules.api

ActionTypeException

Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception

com.liferay.mobile.device.rules.api

AlternateKeywordQueryHitsProcessor

Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.hits

com.liferay.portal.search

ArticleContentException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception

com.liferay.journal.api

ArticleContentSizeException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception

com.liferay.journal.api

ArticleCreateDateComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay.journal.api
ArticleDisplayDateComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay.journal.api
ArticleDisplayDateException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
ArticleExpirationDateException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
ArticleIDComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay.journal.api
ArticleIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
ArticleModifiedDateComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay.journal.api
ArticleReviewDateComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay.journal.api
ArticleReviewDateException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
ArticleSmallImageNameException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
ArticleSmallImageSizeException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
ArticleTitleComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay.journal.api
ArticleTitleException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
ArticleVersionComparator
Old: com.liferay.portlet.journal.util.comparator New: com.liferay.journal.util.comparator
com.liferay.journal.api
ArticleVersionException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
AssetPublisherUtil
Old: com.liferay.portlet.assetpublisher.util New: com.liferay.asset.publisher.web.util
com.liferay.asset.publisher.web

AuditMessageProcessor
Old: com.liferay.portal.kernel.audit New: com.liferay.portal.security.audit
com.liferay.portal.security.audit.api
AverageStatistics
Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics
com.liferay.portal.monitoring
BackgroundTaskLocalService
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay.portal.background.task.api
BackgroundTaskLocalServiceUtil
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay.portal.background.task.api
BackgroundTaskLocalServiceWrapper
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay.portal.background.task.api
BackgroundTaskModel
Old: com.liferay.portal.model New: com.liferay.portal.background.task.model
com.liferay.portal.background.task.api
BackgroundTaskPersistence
Old: com.liferay.portal.service.persistence New: com.liferay.portal.background.task.service.persistence
com.liferay.portal.background.task.api
BackgroundTaskService
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay.portal.background.task.api
BackgroundTaskServiceUtil
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay.portal.background.task.api
BackgroundTaskServiceWrapper
Old: com.liferay.portal.service New: com.liferay.portal.background.task.service
com.liferay.portal.background.task.api
BackgroundTaskSoap
Old: com.liferay.portal.model New: com.liferay.portal.background.task.model
com.liferay.portal.background.task.api
BackgroundTaskUtil
Old: com.liferay.portal.service.persistence New: com.liferay.portal.background.task.service.persistence
com.liferay.portal.background.task.api
BackgroundTaskWrapper
Old: com.liferay.portal.model New: com.liferay.portal.background.task.model
com.liferay.portal.background.task.api
BaseCmisRepository
Old: com.liferay.portal.kernel.repository.cmis New: com.liferay.document.library.repository.cmis
com.liferay.document.library.repository.cmis
BaseCmisSearchQueryBuilder
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
BaseDDLExporter

Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.exporter.impl
com.liferay.dynamic.data.lists.service
BaseDDMDisplay
Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util
com.liferay.dynamic.data.mapping.api
BaseFieldRenderer
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay.dynamic.data.mapping.api
BaseScriptingExecutor
Old: com.liferay.portal.kernel.scripting New: com.liferay.portal.scripting
com.liferay.portal.scripting
BaseStatistics
Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics
com.liferay.portal.monitoring
BaseStorageAdapter
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay.dynamic.data.mapping.api
BillingCityException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BillingCountryException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BillingEmailAddressException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BillingFirstNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BillingLastNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BillingPhoneException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BillingStateException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BillingStreetException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BillingZipException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
BlockingPortalCache
Old: com.liferay.portal.kernel.cache New: com.liferay.portal.cache
com.liferay.portal.cache

BookmarksEntry
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
BookmarksEntryFinder
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay.bookmarks.api
BookmarksEntryLocalService
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksEntryLocalServiceUtil
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksEntryLocalServiceWrapper
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksEntryModel
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
BookmarksEntryPersistence
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay.bookmarks.api
BookmarksEntryService
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksEntryServiceUtil
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksEntryServiceWrapper
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksEntrySoap
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
BookmarksEntryUtil
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay.bookmarks.api
BookmarksEntryWrapper
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
BookmarksFolder
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
BookmarksFolderConstants
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
BookmarksFolderFinder
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay.bookmarks.api

BookmarksFolderLocalService
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksFolderLocalServiceUtil
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksFolderLocalServiceWrapper
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksFolderModel
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
BookmarksFolderPersistence
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay.bookmarks.api
BookmarksFolderService
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksFolderServiceUtil
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksFolderServiceWrapper
Old: com.liferay.portlet.bookmarks.service New: com.liferay.bookmarks.service
com.liferay.bookmarks.api
BookmarksFolderSoap
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
BookmarksFolderUtil
Old: com.liferay.portlet.bookmarks.service.persistence New: com.liferay.bookmarks.service.persistence
com.liferay.bookmarks.api
BookmarksFolderWrapper
Old: com.liferay.portlet.bookmarks.model New: com.liferay.bookmarks.model
com.liferay.bookmarks.api
ByteArrayReportResultContainer
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
CCExpirationException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CCNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CCNumberException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CCTypeException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api

CMISBetweenExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISConjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISContainsExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISContainsNotExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISContainsValueExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISCriterion
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISDisjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISFullTextConjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISInFolderExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISInTreeExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISJunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis
CMISNotExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis

CMISParameterValueUtil

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay.document.library.repository.cmis

CMISRepositoryHandler

Old: com.liferay.portal.kernel.repository.cmis New: com.liferay.document.library.repository.cmis

com.liferay.document.library.repository.cmis

CMISRepositoryUtil

Old: com.liferay.portal.kernel.repository.cmis New: com.liferay.document.library.repository.cmis.internal

com.liferay.document.library.repository.cmis.impl

CMISSearchQueryBuilder

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay.document.library.repository.cmis

CMISSimpleExpression

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay.document.library.repository.cmis

CMISSimpleExpressionOperator

Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search

com.liferay.document.library.repository.cmis

CartMinOrderException

Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception

com.liferay.shopping.api

CartMinQuantityException

Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception

com.liferay.shopping.api

CollatedSpellCheckHitsProcessor

Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.hits

com.liferay.portal.search

CompoundSessionIdServletRequest

Old: com.liferay.portal.kernel.servlet.filters.compoundsessionid New: com.liferay.portal.compound.session.id

com.liferay.portal.compound.session.id

Condition

Old: com.liferay.portlet.dynamicdatamapping.storage.query New: com.liferay.portal.workflow.kaleo.definition

com.liferay.portal.workflow.kaleo.definition.api

ContactConverterKeys

Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap

com.liferay.portal.security.ldap

ContentException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

ContentNameException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api
ContextClassloaderReportDesignRetriever
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
CountStatistics
Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics
com.liferay.portal.monitoring
CouponActiveException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponCodeException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponDateException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponDescriptionException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponDiscountException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponEndDateException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponLimitCategoriesException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponLimitSKUsException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponMinimumOrderException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
CouponStartDateException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
DDL
Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.util
com.liferay.dynamic.data.lists.api
DDLExporter
Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.exporter
com.liferay.dynamic.data.lists.api
DDLExporterFactory

Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.exporter
com.liferay.dynamic.data.lists.api
DDLRecord
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordConstants
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordFinder
Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence
com.liferay.dynamic.data.lists.api
DDLRecordLocalService
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordLocalServiceUtil
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordModel
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordPersistence
Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence
com.liferay.dynamic.data.lists.api
DDLRecordService
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordServiceUtil
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordServiceWrapper
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordSet
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordSetConstants
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordSetFinder
Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence
com.liferay.dynamic.data.lists.api
DDLRecordSetLocalService

Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordSetLocalServiceUtil
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordSetLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordSetModel
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordSetPersistence
Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence
com.liferay.dynamic.data.lists.api
DDLRecordSetService
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordSetServiceUtil
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordSetServiceWrapper
Old: com.liferay.portlet.dynamicdatalists.service New: com.liferay.dynamic.data.lists.service
com.liferay.dynamic.data.lists.api
DDLRecordSetSoap
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordSetUtil
Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence
com.liferay.dynamic.data.lists.api
DDLRecordSetWrapper
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordSoap
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordUtil
Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence
com.liferay.dynamic.data.lists.api
DDLRecordVersion
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordVersionModel
Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model
com.liferay.dynamic.data.lists.api
DDLRecordVersionPersistence

Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence

com.liferay.dynamic.data.lists.api

DDLRecordVersionSoap

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay.dynamic.data.lists.api

DDLRecordVersionUtil

Old: com.liferay.portlet.dynamicdatalists.service.persistence New: com.liferay.dynamic.data.lists.service.persistence

com.liferay.dynamic.data.lists.api

DDLRecordVersionVersionComparator

Old: com.liferay.portlet.dynamicdatalists.util.comparator New: com.liferay.dynamic.data.lists.util.comparator

com.liferay.dynamic.data.lists.api

DDLRecordVersionWrapper

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay.dynamic.data.lists.api

DDLRecordWrapper

Old: com.liferay.portlet.dynamicdatalists.model New: com.liferay.dynamic.data.lists.model

com.liferay.dynamic.data.lists.api

DDM

Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util

com.liferay.dynamic.data.mapping.api

DDMContent

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay.dynamic.data.mapping.api

DDMContentLocalService

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay.dynamic.data.mapping.api

DDMContentLocalServiceUtil

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay.dynamic.data.mapping.api

DDMContentLocalServiceWrapper

Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay.dynamic.data.mapping.api

DDMContentModel

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay.dynamic.data.mapping.api

DDMContentPersistence

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay.dynamic.data.mapping.api

DDMContentSoap

Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model

com.liferay.dynamic.data.mapping.api

DDMContentUtil

Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay.dynamic.data.mapping.api
DDMContentWrapper
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMDisplay
Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util
com.liferay.dynamic.data.mapping.api
DDMDisplayRegistry
Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util
com.liferay.dynamic.data.mapping.api
DDMIndexer
Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util
com.liferay.dynamic.data.mapping.api
DDMStorageLink
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStorageLinkLocalService
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStorageLinkLocalServiceUtil
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStorageLinkLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStorageLinkModel
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStorageLinkPersistence
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMStorageLinkSoap
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStorageLinkUtil
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMStorageLinkWrapper
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStructureConstants
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStructureFinder
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence

com.liferay.dynamic.data.mapping.api
DDMStructureLinkLocalService
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStructureLinkLocalServiceUtil
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStructureLinkLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStructureLinkModel
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStructureLinkPersistence
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMStructureLinkSoap
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStructureLinkUtil
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMStructureLinkWrapper
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStructureLocalService
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStructureLocalServiceUtil
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStructureLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStructureModel
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStructurePersistence
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMStructureService
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStructureServiceUtil
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay.dynamic.data.mapping.api
DDMStructureServiceWrapper
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMStructureSoap
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMStructureUtil
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMStructureWrapper
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMTemplateConstants
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMTemplateFinder
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMTemplateHelper
Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util
com.liferay.dynamic.data.mapping.api
DDMTemplateLocalService
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMTemplateLocalServiceUtil
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMTemplateLocalServiceWrapper
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMTemplateModel
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMTemplatePersistence
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMTemplateService
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMTemplateServiceUtil
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service
com.liferay.dynamic.data.mapping.api
DDMTemplateServiceWrapper
Old: com.liferay.portlet.dynamicdatamapping.service New: com.liferay.dynamic.data.mapping.service

com.liferay.dynamic.data.mapping.api
DDMTemplateSoap
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMTemplateUtil
Old: com.liferay.portlet.dynamicdatamapping.service.persistence New: com.liferay.dynamic.data.mapping.service.persistence
com.liferay.dynamic.data.mapping.api
DDMTemplateWrapper
Old: com.liferay.portlet.dynamicdatamapping.model New: com.liferay.dynamic.data.mapping.model
com.liferay.dynamic.data.mapping.api
DDMUtil
Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util
com.liferay.dynamic.data.mapping.api
DDMXML
Old: com.liferay.portlet.dynamicdatamapping.util New: com.liferay.dynamic.data.mapping.util
com.liferay.dynamic.data.mapping.api
DefaultAttributesTransformer
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.internal
com.liferay.portal.security.ldap
DefaultMessageBus
Old: com.liferay.portal.kernel.messaging New: com.liferay.portal.messaging.internal
com.liferay.portal.messaging
DefaultSingleDestinationMessageSender
Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender
com.liferay.portal.messaging
DefaultSingleDestinationSynchronousMessageSender
Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender
com.liferay.portal.messaging
DefaultSynchronousMessageSender
Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender
com.liferay.portal.messaging
DestinationStatisticsManager
Old: com.liferay.portal.kernel.messaging.jmx New: com.liferay.portal.messaging.internal.jmx
com.liferay.portal.messaging
DestinationStatisticsManagerMBean
Old: com.liferay.portal.kernel.messaging.jmx New: com.liferay.portal.messaging.internal.jmx
com.liferay.portal.messaging
DirectSynchronousMessageSender
Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender
com.liferay.portal.messaging
DummyContext
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.dummy
com.liferay.portal.security.ldap
DummyDirContext
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.dummy
com.liferay.portal.security.ldap
DuplicateArticleIdException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
DuplicateArticleImageIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
DuplicateCouponCodeException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
DuplicateFeedIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
DuplicateItemSKUException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
DuplicateLDAPServerNameException
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap
com.liferay.portal.security.ldap
DuplicateNodeNameException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
DuplicatePageException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
DuplicateRuleGroupInstanceException
Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception
com.liferay.mobile.device.rules.api
DuplicateVoteException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api
EntryNameComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay.bookmarks.api
EntryPriorityComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay.bookmarks.api
EntryURLComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay.bookmarks.api
EntryVisitsComparator
Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator
com.liferay.bookmarks.api
Fact
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay.portal.rules.engine.api
FeedContentFieldException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
FeedIdException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
FeedNameException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
FeedTargetLayoutFriendlyUrlException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
FeedTargetPortletIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
FieldConstants
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay.dynamic.data.mapping.api
FieldRenderer
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay.dynamic.data.mapping.api
FieldRendererFactory
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay.dynamic.data.mapping.api
Fields
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay.dynamic.data.mapping.api
FlagsEntryService
Old: com.liferay.portlet.flags.service New: com.liferay.flags.service
com.liferay.flags.api
FlagsEntryServiceUtil
Old: com.liferay.portlet.flags.service New: com.liferay.flags.service
com.liferay.flags.api
FlagsEntryServiceWrapper
Old: com.liferay.portlet.flags.service New: com.liferay.flags.service
com.liferay.flags.api
FlagsRequest
Old: com.liferay.portlet.flags.messaging New: com.liferay.flags.messaging
com.liferay.flags.service
GroupConverterKeys
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap
com.liferay.portal.security.ldap
ImportFilesException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
ItemLargeImageNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ItemLargeImageSizeException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ItemMediumImageNameException

Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ItemMediumImageSizeException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ItemMinQuantityComparator
Old: com.liferay.portlet.shopping.util.comparator New: com.liferay.shopping.util.comparator
com.liferay.shopping.api
ItemNameComparator
Old: com.liferay.portlet.shopping.util.comparator New: com.liferay.shopping.util.comparator
com.liferay.shopping.api
ItemNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ItemPriceComparator
Old: com.liferay.portlet.shopping.util.comparator New: com.liferay.shopping.util.comparator
com.liferay.shopping.api
ItemSKUComparator
Old: com.liferay.portlet.shopping.util.comparator New: com.liferay.shopping.util.comparator
com.liferay.shopping.api
ItemSKUException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ItemSmallImageNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ItemSmallImageSizeException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
JobStateSerializeUtil
Old: com.liferay.portal.kernel.scheduler New: com.liferay.portal.scheduler
com.liferay.portal.scheduler
JournalArticle
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleConstants
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleDisplay
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleFinder
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalArticleImage
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleImageLocalService

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleImageLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleImageLocalServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleImageModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleImagePersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalArticleImageSoap
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleImageUtil
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalArticleImageWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleLocalServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticlePersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalArticleResource
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleResourceLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleResourceLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleResourceLocalServiceWrapper

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleResourceModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleResourcePersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalArticleResourceSoap
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleResourceUtil
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalArticleResourceWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalArticleSoap
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalArticleUtil
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalArticleWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalContent
Old: com.liferay.portlet.journalcontent.util New: com.liferay.journal.util
com.liferay.journal.api
JournalContentSearch
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalContentSearchLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalContentSearchLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalContentSearchLocalServiceWrapper

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalContentSearchModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalContentSearchPersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalContentSearchSoap
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalContentSearchUtil
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalContentSearchWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalConverter
Old: com.liferay.portlet.journal.util New: com.liferay.journal.util
com.liferay.journal.api
JournalFeed
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalFeedConstants
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalFeedFinder
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalFeedLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFeedLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFeedLocalServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFeedModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalFeedPersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalFeedService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFeedServiceUtil

Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFeedServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFeedSoap
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalFeedUtil
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalFeedWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalFolder
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalFolderFinder
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalFolderLocalService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFolderLocalServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFolderLocalServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFolderModel
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalFolderPersistence
Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalFolderService
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFolderServiceUtil
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFolderServiceWrapper
Old: com.liferay.portlet.journal.service New: com.liferay.journal.service
com.liferay.journal.api
JournalFolderSoap
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalFolderUtil

Old: com.liferay.portlet.journal.service.persistence New: com.liferay.journal.service.persistence
com.liferay.journal.api
JournalFolderWrapper
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalSearchConstants
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
JournalStructureConstants
Old: com.liferay.portlet.journal.model New: com.liferay.journal.model
com.liferay.journal.api
LDAPFilterException
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.validator
com.liferay.portal.security.ldap
LDAPGroup
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay.portal.security.ldap
LDAPServerNameException
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap
com.liferay.portal.security.ldap
LDAPToPortalConverter
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay.portal.security.ldap
LDAPUser
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay.portal.security.ldap
LDAPUtil
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.util
com.liferay.portal.security.ldap
LockLocalService
Old: com.liferay.portal.service New: com.liferay.portal.lock.service
com.liferay.portal.lock.api
LockLocalServiceUtil
Old: com.liferay.portal.service New: com.liferay.portal.lock.service
com.liferay.portal.lock.api
LockLocalServiceWrapper
Old: com.liferay.portal.service New: com.liferay.portal.lock.service
com.liferay.portal.lock.api
LockModel
Old: com.liferay.portal.model New: com.liferay.portal.lock.model
com.liferay.portal.lock.api
LockPersistence
Old: com.liferay.portal.service.persistence New: com.liferay.portal.lock.service.persistence
com.liferay.portal.lock.api
LockSoap
Old: com.liferay.portal.model New: com.liferay.portal.lock.model
com.liferay.portal.lock.api
LockUtil

Old: com.liferay.portal.service.persistence New: com.liferay.portal.lock.service.persistence
com.liferay.portal.lock.api
LockWrapper
Old: com.liferay.portal.model New: com.liferay.portal.lock.model
com.liferay.portal.lock.api
MBeanRegistry
Old: com.liferay.portal.kernel.jmx New: com.liferay.portal.jmx
com.liferay.portal.jmx
MDRAction
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay.mobile.device.rules.api
MDRActionLocalService
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRActionLocalServiceUtil
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRActionLocalServiceWrapper
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRActionModel
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay.mobile.device.rules.api
MDRActionPersistence
Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence
com.liferay.mobile.device.rules.api
MDRActionService
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRActionServiceUtil
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRActionServiceWrapper
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRActionSoap
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay.mobile.device.rules.api
MDRActionUtil
Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence
com.liferay.mobile.device.rules.api
MDRActionWrapper
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay.mobile.device.rules.api
MDRPermission

Old: com.liferay.portlet.mobiledevicerules.service.permission New: com.liferay.mobile.device.rules.service.permission

com.liferay.mobile.device.rules.service

MDRRule

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay.mobile.device.rules.api

MDRRuleGroup

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay.mobile.device.rules.api

MDRRuleGroupFinder

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceLocalService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceLocalServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceLocalServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceModel

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay.mobile.device.rules.api

MDRRuleGroupInstancePermission

Old: com.liferay.portlet.mobiledevicerules.service.permission New: com.liferay.mobile.device.rules.service.permission

com.liferay.mobile.device.rules.service

MDRRuleGroupInstancePersistence

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceSoap

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceUtil

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence com.liferay.mobile.device.rules.api

MDRRuleGroupInstanceWrapper

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay.mobile.device.rules.api

MDRRuleGroupLocalService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupLocalServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupLocalServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupModel

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay.mobile.device.rules.api

MDRRuleGroupPermission

Old: com.liferay.portlet.mobiledevicerules.service.permission New: com.liferay.mobile.device.rules.service.permission

com.liferay.mobile.device.rules.service

MDRRuleGroupPersistence

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay.mobile.device.rules.api

MDRRuleGroupService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupServiceWrapper

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleGroupSoap

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay.mobile.device.rules.api

MDRRuleGroupUtil

Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence

com.liferay.mobile.device.rules.api

MDRRuleGroupWrapper

Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model

com.liferay.mobile.device.rules.api

MDRRuleLocalService

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api

MDRRuleLocalServiceUtil

Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service

com.liferay.mobile.device.rules.api
MDRRuleLocalServiceWrapper
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRRuleModel
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay.mobile.device.rules.api
MDRRulePersistence
Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence
com.liferay.mobile.device.rules.api
MDRRuleService
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRRuleServiceUtil
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRRuleServiceWrapper
Old: com.liferay.portlet.mobiledevicerules.service New: com.liferay.mobile.device.rules.service
com.liferay.mobile.device.rules.api
MDRRuleSoap
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay.mobile.device.rules.api
MDRRuleUtil
Old: com.liferay.portlet.mobiledevicerules.service.persistence New: com.liferay.mobile.device.rules.service.persistence
com.liferay.mobile.device.rules.api
MDRRuleWrapper
Old: com.liferay.portlet.mobiledevicerules.model New: com.liferay.mobile.device.rules.model
com.liferay.mobile.device.rules.api
MemoryReportDesignRetriever
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
MessageBusManager
Old: com.liferay.portal.kernel.messaging.jmx New: com.liferay.portal.messaging.internal.jmx
com.liferay.portal.messaging
MessageBusManagerMBean
Old: com.liferay.portal.kernel.messaging.jmx New: com.liferay.portal.messaging.internal.jmx
com.liferay.portal.messaging
Modifications
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay.portal.security.ldap
NoSuchArticleException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
NoSuchArticleImageException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api

NoSuchArticleResourceException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
NoSuchCartException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
NoSuchChoiceException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api
NoSuchContentSearchException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
NoSuchCouponException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
NoSuchFeedException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
NoSuchItemException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
NoSuchItemFieldException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
NoSuchItemPriceException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
NoSuchNodeException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
NoSuchOrderException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
NoSuchOrderItemException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
NoSuchPageException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
NoSuchPageResourceException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
NoSuchQuestionException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api
NoSuchRecordException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay.dynamic.data.lists.api

NoSuchRecordSetException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay.dynamic.data.lists.api
NoSuchRecordVersionException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay.dynamic.data.lists.api
NoSuchRuleException
Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception
com.liferay.mobile.device.rules.api
NoSuchRuleGroupException
Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception
com.liferay.mobile.device.rules.api
NoSuchRuleGroupInstanceException
Old: com.liferay.portlet.mobiledevicerules New: com.liferay.mobile.device.rules.exception
com.liferay.mobile.device.rules.api
NoSuchStorageLinkException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
NoSuchStructureLinkException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
NoSuchTemplateException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
NoSuchTemplateException
Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
NoSuchVoteException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api
NodeNameException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
OrderDateComparator
Old: com.liferay.portlet.shopping.util.comparator New: com.liferay.shopping.util.comparator
com.liferay.shopping.api
PageContentException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
PageCreateDateComparator
Old: com.liferay.portlet.wiki.util.comparator New: com.liferay.wiki.util.comparator
com.liferay.wiki.service
PageTitleComparator
Old: com.liferay.portlet.wiki.util.comparator New: com.liferay.wiki.util.comparator
com.liferay.wiki.service
PageTitleException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api

PageVersionComparator
Old: com.liferay.portlet.wiki.util.comparator New: com.liferay.wiki.util.comparator
com.liferay.wiki.service
PageVersionException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
PollsChoice
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsChoiceLocalService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsChoiceLocalServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsChoiceLocalServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsChoiceModel
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsChoicePersistence
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay.polls.api
PollsChoiceService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsChoiceServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsChoiceServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsChoiceSoap
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsChoiceUtil
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay.polls.api
PollsChoiceWrapper
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsQuestion
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsQuestionLocalService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api

PollsQuestionLocalServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsQuestionLocalServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsQuestionModel
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsQuestionPersistence
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay.polls.api
PollsQuestionService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsQuestionServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsQuestionServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsQuestionSoap
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsQuestionUtil
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay.polls.api
PollsQuestionWrapper
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsVote
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsVoteLocalService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsVoteLocalServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsVoteLocalServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsVoteModel
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsVotePersistence
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay.polls.api

PollsVoteService
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsVoteServiceUtil
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsVoteServiceWrapper
Old: com.liferay.portlet.polls.service New: com.liferay.polls.service
com.liferay.polls.api
PollsVoteSoap
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PollsVoteUtil
Old: com.liferay.portlet.polls.service.persistence New: com.liferay.polls.service.persistence
com.liferay.polls.api
PollsVoteWrapper
Old: com.liferay.portlet.polls.model New: com.liferay.polls.model
com.liferay.polls.api
PortalExecutorFactory
Old: com.liferay.portal.kernel.executor New: com.liferay.portal.executor.internal
com.liferay.portal.executor
PortalToLDAPConverter
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay.portal.security.ldap
PortletDisplayTemplate
Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.portlet.display.template
com.liferay.portlet.display.template
PortletDisplayTemplateConstants
Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.portlet.display.template
com.liferay.portlet.display.template
PortletDisplayTemplateUtil
Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.portlet.display.template
com.liferay.portlet.display.template
QueryIndexingHitsProcessor
Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.hits
com.liferay.portal.search
QuerySuggestionHitsProcessor
Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal.hits
com.liferay.portal.search
QueryType
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay.portal.rules.engine.api
QuestionChoiceException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api
QuestionDescriptionException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api

QuestionExpirationDateException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api
QuestionExpiredException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api
QuestionTitleException
Old: com.liferay.portlet.polls New: com.liferay.polls.exception
com.liferay.polls.api
RecordSetDDMStructureIdException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay.dynamic.data.lists.api
RecordSetDuplicateRecordSetKeyException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay.dynamic.data.lists.api
RecordSetNameException
Old: com.liferay.portlet.dynamicdatalists New: com.liferay.dynamic.data.lists.exception
com.liferay.dynamic.data.lists.api
RegistryAwareMBeanServer
Old: com.liferay.portal.kernel.jmx New: com.liferay.portal.jmx.internal
com.liferay.portal.jmx
ReportCompilerRequestMessageListener
Old: com.liferay.portal.kernel.bi.reporting.messaging New: com.liferay.portal.reports.engine.messag-
ing
com.liferay.portal.reports.engine.api
ReportDataSourceType
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportDesignRetriever
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportEngine
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportExportException
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportFormat
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportFormatExporter
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportFormatExporterRegistry
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportGenerationException
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine

com.liferay.portal.reports.engine.api
ReportRequest
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportRequestContext
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
ReportRequestMessageListener
Old: com.liferay.portal.kernel.bi.reporting.messaging New: com.liferay.portal.reports.engine.messaging
com.liferay.portal.reports.engine.api
ReportResultContainer
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
RequestStatistics
Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics
com.liferay.portal.monitoring
RequiredCouponException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
RequiredNodeException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
RequiredTemplateException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
RequiredTemplateException
Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
RuleGroupInstancePriorityComparator
Old: com.liferay.portlet.mobiledevicerules.util New: com.liferay.mobile.device.rules.util.comparator
com.liferay.mobile.device.rules.api
RuleGroupProcessor
Old: com.liferay.portal.kernel.mobile.device.rulegroup New: com.liferay.mobile.device.rules.rule
com.liferay.mobile.device.rules.api
RuleGroupProcessorUtil
Old: com.liferay.portal.kernel.mobile.device.rulegroup New: com.liferay.mobile.device.rules.rule
com.liferay.mobile.device.rules.api
RuleHandler
Old: com.liferay.portal.kernel.mobile.device.rulegroup.rule New: com.liferay.mobile.device.rules.rule
com.liferay.mobile.device.rules.api
RulesEngine
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay.portal.rules.engine.api
RulesEngineException
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay.portal.rules.engine.api

RulesEngineUtil
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay.portal.rules.engine.api
RulesLanguage
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay.portal.rules.engine.api
RulesResourceRetriever
Old: com.liferay.portal.kernel.bi.rules New: com.liferay.portal.rules.engine
com.liferay.portal.rules.engine.api
ServletContextReportDesignRetriever
Old: com.liferay.portal.kernel.bi.reporting.servlet New: com.liferay.portal.reports.engine.servlet
com.liferay.portal.reports.engine.api
ShippingCityException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShippingCountryException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShippingEmailAddressException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShippingFirstNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShippingLastNameException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShippingPhoneException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShippingStateException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShippingStreetException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShippingZipException
Old: com.liferay.portlet.shopping New: com.liferay.shopping.exception
com.liferay.shopping.api
ShoppingCart
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCartItem
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCartLocalService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api

ShoppingCartLocalServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCartLocalServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCartModel
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCartPersistence
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingCartSoap
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCartUtil
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingCartWrapper
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCategory
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCategoryConstants
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCategoryLocalService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCategoryLocalServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCategoryLocalServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCategoryModel
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCategoryPersistence
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingCategoryService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCategoryServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api

ShoppingCategoryServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCategorySoap
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCategoryUtil
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingCategoryWrapper
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCoupon
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCouponConstants
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCouponFinder
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingCouponLocalService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCouponLocalServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCouponLocalServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCouponModel
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingCouponPersistence
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingCouponService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCouponServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCouponServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingCouponSoap
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api

ShoppingCouponUtil
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingCouponWrapper
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItem
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemField
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemFieldLocalService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemFieldLocalServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemFieldLocalServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemFieldModel
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemFieldPersistence
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingItemFieldSoap
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemFieldUtil
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingItemFieldWrapper
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemFinder
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingItemLocalService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemLocalServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemLocalServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api

ShoppingItemModel
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemPersistence
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingItemPrice
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemPriceConstants
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemPriceLocalService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemPriceLocalServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemPriceLocalServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemPriceModel
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemPricePersistence
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingItemPriceSoap
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemPriceUtil
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingItemPriceWrapper
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingItemService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingItemSoap
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api

ShoppingItemUtil
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingItemWrapper
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrder
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrderConstants
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrderFinder
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingOrderItem
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrderItemLocalService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingOrderItemLocalServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingOrderItemLocalServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingOrderItemModel
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrderItemPersistence
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingOrderItemSoap
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrderItemUtil
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingOrderItemWrapper
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrderLocalService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingOrderLocalServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api

ShoppingOrderLocalServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingOrderModel
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrderPersistence
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingOrderService
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingOrderServiceUtil
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingOrderServiceWrapper
Old: com.liferay.portlet.shopping.service New: com.liferay.shopping.service
com.liferay.shopping.api
ShoppingOrderSoap
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
ShoppingOrderUtil
Old: com.liferay.portlet.shopping.service.persistence New: com.liferay.shopping.service.persistence
com.liferay.shopping.api
ShoppingOrderWrapper
Old: com.liferay.portlet.shopping.model New: com.liferay.shopping.model
com.liferay.shopping.api
SortFactoryImpl
Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal
com.liferay.portal.search
Statistics
Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.statistics
com.liferay.portal.monitoring
StorageAdapter
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay.dynamic.data.mapping.api
StorageEngine
Old: com.liferay.portlet.dynamicdatamapping.storage New: com.liferay.dynamic.data.mapping.storage
com.liferay.dynamic.data.mapping.api
StorageException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
StorageFieldNameException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
StructureDuplicateStructureKeyException
Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api

StructureFieldException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

StructureIdComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay.dynamic.data.mapping.api

StructureModifiedDateComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay.dynamic.data.mapping.api

StructureStructureKeyComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay.dynamic.data.mapping.api

SummaryStatistics

Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.statistics

com.liferay.portal.monitoring

SynchronousMessageListener

Old: com.liferay.portal.kernel.messaging.sender New: com.liferay.portal.messaging.internal.sender

com.liferay.portal.messaging

TemplateDuplicateTemplateKeyException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

TemplateIdComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay.dynamic.data.mapping.api

TemplateModifiedDateComparator

Old: com.liferay.portlet.dynamicdatamapping.util.comparator New: com.liferay.dynamic.data.mapping.util.comparator

com.liferay.dynamic.data.mapping.api

TemplateNameException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

TemplateNameException

Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

TemplateScriptException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

TemplateSmallImageNameException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

TemplateSmallImageNameException

Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

TemplateSmallImageSizeException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
TemplateSmallImageSizeException
Old: com.liferay.portlet.journal New: com.liferay.dynamic.data.mapping.exception
com.liferay.dynamic.data.mapping.api
UnknownRuleHandlerException
Old: com.liferay.portal.kernel.mobile.device.rulegroup.rule New: com.liferay.mobile.device.rules.rule
com.liferay.mobile.device.rules.api
UserConverterKeys
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap
com.liferay.portal.security.ldap
WikiFormatException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
WikiNode
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiNodeLocalService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiNodeLocalServiceUtil
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiNodeLocalServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiNodeModel
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiNodePersistence
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay.wiki.api
WikiNodeService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiNodeServiceUtil
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiNodeServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiNodeSoap
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiNodeUtil
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay.wiki.api
WikiNodeWrapper

Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPage
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPageConstants
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPageDisplay
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPageFinder
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay.wiki.api
WikiPageLocalService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageLocalServiceUtil
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageLocalServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageModel
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPagePersistence
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay.wiki.api
WikiPageResource
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPageResourceLocalService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageResourceLocalServiceUtil
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageResourceLocalServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageResourceModel
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPageResourcePersistence
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay.wiki.api
WikiPageResourceSoap

Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPageResourceUtil
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay.wiki.api
WikiPageResourceWrapper
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPageService
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageServiceUtil
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageServiceWrapper
Old: com.liferay.portlet.wiki.service New: com.liferay.wiki.service
com.liferay.wiki.api
WikiPageSoap
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api
WikiPageUtil
Old: com.liferay.portlet.wiki.service.persistence New: com.liferay.wiki.service.persistence
com.liferay.wiki.api
WikiPageWrapper
Old: com.liferay.portlet.wiki.model New: com.liferay.wiki.model
com.liferay.wiki.api

Related Topics

Liferay Upgrade Planner
Development Reference

Chapter 126

Theme Gulp Tasks

Theme projects that use the liferay JS Theme Toolkit, such as those created with the Liferay Theme Generator have access to several gulp tasks that you can execute to manage and deploy your theme.

Note: Gulp is included as a local dependency in generated themes, so you are not required to install it. It can be accessed by running `node_modules\.bin\gulp` followed by the Gulp task from a generated theme's root folder.

Here are the gulp tasks you can execute:

- `build`: generates the base theme files, compiles Sass into CSS, and zips all theme files into a WAR file that you can deploy to a Liferay server.
- `deploy`: runs the `build` task and deploys the WAR file to the configured local app server.

```
**Note:** If you're running the [Felix Gogo shell](/docs/7-0/reference/-/knowledge_base/r/using-the-felix-gogo-shell),
you can also deploy your theme using the `gulp deploy:gogo` command. **This
task will NOT work for 6.2 themes.**
```

- `extend`: allows you to specify a base theme or themelet to extend. By default, themes created with the Liferay Theme Generator are based off of the styled theme.

  You first are prompted if you want to extend a Base theme or Themelet, then you're prompted for where you would like to search for modules. Selecting *Globally installed npm modules* searches globally accessible npm modules on your computer. Selecting *npm registry* searches for published modules on npm. If you have v8.x.x of the Liferay JS Theme Toolkit installed, you can also *Specify a package URL* to locate a themelet.

```
**Note:** You can retrieve the URL for a package by running
`npm show package-name dist.tarball`.
```

```
After you've selected modules from the options it provides, the modules are
added to your `package.json` file as dependencies. Run `npm install` to
install them.
```

```
 **Note:** The Classic theme is an implementation of an existing base theme
 and is therefore not meant to be extended. Extending Liferay's Classic
 theme is strongly discouraged.
```

- `kickstart`: allows you to copy the CSS, images, JS, and templates from another theme into the `src` directory of your own. While this is similar to the `extend` task, kickstarting from another theme is a one time inheritance, whereas extending from another theme is a dynamic inheritance that applies your src files on top of the base theme on every build.
- `init`: prompts you for local and remote app server information to use in theme deployment.
- `status`: displays the name of the base theme/themelets your theme extends.
- `watch`: allows you to preview the changes you make to your theme without requiring a full redeploy. After invoking the `watch` task, every time you save any changes to a file in your theme, applicable changes are compiled and they're copied directly to a proxy port (e.g. `9080`) for you to preview live. **Note:** you must have Developer Mode enabled to use the `watch` task. Once you're happy with the live preview, deploy your theme to apply the changes.

Related Topics
Liferay Theme Generator

CHAPTER 127

THEME REFERENCE GUIDE

A theme is made up of several files. Although most of the files are named after their matching components, their function may be unclear.

This document explains each file's usage to make clear which files to modify and which files to leave untouched.

Theme Anatomy

There are two main approaches to theme development for 7.0: themes built using the Node.js build tools with the theme generator and themes built using @ide@.

Themes developed with the theme generator have the anatomy shown below. Although themes developed with @ide@ have a slightly different anatomy built with the theme project template, the core theme files are the same. Note that the `build` folder is shown for reference, and is generated when the theme is compiled.

- `theme-name/`
  - `build/`(generated when the theme is compiled)
    - `css/`
      - `_application.scss`
      - `_aui_custom.scss`
      - `_aui_variables.scss`
      - `_base.scss`
      - `_custom.scss`
      - `_extras.scss`
      - `_imports.scss`
      - `_layout.scss`
      - `_liferay_custom.scss`
      - `_liferay_variables_custom.scss`
      - `_liferay_variables.scss`
      - `_navigation.scss`
      - `_portal.scss`
      - `_portlet.scss`

        - _taglib.scss
        - application/
        - aui/
        - aui.scss
        - base/
        - font-awesome.scss
        - layout/
        - main.scss
        - navigation/
        - portal/
        - portlet/
        - taglib/

    * templates/

        - init_custom.ftl
        - init.ftl
        - navigation.ftl
        - portal_normal.ftl
        - portal_pop_up.ftl
        - portlet.ftl
        - (other directories that have been copied from src)

- `dist/` (generated when the theme is compiled. This is where the theme's war file is placed after a build/deploy.)
- `gulpfile.js`
- `liferay-theme.json`
- `node_modules/` (generated when an `npm install` command is run from the root of the theme, and can be deleted at anytime and re-generated by running `npm install`.)

    * (many directories)

- `package.json`
- `src/`

    * `css/`

        - (modified CSS files)

    * `images/`

        - (many directories)

    * `js/`

        - `main.js`

    * `templates/`

        - (Modified theme templates)

* WEB-INF/ - lib/

  - liferay-look-and-feel.xml
  - liferay-plugin-package.properties
  - src/
  - resources-importer/
  - (Many directories)

Regarding CSS files, it is recommended that you only modify `_custom.scss`, `_aui_custom.scss`, `_aui_variables.scss`, and `_liferay_variables_custom.scss`.

You can of course overwrite any CSS file that you wish, but if you modify any other files, you will most likely be removing styling that 7.0 needs to work properly.

127.1 Theme Files

_application.scss

Contains imports for application styles. Generally these files style components that aren't Liferay specific, i.e. Alloy or Bootstrap components.

_aui_custom.scss

Used for AUI custom styles, i.e. styles for a third party Bootstrap theme. Anything written in this file is compiled in the same scope as Bootstrap/Lexicon, so you can use their variables, mixins, etc. You can also implement any of the variables you define in `_aui_variables.scss`.

_aui_variables.scss

Used to store custom Sass variables. This file get's injected into the Bootstrap/Lexicon build, so you can overwrite variables and change how those libraries are compiled.

_base.scss

Contains imports for the base styles for Liferay.

_custom.scss

Used for custom CSS styles. It is recommended that you place all of your custom CSS modfications in this file.

_extras.scss

Contains styling that is considered non-essential and potentially dated in the near future i.e. box-shadows, rounded corners, etc. This allows for easy maintenance.

_imports.scss

Contains imports for third-party libraries required for the theme e.g. Bourbon, Liferay Mixins, Lexicon Base Variables, and Bootstrap Mixins.

_layout.scss

Contains imports for layout styles and variables.

_liferay_custom.scss

Contains Liferay DXP styles that are compiled in the same scope as Bootstrap/Lexicon.

It's recommended that you NOT overwrite this file.

_liferay_variables_custom.scss

Used for overwriting variables defined in `_liferay_variables.scss` without wiping out the whole file.

_liferay_variables.scss

Contains variables that are used in `_liferay_custom.scss`.

It's recommended that you NOT overwrite this file.

_navigation.scss

Contains imports for navigation styles.

_portal.scss

Contains imports for Portal components.

_portlet.scss

Contains imports for portlet components.

_taglib.scss

Contains imports for taglib styles.

aui.scss

Contains the Lexicon base CSS import. If you want to just use Bootstrap, or use Atlas, you can do so by adding one of the following imports:

```
@import "aui/lexicon/bootstrap";
```

or

```
@import "aui/lexicon/atlas";
```

font-awesome.scss

Contains the Font Awesome icon imports for Liferay.

main.scss

Contains imports for the core CSS files.

init_custom.ftl

Used for custom FreeMarker variables i.e. theme setting variables.

init.ftl

Contains common FreeMarker variables that are available to use in your theme templates. Useful for reference if you need access to theme objects.

It's recommended that you NOT overwrite this file.

navigation.ftl

The theme template for the theme's navigation.

portal_normal.ftl

Similar to the `index.html` of a website, this file acts as a hub for all of the theme templates.

portal_pop_up.ftl

The theme template for pop up dialogs for the theme's portlets.

portlet.ftl

The theme template for the theme's portlets. If your theme uses Application Decorators, you can modify this file to create application decorator specific theme settings. See the Portlet Decorators tutorial for more info.

gulpfile.js

Defines the required gulp tasks for Node.js tool developed themes.

It's recommended that you NOT overwrite this file.

liferay-theme.json

Contains the configuration settings for your app server, in Node.js tool based themes. You can change this file manually at any time to update your server settings. The file can also be updated via the `gulp init` task.

package.json

contains theme setting information such as the theme template langauge, version, and base theme, for Node.js tool developed themes. This file can be updated manually. The `gulp extend` task can also be used to change the base theme.

main.js

Used for custom JavaScript.

liferay-look-and-feel.xml

Contains basic information for the theme. If your theme has theme settings , they are defined in this file. For a full explanation of this file please see the Definitions docs.

liferay-plugin-package.properties

Contains general properties for the theme. [Resources Importer]{/docs/7-0/tutorials/-/knowledge_base/t/importing-resources-with-a-theme} configuration settings are also placed in this file. For a full explanation of the properties available for this file please see the 7.0 Propertiesdoc.

Chapter 128

Screenlets in Liferay Screens for Android

Liferay Screens for Android contains several Screenlets that you can use in your Android apps. This section contains the reference documentation for each. If you're looking for instructions on using Screens, see the Screens tutorials. The Screens tutorials contain instructions on using Screenlets and using views in Screenlets. Each Screenlet reference document here lists the Screenlet's features, compatibility, its module (if any), available Views, attributes, listener methods, and more. The available Screenlets are listed here with links to their reference documents:

- **Login Screenlet:** Signs users in to a Liferay DXP instance.
- **Sign Up Screenlet:** Registers new users in a Liferay DXP instance.
- **Forgot Password Screenlet:** Sends emails containing a new password or password reset link to users.
- **User Portrait Screenlet:** Show the user's portrait picture.
- **DDL Form Screenlet:** Presents dynamic forms to be filled out by users and submitted back to the server.
- **DDL List Screenlet:** Shows a list of records based on a pre-existing DDL in a Liferay instance.
- **Asset List Screenlet:** Shows a list of assets managed by Liferay's Asset Framework. This includes web content, blog entries, documents, users, and more.
- **Web Content Display Screenlet:** Shows the web content's HTML or structured content. This Screenlet uses the features available in Web Content Management.
- **Web Content List Screenlet:** Shows a list of web contents from a folder, usually based on a pre-existing `DDMStructure`.
- **Image Gallery Screenlet:** Shows a list of images from a folder. This Screenlet also lets users upload and delete images.
- **Rating Screenlet:** Shows the rating for an asset. This Screenlet also lets the user update or delete the rating.
- **Comment List Screenlet:** Shows a list of comments for an asset.

- **Comment Display Screenlet:** Shows a single comment for an asset.
- **Comment Add Screenlet:** Lets the user comment on an asset.
- **Asset Display Screenlet:** Displays an asset. Currently, this Screenlet can display Documents and Media Library files (`DLFileEntry` entities), blog articles (`BlogsEntry` entities), and web content articles (`WebContent` entities). You can also use it to display custom assets.
- **Blogs Entry Display Screenlet:** Shows a single blog entry.
- **Image Display Screenlet:** Shows a single image file from a Liferay DXP instance's Documents and Media Library.
- **Video Display Screenlet:** Shows a single video file from a Liferay DXP instance's Documents and Media Library.
- **Audio Display Screenlet:** Shows a single audio file from a Liferay DXP instance's Documents and Media Library.
- **PDF Display Screenlet:** Shows a single PDF file from a Liferay DXP instance's Documents and Media Library.
- **Web Screenlet:** Displays any web page. You can also customize the web page through injection of local and remote JavaScript and CSS files.

128.1 Login Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Login Screenlet lets you authenticate portal users in your Android app. The following types of authentication are supported:

- **Basic:** uses user login and password according to HTTP Basic Access Authentication specification. Depending on the authentication method used by your Liferay instance, you need to provide the user's email address, screen name, or user ID. You also need to provide the user's password.
- **OAuth:** implements the OAuth 1.0a specification.

- **Cookie:** uses a cookie to log in. This lets you access documents and images in the portal's document library without the guest view permission in the portal. The other authentication types require this permission to access such files.

Note: Cookie authentication with Login Screenlet is broken in fix packs 14 through 18 of Liferay Digital Enterprise 7.0. This issue is fixed in newer fix packs for Liferay Digital Enterprise 7.0.

For instructions on configuring the Screenlet to use these authentication types, see the below Portal Configuration and Screenlet Attributes sections.

When a user successfully authenticates, their user attributes are retrieved for use in the app. You can use the `SessionContext` class to get the current user's attributes.

Note that user credentials and attributes can be stored in an app's data store (see the `saveCredentials` attribute). Android's `SharedPreferences` is currently the only data store implemented. However, new and more secure data stores will be added in the future. Stored user credentials can be used to automatically log the user in to subsequent sessions. To do this, you can use the method `SessionContext.loadStoredCredentials()`.

JSON Services Used

Screenlets in Liferay Screens call the portal's JSON web services. This Screenlet calls the following services and methods.

Service	Method	Notes
`UserService`	`getUserByEmailAddress`	Basic login
`UserService`	`getUserByScreenName`	Basic login
`UserService`	`getUserById`	Basic login
`UserService`	`getCurrentUser`	Cookie and OAuth login

Module

- Auth

Views

- Default
- Material

For instructions on using these Views, see the `layoutId` attribute in the Attributes section below.

Portal Configuration

Basic Authentication

Before using Login Screenlet, you should make sure your portal is configured with the authentication option you want to use. You can choose email address, screen name, or user ID. You can set this in the Control Panel by selecting *Configuration → Instance Settings*, and then selecting the *Authentication* section. The authentication options are in the *How do users authenticate?* selector menu.

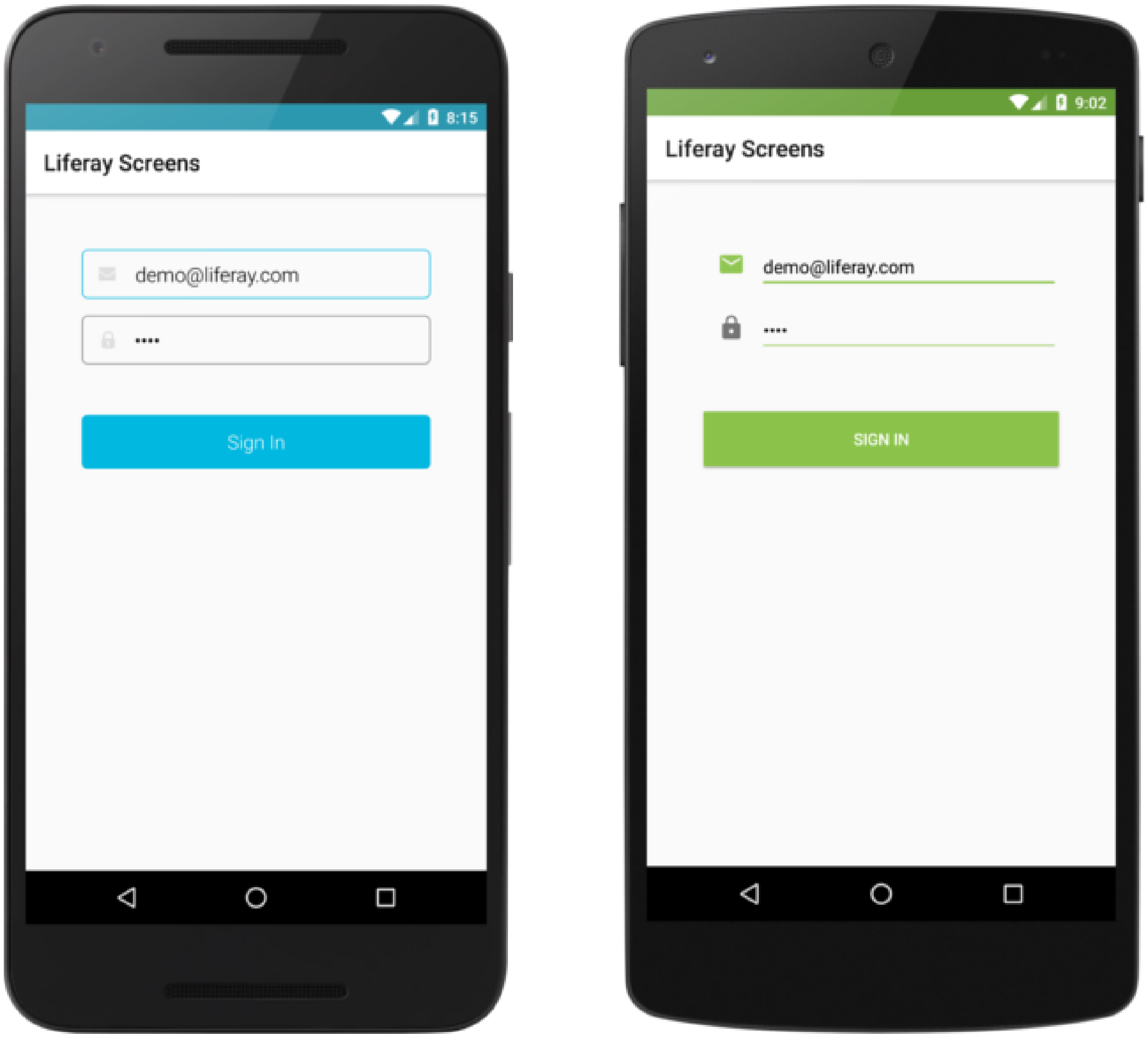
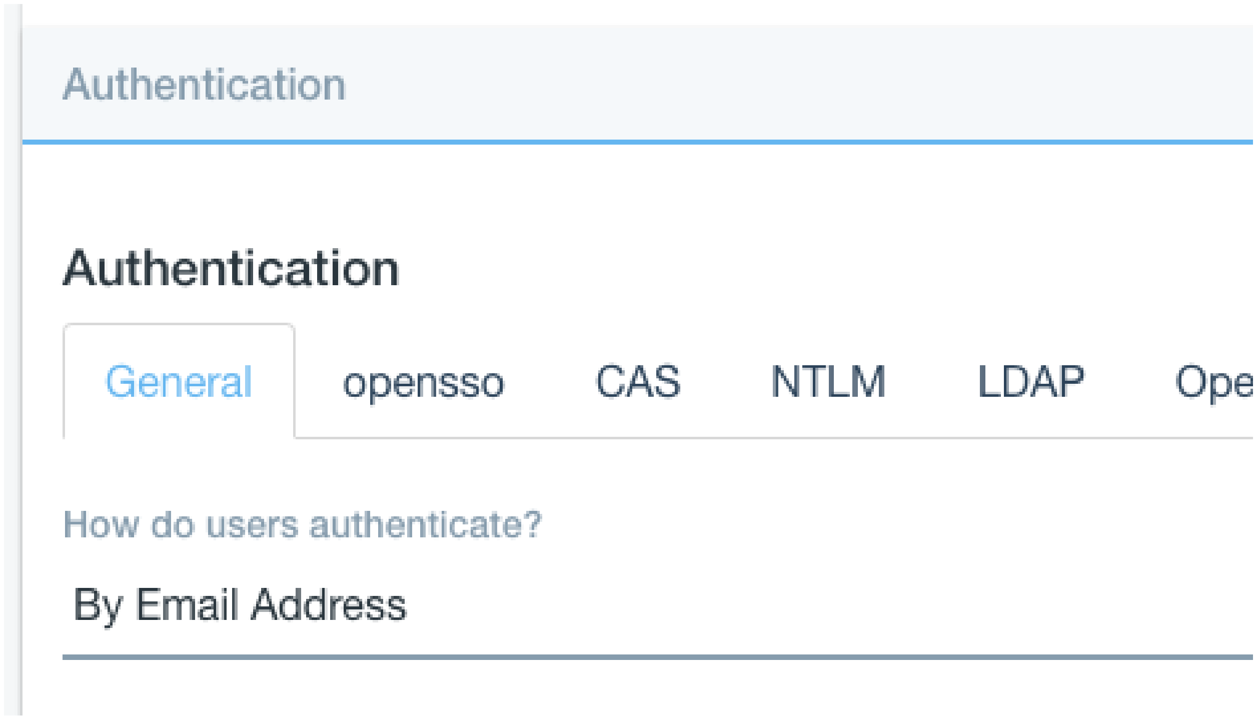


Figure 128.1: The Login Screenlet using the Default (left) and Material (right) Viewsets.


Authentication

Authentication

How do users authenticate?

By Email Address

Figure 128.2: Setting the authentication method in your Liferay instance.

For more details, see the Setting up a Liferay Instance section of the User Guide.

OAuth Authentication

Note: OAuth authentication is only available in Liferay DXP instances.

To use OAuth authentication, first install the OAuth provider app from the Liferay Marketplace. Click here to get this app. Once it's installed, go to *Control Panel* → *Users* → *OAuth Admin*, and add a new application to be used from Liferay Screens. Once the application exists, copy the *Consumer Key* and *Consumer Secret* values for later use in Login Screenlet.

Offline

This Screenlet doesn't support offline mode. It requires network connectivity. If you need to log in users automatically, even when there's no network connection, you can use the `credentialsStorage` attribute together with the `SessionContext.loadStoredCredentials` method.

Required Attributes

- None

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The ID of the View's layout. You can set this attribute to `@layout/login_default` (Default View) or `@layout/login_material` (Material View). To use the Material View, you must first install the Material View Set. Click here for instructions on installing and using Views and View Sets, including the Material View Set. | `companyId` | `number` | The ID of the portal instance to authenticate to. If you don't set this attribute or set it to `0`, the Screenlet uses the `companyId` setting in `LiferayServerContext`. | `loginMode` | `enum` | The Screenlet's authentication type. You can set this attribute to `basic`, `oauth`, or `cookie`. If you don't set this attribute, the Screenlet defaults to basic authentication. | `basicAuthMethod` | `string` | Specifies the authentication option to use with basic or cookie authentication. You can set this attribute to `email`, `screenName` or `userId`. This must match the server's authentication option. If you don't set this attribute, and don't set the `loginMode` attribute to `oauth`, the Screenlet defaults to basic authentication with the `email` option. | `OAuthConsumerKey` | `string` | Specifies the *Consumer Key* to use in OAuth authentication. | `OAuthConsumerSecret` | `string` | Specifies the *Consumer Secret* to use in OAuth authentication. | `credentialsStorage` | `enum` | Sets the mode for storing user credentials. The possible values are `none`, `auto`, and `shared_preferences`. If set to `shared_preferences`, the user credentials and attributes are stored using Android's `SharedPreferences` class. If set to `none`, user credentials and attributes aren't saved at all. If set to `auto`, the best of the available storage modes is used. Currently, this is equivalent to `shared_preferences`. The default value is `none`. | `shouldHandleCookieExpiration` | `bool` | Whether to refresh the cookie automatically when using cookie login. When set to `true` (the default value), the cookie refreshes as it's about to expire. | `cookieExpirationTime` | `int` | How long the cookie lasts, in seconds. This value depends on your portal instance's configuration. The default value is `900`. | `authenticator` | `Authenticator` | An instance of a class that implements the `Authenticator` interface. The *Challenge-Response Authentication* section below discusses this further. |

Listener

The Login Screenlet delegates some events to an object that implements the `LoginListener` interface. This interface let you implement the following methods:

- `onLoginSuccess(User user)`: Called when login successfully completes. The `user` parameter contains a set of the logged in user's attributes. The supported keys are the same as those in the portal's User entity.

- `onLoginFailure(Exception e)`: Called when an error occurs in the process.

Challenge-Response Authentication

To support challenge-response authentication when using a cookie to log in to the portal, Login Screenlet has an `authenticator` attribute. As mentioned in the above *Attributes* table, this attribute's value is a class that implements the `Authenticator` interface.

Here's an example of such a class. It sends a basic authorization in response to an authentication challenge:

```
public class BasicAuthAutenticator extends BasicAuthentication implements Authenticator {

    public BasicAuthAutenticator(String username, String password) {
        super(username, password);
    }

    @Override
    public Request authenticate(Proxy proxy, Response response) throws IOException {
        String credential = Credentials.basic(username, password);
        return response.request().newBuilder().header(Headers.AUTHORIZATION, credential).build();
    }

    @Override
    public Request authenticateProxy(Proxy proxy, Response response) throws IOException {
        return null;
    }
}
```

128.2 Sign Up Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Sign Up Screenlet creates a new user in your Liferay instance: a new user of your app can become a new user in your portal. You can also use this Screenlet to save new users' credentials on their devices. This enables auto login for future sessions. The Screenlet also supports navigation of form fields from the device's keyboard.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`UserService`	`addUser`	

Module

- Auth

Views

- Default
- Material

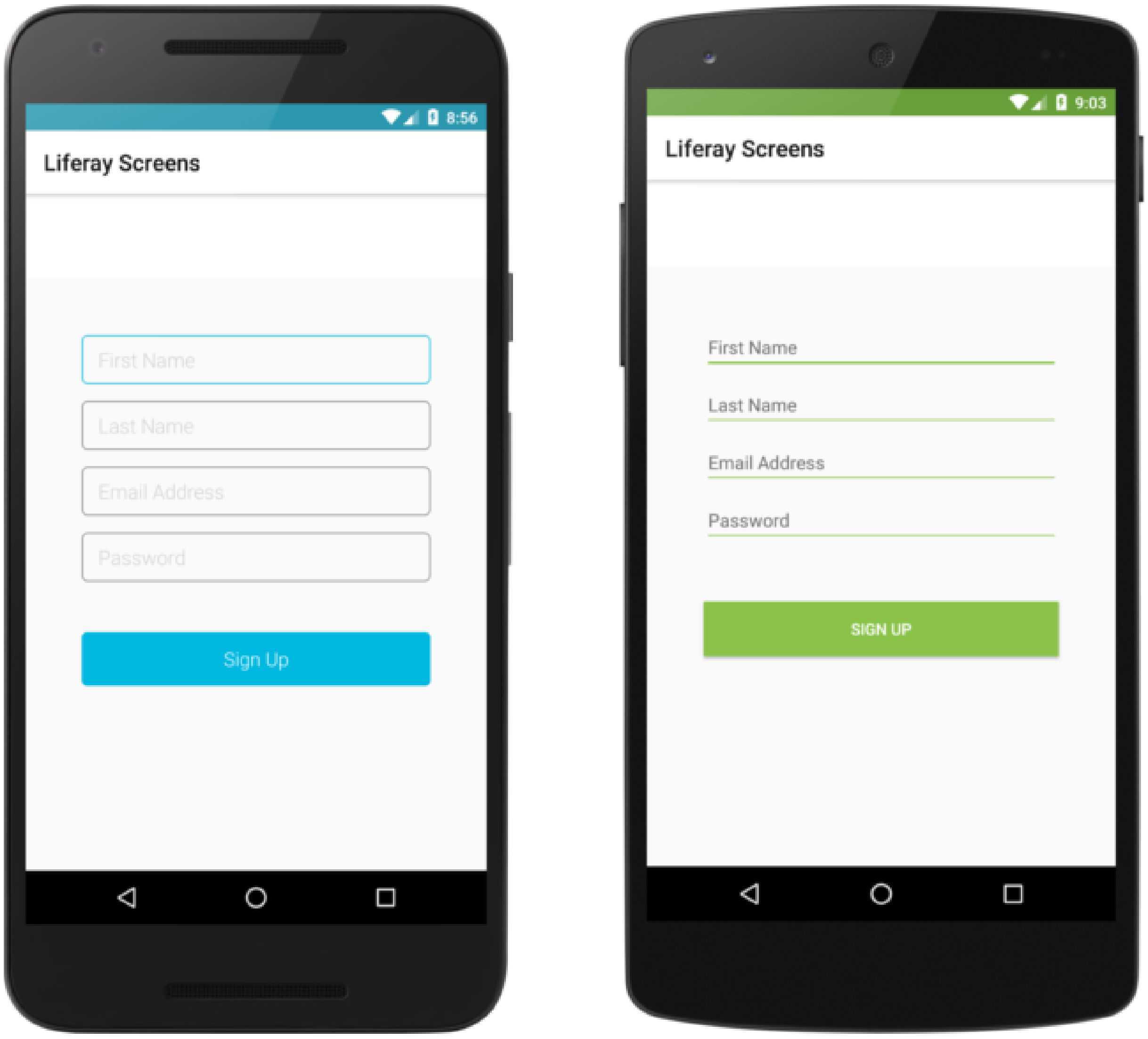
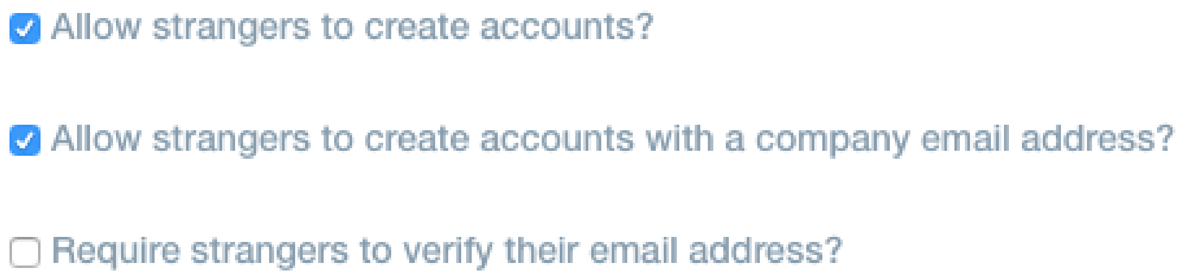


Figure 128.3: The Sign Up Screenlet with the Default (left) and Material (right) Viewsets.

Portal Configuration

Sign Up Screenlet's corresponding configuration in the Liferay instance can be set in the Control Panel by selecting *Configuration* → *Instance Settings*, and then selecting the *Authentication* section.



- [✓] Allow strangers to create accounts?
- [✓] Allow strangers to create accounts with a company email address?
- [ ] Require strangers to verify their email address?

Figure 128.4: The Liferay instance's authentication settings.

For more details, refer to the Setting up a Liferay Instance section of the User Guide.

Anonymous Requests

Anonymous requests are unauthenticated requests. Authentication is still required, however, to call the API. To allow this operation, the portal administrator should create a user with minimal permissions. To use Sign Up Screenlet, you need to use that user in your layout. You should add that user's credentials to `server_context.xml`.

Offline

This Screenlet doesn't support offline mode. It requires network connectivity.

Required Attributes

- `anonymousApiUserName`
- `anonymousApiPassword`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout used to show the View.| `anonymousApiUserName` | `string` | The user's name, email address, or ID to use for authenticating the request. The portal's authentication method defines which of these is used. | `anoymousApiPassword` | `string` | The password used to authenticate the request. | `companyId` | `number` | When set, a user in the specified company is authenticated. If not set, the company specified in `LiferayServerContext` is used. | `autoLogin` | `boolean` | Sets whether the user is logged in automatically after a successful sign up. | `credentialsStorage` | `enum` | Sets the mode for storing user credentials. The possible values are `none`, `auto`, and `shared_preferences`. If set to `shared_preferences`, the user credentials and attributes are stored using Android's `SharedPreferences` class. If set to `none`, user credentials and attributes aren't saved at all. If set to `auto`, the best of the available storage modes is used. Currently, this is equivalent to `shared_preferences`. The default value is `none`. |

`basicAuthMethod`|`enum`| Specifies the authentication method to use after a successful sign up. This must match the authentication method configured on the server. You can set this attribute to `email`, `screenName` or `userId`. The default value is `email`. |

Listener

The Sign Up Screenlet delegates some events to an object that implements the `SignUpListener` interface. This interface lets you implement the following methods:

- `onSignUpSuccess(User user)`: Called when sign up successfully completes. The `user` parameter contains a set of the created user's attributes. The supported keys are the same as those in the portal's User entity.
- `onSignUpFailure(Exception e)`: Called when an error occurs in the process.

128.3 Forgot Password Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Forgot Password Screenlet sends an email to registered users with their new passwords or password reset links, depending on the server configuration. The available authentication methods are

- Email address
- Screen name
- User id

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`UserService`	`sendPasswordByEmailAddress`	
`UserService`	`sendPasswordByUserId`	
`UserService`	`sendPasswordByScreenName`	

Module

- Auth

Views

- Default
- Material

Portal Configuration

To use Forgot Password Screenlet, the portal must be configured to allow users to request new passwords. The below sections show you how to do this. Also, Liferay Screens' Compatibility Plugin must be installed.

Authentication Method

The authentication method configured in the portal can be different from the one used by this Screenlet. For example, it's *perfectly fine* to use `screenName` for sign in authentication, but allow users to recover their password using the `email` authentication method.

Password Reset

You can set the Liferay instance's corresponding password reset options in the Control Panel by selecting *Configuration → Instance Settings*, and then selecting the *Authentication* section. The Screenlet's password functionality depends on the authentication settings in the portal:

If these options are both unchecked, password recovery is disabled. If both options are checked, an email containing a password reset link is sent when a user requests it. If only the first option is checked, an email containing a new password is sent when a user requests it.

For more details on authentication in Liferay Portal, please refer to the Setting up a Liferay Instance section of the User Guide.

Anonymous Request

An anonymous request can be made without the user being logged in. However, authentication is needed to call the API. To allow this operation, the portal administrator should create a specific user with minimal permissions.

Offline

This Screenlet doesn't support offline mode. It requires network connectivity.

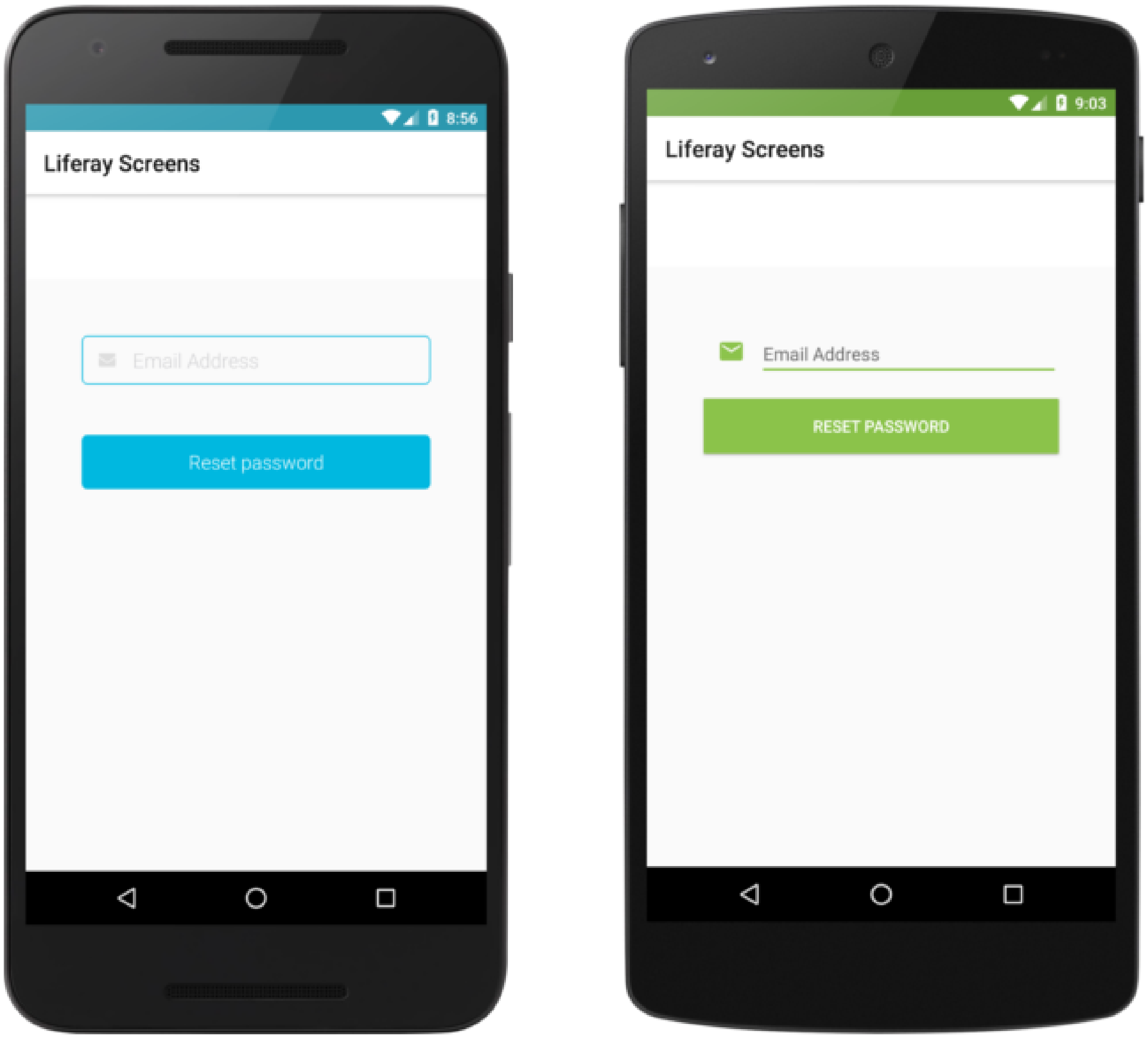
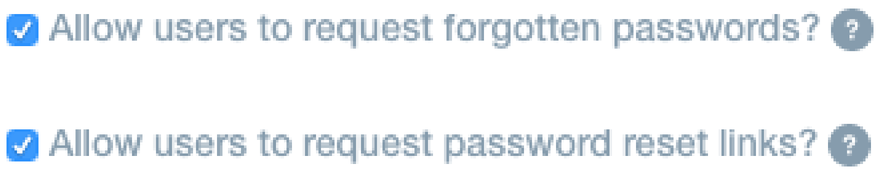


Figure 128.5: The Forgot Password Screenlet with the Default (left) and Material (right) Viewsets.



☑ Allow users to request forgotten passwords?

☑ Allow users to request password reset links?

Figure 128.6: Checkboxes for the password recovery features in your Liferay instance.

Required Attributes

- `anonymousApiUserName`
- `anonymousApiPassword`

Attributes

Attribute	Data type	Explanation
`layoutId`	`@layout`	The layout used to show the View.
`anonymousApiUserName`	`string`	The user name, email address, or `userId` to use for authenticating the request. This depends on the portal's authentication settings.
`anonymousApiPassword`	`string`	The password to use to authenticate the request.
`companyId`	`number`	When set, a user within the specified company is authenticated. If the value is set to `0`, the company specified in `LiferayServerContext` is used.
`basicAuthMethod`	`string`	The authentication method presented to the user. This can be `email`, `screenName`, or `userId`. The default value is `email`.

Listener

The Forgot Password Screenlet delegates some events to an object that implements the `ForgotPasswordListener` interface. This interface lets you implement the following methods:

- `onForgotPasswordRequestSuccess(boolean passwordSent)`: Called when a password reset email is successfully sent. The boolean parameter determines whether the email contains the new password or a password reset link.

- `onForgotPasswordRequestFailure(Exception e)`: Called when an error occurs in the process.

128.4 User Portrait Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Picasso library

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The User Portrait Screenlet shows the users' profile pictures. If a user doesn't have a profile picture, a placeholder image is shown. The Screenlet allows the profile picture to be edited via the `editable` property.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`UserService`	`getUserById`	

Module

- None

Views

- Default
- Material

Portal Configuration

No additional steps required.

Activity Configuration

The User Portrait Screenlet needs the following user permissions:

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture.

When loading the portrait, the Screenlet supports the following offline mode policies:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the user portrait from the portal. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet loads the portrait, it stores the received image in the local cache for later use. | Use this policy when you always need to show updated portraits, and show the default placeholder when there's no connection. | `CACHE_ONLY` | The Screenlet loads the user portrait from the local cache. If the portrait isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show local portraits, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the user portrait from the portal. The Screenlet displays the portrait to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the portrait from the local cache. If the portrait doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent portrait when connected, but show a potentially outdated version when there's no connection. | `CACHE_FIRST` | If the portrait exists in the local cache, the Screenlet loads it from there. If it doesn't exist

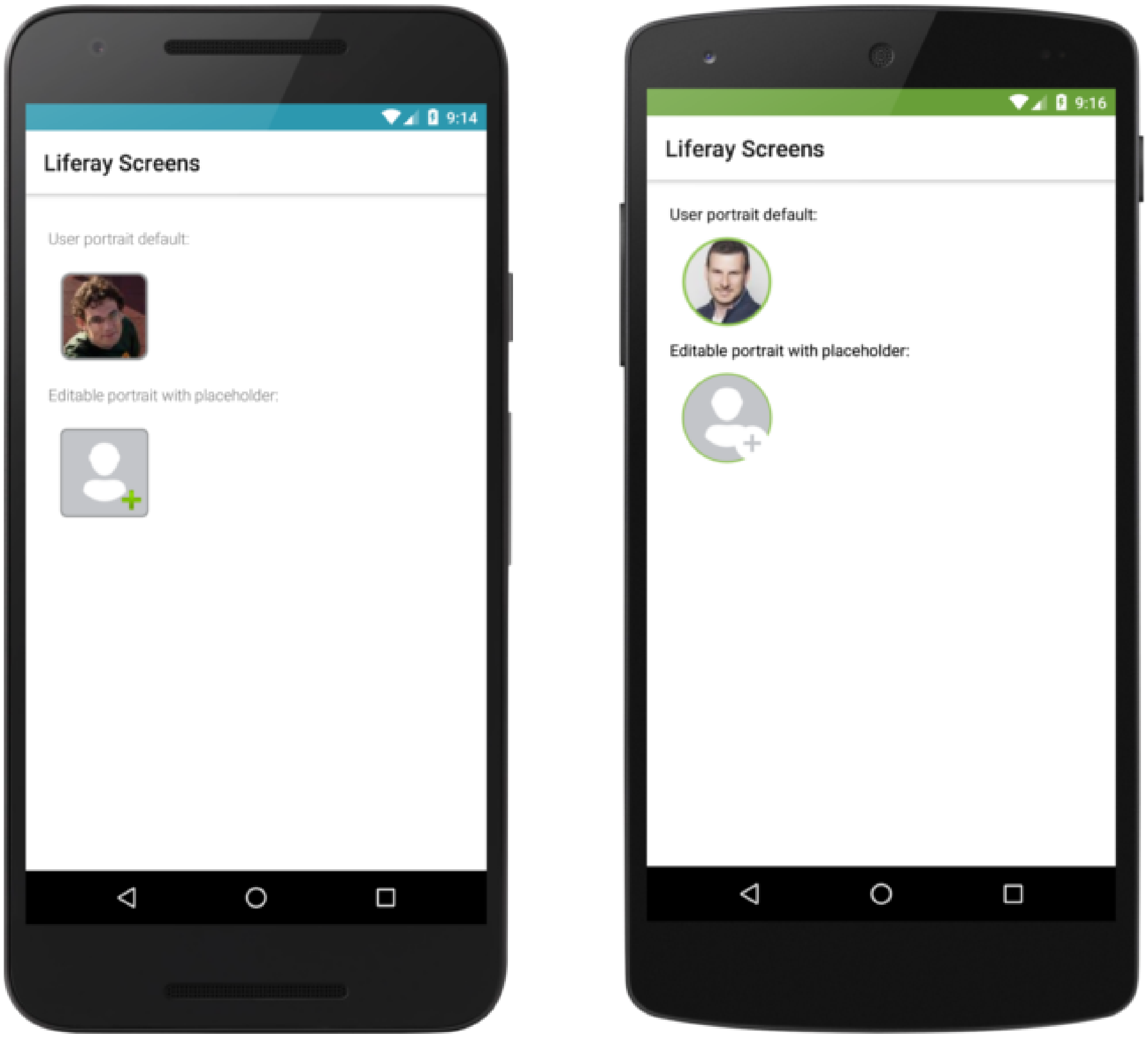


Figure 128.7: The User Portrait Screenlet using the Default (left) and Material (right) Views.

there, the Screenlet requests the portrait from the portal and uses the listener to notify the developer about any connection errors. | Use this policy to save bandwidth and loading time in the event a local (but probably outdated) portrait exists. |

When editing the portrait, the Screenlet supports the following offline mode policies:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet sends the user portrait to the portal. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error, but it also discards the new portrait. | Use this policy when you need to make sure portal always has the most recent version of the portrait. | `CACHE_ONLY` | The Screenlet stores the user portrait in the local cache. | Use this policy when you need to save the portrait locally, but don't want to change the portrait in the portal. | `REMOTE_FIRST` | The Screenlet sends the user portrait to the portal. If this succeeds, the Screenlet also stores the portrait in the local cache for later usage. If a connection issue occurs, the Screenlet stores the portrait in the local cache with the *dirty flag* enabled. This causes the portrait to be sent to the portal when the synchronization process runs. | Use this policy when you need to make sure the Screenlet sends the new portrait to the portal as soon as the connection is restored. | `CACHE_FIRST` | The Screenlet stores the user portrait in the local cache and then sends it to the portal. If a connection issue occurs, the Screenlet stores the portrait in the local cache with the *dirty flag* enabled. This causes the portrait to be sent to the portal when the synchronization process runs. | Use this policy when you need to make sure the Screenlet sends the new portrait to the portal as soon as the connection is restored. Compared to `REMOTE_FIRST`, this policy always stores the portrait in the cache. The `REMOTE_FIRST` policy only stores the new image in the cache in the event of a network error or a successful upload. |

Required Attributes

- None

Note that if you don't set any attributes, the Screenlet loads the logged-in user's portrait.

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout used to show the View. | `autoLoad` | `boolean` | Whether the portrait should load when the Screenlet is attached to the window. | `userId` | `number` | The ID of the user whose portrait is being requested. If this attribute is set, the `male`, `portraitId`, and `uuid` attributes are ignored. | `male` | `boolean` | Whether the default portrait placeholder shows a male or female outline. This attribute is used if `userId` isn't specified. | `portraitId` | `number` | The ID of the portrait to load. This attribute is used if `userId` isn't specified. | `uuid` | `string` | The `uuid` of the user whose portrait is being requested. This attribute is used if `userId` isn't specified. | `editable` | `boolean` | Lets the user change the portrait image by taking a photo or selecting a gallery picture. | `offlinePolicy` | `enum` | Configure the loading and saving behavior in case of connectivity issues. For more details, read the "Offline" section below. |

Methods

Method | Return | Explanation | `load()` | `void` | Starts the request to load the user specified in the `userId` property, or the portrait specified in the `portraitIdand` `uuid` properties. | `upload(int requestCode,Intent` `onActivityResultData)` | `void` | Starts the request to upload a profile picture from the source specified in the `requestCode` property (gallery or camera), and with the path stored in the `onActivityResultData` variable. |

Listener

The User Portrait Screenlet delegates some events to an object that implements the `UserPortraitListener` interface. This interface lets you implement the following methods:

- `onUserPortraitLoadReceived(Bitmap bitmap)`: Called when an image is received from the server. You can then apply image filters (grayscale, for example) and return the new image. You can return `null` or the original image supplied as the argument if you don't want to modify it.
- `onUserPortraitUploaded()`: Called when the user portrait upload service finishes.
- `error(Exception e, String userAction)`: Called when an error occurs in the process. For example, an error can occur when receiving or uploading a user portrait. The `userAction` argument distinguishes the specific action in which the error occurred.

128.5 DDL Form Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

DDL Form Screenlet shows a set of fields that can be filled in by the user. Initial or existing values can be shown in the fields. Fields of the following data types are supported:

- *Boolean*: A two state value typically represented by a checkbox.
- *Date*: A formatted date value. The format depends on the device's current locale.
- *Decimal, Integer, and Number*: A numeric value.
- *Documents & Media*: A file stored on the device. It can be uploaded to a specific portal repository.
- *Radio*: A set of options to choose from. A single option must be chosen.
- *Select*: A selection box of options to choose from. A single option must be chosen.
- *Text*: A single line of text.
- *Text Area*: Multiple lines of text.

The DDL Form Screenlet also supports the following features:

- Stored records can support a specific workflow.
- A Submit button can be shown at the end of the form.
- Required values and validation for fields can be used.
- Users can traverse the form fields from the keyboard.
- Supports i18n in record values and labels.

There are also a few limitations that you should be aware of when using DDL Form Screenlet. They are listed here:

- Nested fields in the data definition aren't supported.
- Selection of multiple items in the Radio and Select data types isn't supported.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensddlrecordService` (Screens compatibility plugin)	`getDDMStructureVersion`	Load form
`ScreensddlrecordService` (Screens compatibility plugin)	`getDdlRecord`	Load record
`DLAppService`	`addFileEntry`	Upload document
`DDLRecordService`	`addRecord`	Submit form
`DDLRecordService`	`updateRecord`	Update form

Module

- DDL

Views

- Default
- Material

The Default View uses a standard vertical `ScrollView` to show a scrollable list of fields. Other Views may use different components, such as `ViewPager` or others, to show the fields. You can find a sample of this implementation in the `DDLFormScreenletPagerView` class.

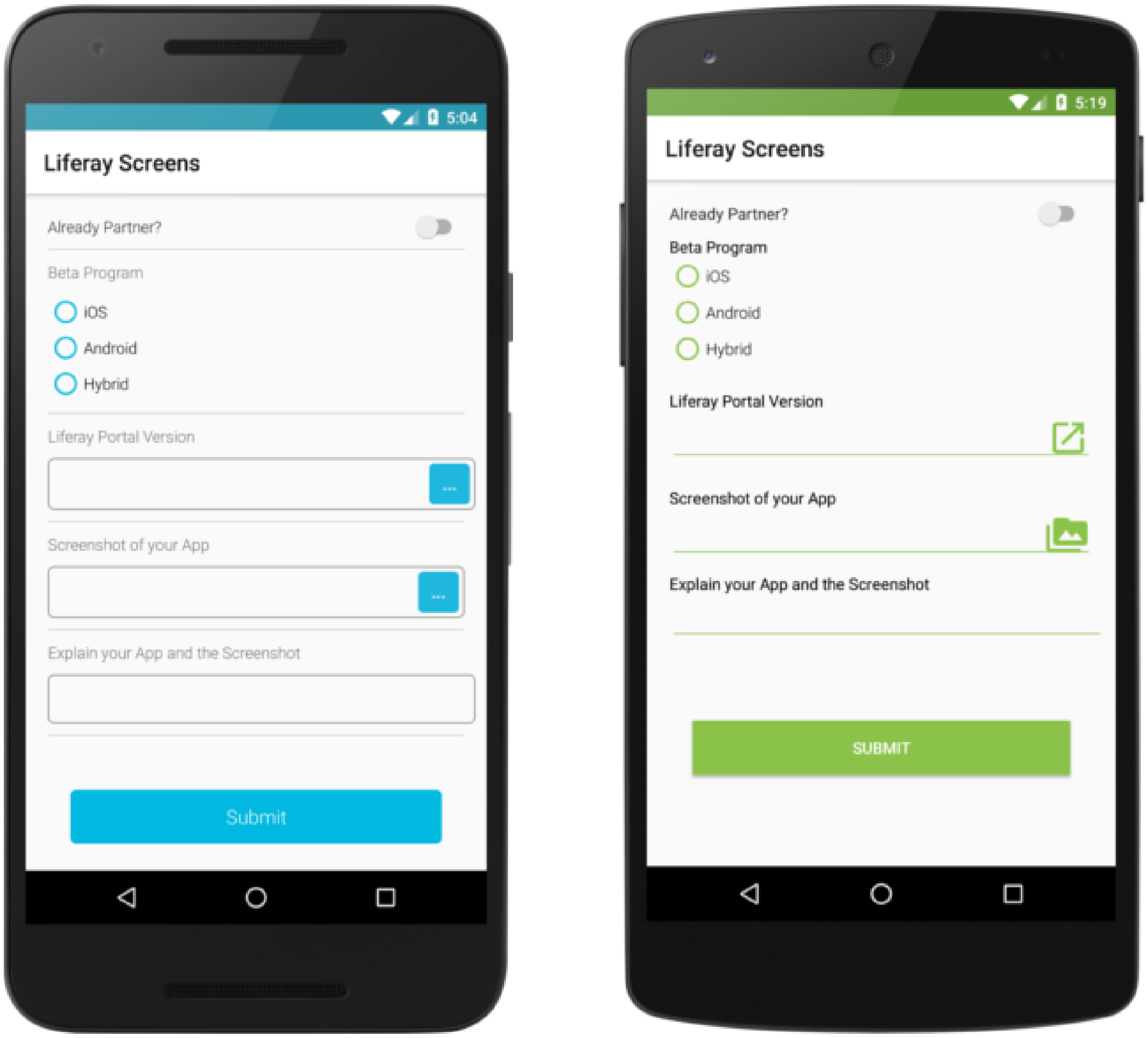


Figure 128.8: DDL Form Screenlet's Default (left) and Material (right) Views.

Editor Types

Each field defines an editor type. You must define each editor type's layout by using the following attributes:

- `checkboxFieldLayoutId`: The layout to use for Boolean fields.
- `dateFieldLayoutId`: The layout to use for Date fields.
- `numberFieldLayoutId`: The layout to use for Number, Decimal, or Integer fields.
- `radioFieldLayoutId`: The layout to use for Radio fields.
- `selectFieldLayoutId`: The layout to use for Select fields.
- `textFieldLayoutId`: The layout to use for Text fields.
- `textAreaFieldLayoutId`: The layout to use for Text Box fields.
- `textDocumentFieldLayoutId`: The layout to use for Documents & Media fields.

If you don't define the editor type's layout in DDL Form Screenlet's attributes, the default layout `ddlfield_xxx_default` is used, where `xxx` is the name of the editor type. It's important to note that you can change the layout used with any editor type at any point.

Custom Editors

If you want to have a unique appearance for one specific field, you can customize your field's editor View by calling the Screenlet's `setCustomFieldLayoutId(fieldName, layoutId)` method, where the first parameter is the name of the field to customize and the second parameter is the layout to use. You can also create custom editor Views. For examples of this, see the files `ddlfield_custom_rating_number.xml` and `CustomRatingNumberView.java`.

Activity Configuration

DDL Form Screenlet needs the following user permissions:

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Both are used by the Documents and Media fields to take a picture/video and store it locally before uploading it to the portal. The Documents and Media fields also need to override the `onActivityResult` method to receive the picture/video information. Here's an example implementation:

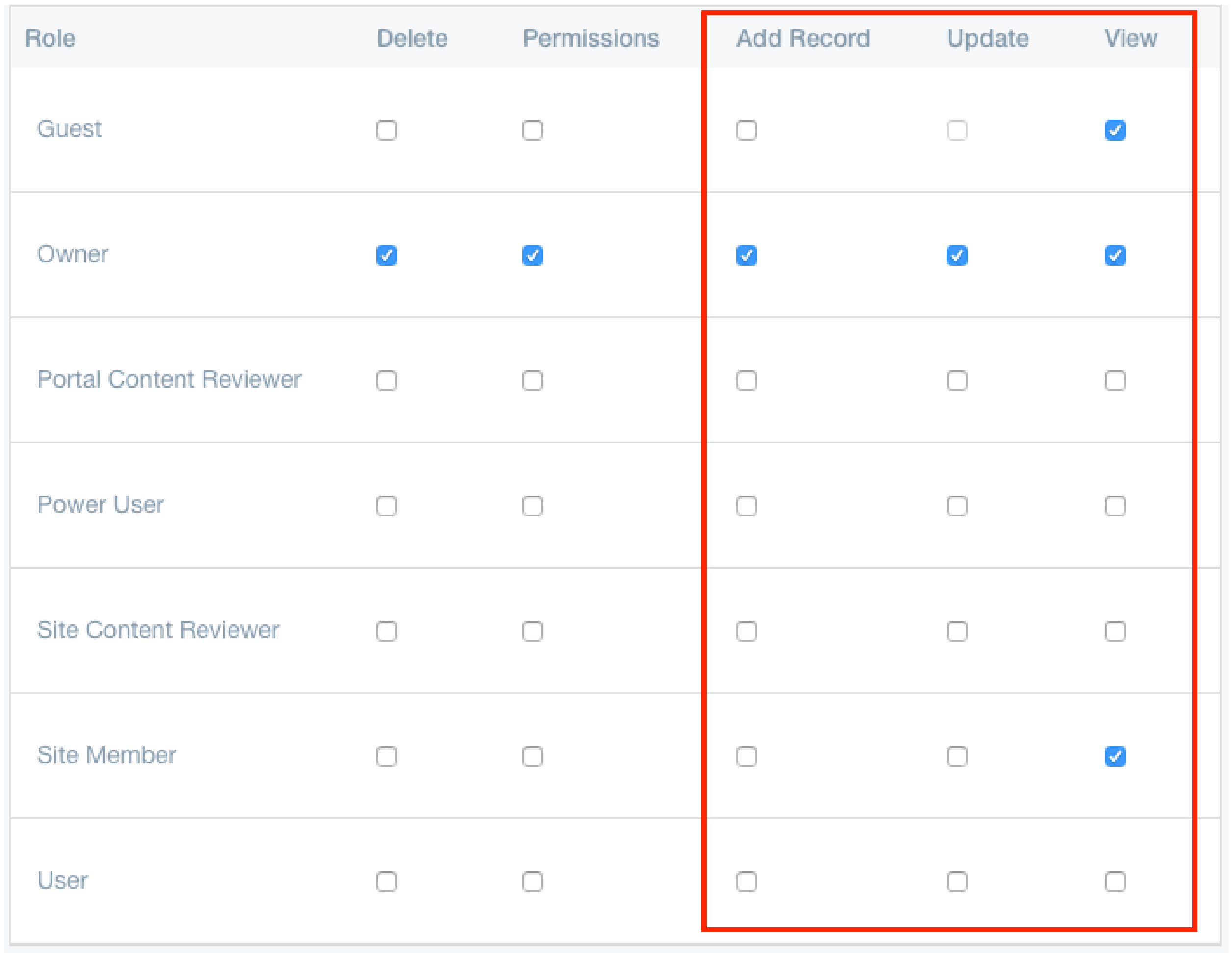
```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    screenlet.startUploadByPosition(requestCode);
}
```

Portal Configuration

Before using DDL Form Screenlet, you should make sure that Dynamic Data Lists and Data Types are configured properly in the portal. Refer to the Creating Data Definitions and Creating Data Lists sections of the User Guide for more details. If Workflow is required, it must also be configured. See the Using Workflow section of the User Guide for details.

Permissions

To use DDL Form Screenlet to add new records, you must grant the Add Record permission in the Dynamic Data List in the portal. If you want to use DDL Form Screenlet to view or edit record values, you must also grant the View and Update permissions, respectively. The Add Record, View, and Update permissions are highlighted by the red boxes in the following screenshot:



Role	Delete	Permissions	Add Record	Update	View
Guest	☐	☐	☐	☐	☑
Owner	☑	☑	☑	☑	☑
Portal Content Reviewer	☐	☐	☐	☐	☐
Power User	☐	☐	☐	☐	☐
Site Content Reviewer	☐	☐	☐	☐	☐
Site Member	☐	☐	☐	☐	☑
User	☐	☐	☐	☐	☐

Figure 128.9: The permissions for adding, viewing, and editing DDL records.

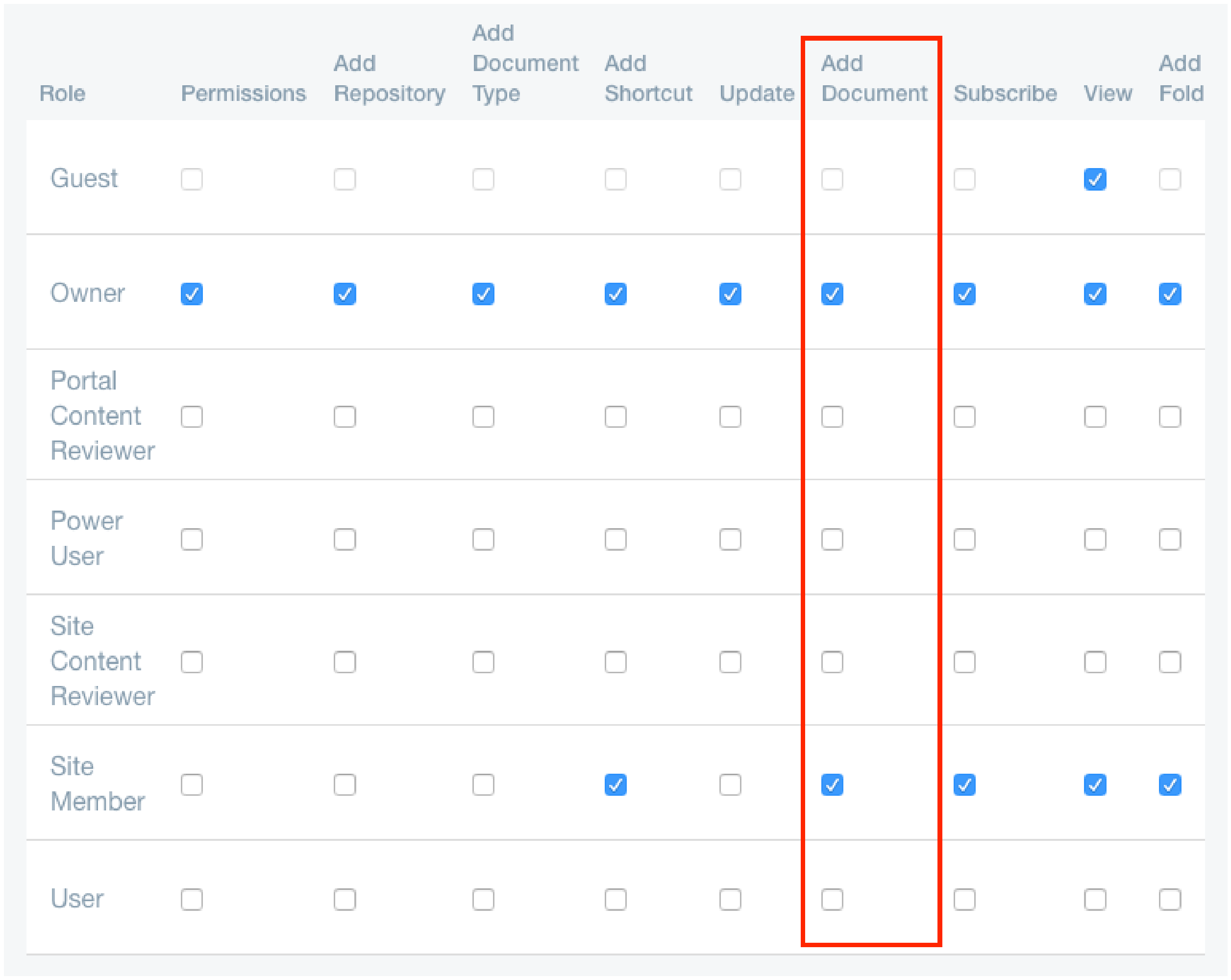
Also, if your form includes at least one Documents and Media field, you must grant permissions in the target repository and folder. For more details, see the `repositoryId` and `folderId` attributes below.

For more details, see the User Guide sections Creating Data Definitions, Creating Data Lists, and Using Workflow.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture.

When loading the form or record, the Screenlet supports the following offline mode policies:



Role	Permissions	Add Repository	Add Document Type	Add Shortcut	Update	Add Document	Subscribe	View	Add Fold
Guest	☐	☐	☐	☐	☐	☐	☐	✓	☐
Owner	✓	✓	✓	✓	✓	✓	✓	✓	✓
Portal Content Reviewer	☐	☐	☐	☐	☐	☐	☐	☐	☐
Power User	☐	☐	☐	☐	☐	☐	☐	☐	☐
Site Content Reviewer	☐	☐	☐	☐	☐	☐	☐	☐	☐
Site Member	☐	☐	☐	✓	☐	✓	✓	✓	✓
User	☐	☐	☐	☐	☐	☐	☐	☐	☐

Figure 128.10: The permission for adding a document to a Documents and Media folder.

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the form or record from the portal. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet loads the form or record, it stores the received data (record structure and data) in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the form or record from the local cache. If the form or record isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet requests the form or record from the portal. The Screenlet shows the record or form to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the form or record from the local cache. If the form or record doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | If the form or record exists in the local cache, the Screenlet loads it from there. If it doesn't exist there, the Screenlet requests it from the portal and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

When editing the record, the Screenlet supports the following offline mode policies:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet sends the record to the portal. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error, but it also discards the record. | Use this policy to make sure the portal always has the most recent version of the record. | `CACHE_ONLY` | The Screenlet stores the record in the local cache. | Use this policy when you need to save the data locally, but don't want to update the data in the portal (update or add record). | `REMOTE_FIRST` | The Screenlet sends the record to the portal. If this succeeds, it also stores the record in the local cache for later usage. If a connection issue occurs, then Screenlet stores the record in the local cache with the *dirty flag* enabled. This causes the synchronization process to send the record to the portal when it runs. | Use this policy when you need to make sure the Screenlet sends the record to the portal as soon as the connection is restored. | `CACHE_FIRST` | The Screenlet stores the record in the local cache and then sends it to the remote portal. If a connection issue occurs, then Screenlet stores the record in the local cache with the *dirty flag* enabled. This causes the the synchronization process to send the record to the portal when it runs. | Use this policy when you need to make sure the Screenlet sends the record to the portal as soon as the connection is restored. Compared to `REMOTE_FIRST`, this policy always stores the record in the cache. The `REMOTE_FIRST` policy only stores the record in the event of a network error. |

Required Attributes

- `structureId`
- `recordSetId`

Attributes

Attribute | Data Type | Explanation | `layoutId` | `@layout` | The layout to use to show the View. | `checkboxFieldLayoutId` | `@layout` | The layout to use to show the view for Boolean fields. | `dateFieldLayoutId` | `@layout` | The layout to use to show the view for Date fields. | `numberFieldLayoutId` | `@layout` | The layout to use to show the view for Number, Decimal, and Integer fields. | `radioFieldLayoutId` | `@layout` | The layout to use to show the view for Radio fields. | `selectFieldLayoutId` | `@layout` | The layout to use to show the view for Select fields. | `textFieldLayoutId` | `@layout` | The layout to use to show the view for Text fields. | `textAreaFieldLayoutId` | `@layout` | The layout to use to show the view for Text Box fields. | `documentFieldLayoutId` | `@layout` | The layout to use to show the view for Documents & Media fields. | `structureId` | `number` | The ID of a data definition in your Liferay site. To find the IDs for your data definitions, click *Admin → Content* from the Dockbar. Then click *Dynamic Data Lists* on the left and click the *Manage Data Definitions* button. The ID of each data definition is in the ID column of the table. | `groupId` | `number` | The ID of the site (group) where the record is stored. If this value is `0`, the `groupId` specified in `LiferayServerContext` is used. | `recordSetId` | `number` | A dynamic data list's ID. To find your dynamic data lists' IDs, click *Admin → Content* from the Dockbar. Then click *Dynamic Data Lists* on the left. Each dynamic data list's ID is in the ID column of the table. | `recordId` | `number` | The ID of the record you want to show. You can also allow the record's values to be edited. This ID can be obtained from other methods or listeners. | `repositoryId` | `number` | The ID of the Documents and Media repository to upload to. If this value is `0`, the default repository for the site specified by `groupId` is used. | `folderId` | `number` | The ID of the folder where Documents and Media files are uploaded. If this value is `0`, the root is used. | `filePrefix` | `string` | The prefix to attach to the names of files uploaded to a Documents and Media repository. The upload date followed by the original file name is appended following the prefix. | `autoLoad` | `boolean` | Sets whether the form loads when the Screenlet is shown. If `recordId` is set, the record value is loaded together with the form definition. The default value is `false`. | `autoScrollOnValidation` | `boolean` | Sets whether the form automatically scrolls to the first failed field when validation is used. The default value is `true`. | `showSubmitButton` | `boolean` | Sets whether the form shows a submit button at the bottom. If this is set to `false`, you should call the `submitForm()` method. The default value is `true`. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Methods

Method | Return Type | Explanation | `loadForm()` | `void` | Starts the request to load the form definition. The form fields are shown when the response is received. | `loadRecord()` | `void` | Starts the request to load the record specified by `recordId`. If needed, the form definition also loads. When the response is received, the form fields are shown filled with record values. | `load()` | `void` | Starts the request to load the record if `recordId` is specified. Otherwise, the form definition is loaded. | `submitForm()` | `void` | Starts the request to submit form values to the dynamic data list specified by `recordSetId`. If the record is new, a new record is added. If `loadRecord` is used to retrieve the record, or the record already exists, its values are updated. Fields are validated prior to the request. If validation fails, the validation errors are shown and the request is terminated. |

Listener

DDL Form Screenlet delegates some events to an object that implements to the `DDLFormListener` interface. This interface lets you implement the following methods:

- `onDDLFormLoaded(Record record)`: Called when the form definition successfully loads.
- `onDDLFormRecordLoaded(Record record, Map<String, Object> valuesAndAttributes)`: Called when the form record data successfully loads.
- `onDDLFormRecordAdded(Record record)`: Called when the form record is successfully added.
- `onDDLFormRecordUpdated(Record record)`: Called when the form record data successfully updates.
- `error(Exception e, String userAction)`: Called when an error occurs in the process. For example, this method is called when an error occurs while loading a form definition or record, or adding or updating a record. The `userAction` variable distinguishes these events.
- `onDDLFormDocumentUploaded(DocumentField field)`: Called when a specified document field's upload completes.
- `onDDLFormDocumentUploadFailed(DocumentField field, Exception e)`: Called when a specified document field's upload fails.

128.6 DDL List Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above

- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The DDL List Screenlet has the following features:

- Shows a scrollable collection of Dynamic Data List (DDL) records.
- Implements fluent pagination with configurable page size.
- Allows record filtering by creator.
- Supports i18n in record values.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensddlrecordService` (Screens compatibility plugin)	`getDdlRecords`	With `ddlRecordSetId`, or `ddlRecordSetId` and `userId`
`ScreensddlrecordService` (Screens compatibility plugin)	`getDdlRecordsCount`	

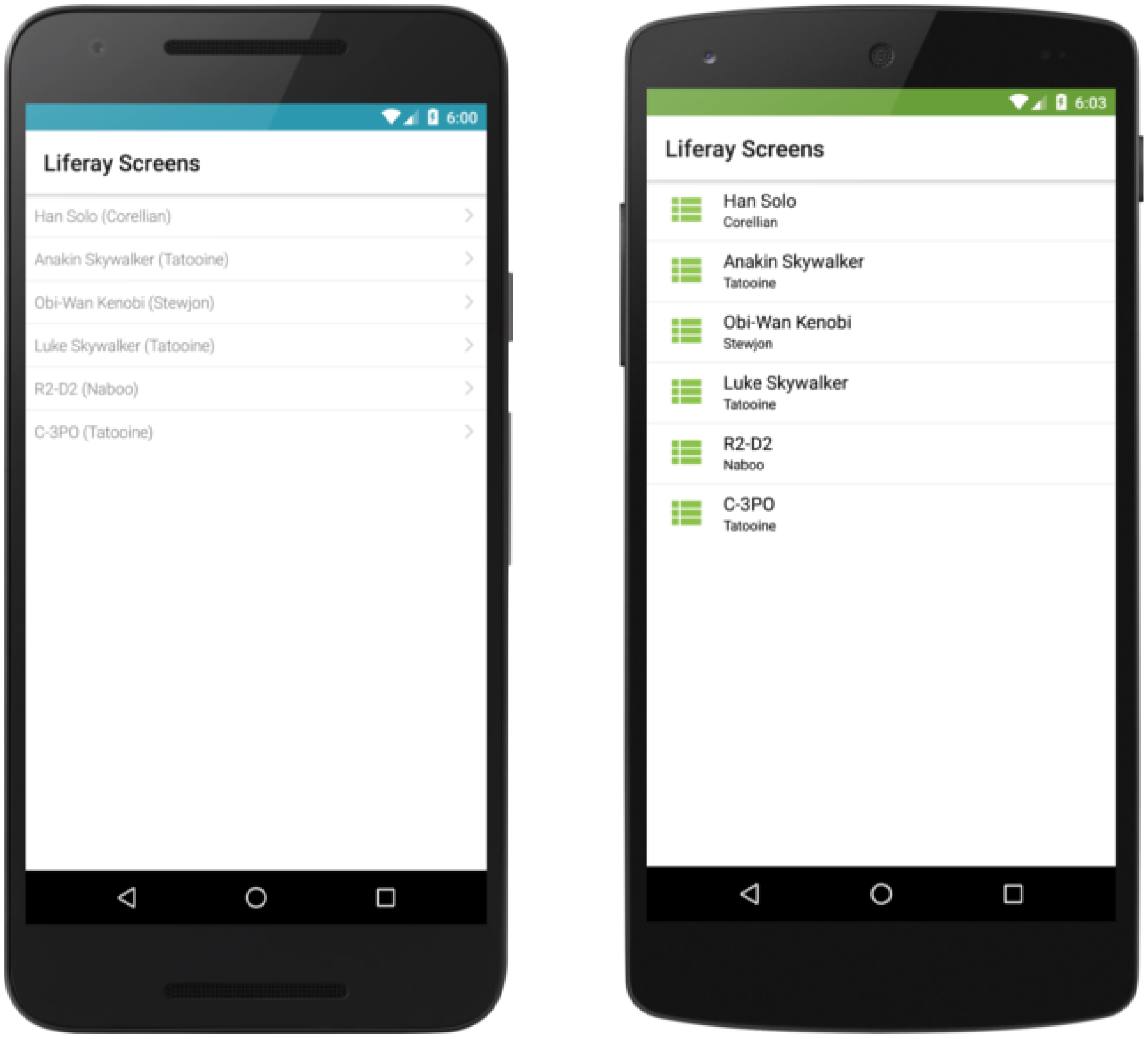
Module

- DDL

Views

- Default
- Material

The Default View uses a standard `RecyclerView` to show the scrollable list. Other Views may use a different component, such as `ViewPager` or others, to show the items.


6:00

Liferay Screens

Han Solo (Corellian)

Anakin Skywalker (Tatooine)

Obi-Wan Kenobi (Stewjon)

Luke Skywalker (Tatooine)

R2-D2 (Naboo)

C-3PO (Tatooine)

6:03

Liferay Screens

Han Solo
Corellian

Anakin Skywalker
Tatooine

Obi-Wan Kenobi
Stewjon

Luke Skywalker
Tatooine

R2-D2
Naboo

C-3PO
Tatooine

Figure 128.11: The DDL List Screenlet using the Default and Material Views.

Portal Configuration

DDLs and Data Types should be configured in the portal before using DDL List Screenlet. For more details, see the Liferay User Guide sections Creating Data Definitions and Creating Data Lists .

Also, to allow remote calls without the `userId`, the Liferay Screens Compatibility app must be installed in your Liferay instance. You can find this app on Liferay Marketplace.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture.

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the list from the portal. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the list, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the list from the portal. If this succeeds, the Screenlet shows the list to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the list from the local cache. If the list doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet requests it from the portal and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `recordSetId`
- `labelFields`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View. | `autoLoad` | `boolean` | Defines whether the list should be loaded when it's presented on the screen. The default value is `true`. | `recordSetId` | `number` | The ID of the DDL being called. To find your DDLs' IDs, click *Admin* → *Content* from the Dockbar. Then click *Dynamic Data Lists* on the left. Each DDL's ID is in the ID column of the table. | `userId` | `number` | The ID of the user to filter records on. Records aren't filtered if the `userId` is `0`. The default value is `0`. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. | `firstPageSize` | `number` | The number of items to retrieve from the server for display on the first page. The default value is `50`. | `pageSize` | `number` | The number of items to retrieve from the server for display on the second and subsequent pages. The default value is `25`. | `labelFields` | `string` | The comma-separated names of the DDL fields to show. Refer to the list's data definition to find the field names. For more information on this, see Creating Data Definitions. Note that the appearance of these values in your app depends on the `layoutId` set. | `obcClassName` | `string` | The name of the `OrderByComparator` class to use to sort the results. Omit this property if you don't want to sort the results. Click here to see some comparator classes. Note, however, that not all of these classes can be used with `obcClassName`. You can only use comparator classes that extend `OrderByComparator<DDLRecord>`. You can also create your own comparator classes that extend `OrderByComparator<DDLRecord>`. |

Methods

Method | Return | Explanation | `loadPage(pageNumber)` | `void` | Starts the request to load the specified page of records. The page is shown when the response is received. |

Listener

DDL List Screenlet delegates some events to an object or a class that implements the `BaseListListener` interface. This interface lets you implement the following methods:

- `onListPageFailed(int startRow, Exception e)`: Called when the server call to retrieve a page of items fails. This method's arguments include the `Exception` generated when the server call fails.

- `onListPageReceived(int startRow, int endRow, List<Record> records, int rowCount)`: Called when the server call to retrieve a page of items succeeds. Note that this method may be called more than once; once for each page received. Because `startRow` and `endRow` change for each page, a `startRow` of `0` corresponds to the first item on the first page.

- `onListItemSelected(Record records, View view)`: Called when an item is selected in the list. This method's arguments include the selected list item (`Record`).

- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.7 Asset List Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Asset List Screenlet can be used to show asset lists from a Liferay instance. For example, you can use the Screenlet to show a scrollable list of assets. It also implements fluent pagination with configurable page size. The Asset List Screenlet can show assets belonging to the following classes:

- `BlogsEntry`
- `BookmarksEntry`
- `BookmarksFolder`
- `CalendarEvent`
- `DLFileEntry`
- `DDLRecord`
- `DDLRecordSet`
- `Group`
- `JournalArticle` (Web Content)
- `JournalFolder`
- `Layout`
- `LayoutRevision`
- `MBThread`
- `MBCategory`
- `MBDiscussion`
- `MBMailingList`
- `Organization`
- `User`
- `WikiPage`
- `WikiPageResource`
- `WikiNode`

The Asset List Screenlet also supports i18n in asset values.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensddlrecordService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensddlrecordService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`
`AssetEntryService`	`getEntriesCount`	

Module

- None

Views

- Default
- Material

The Default Views use a standard `RecyclerView` to show the scrollable list. Other Views may use a different component, such as `ViewPager` or others, to show the items.

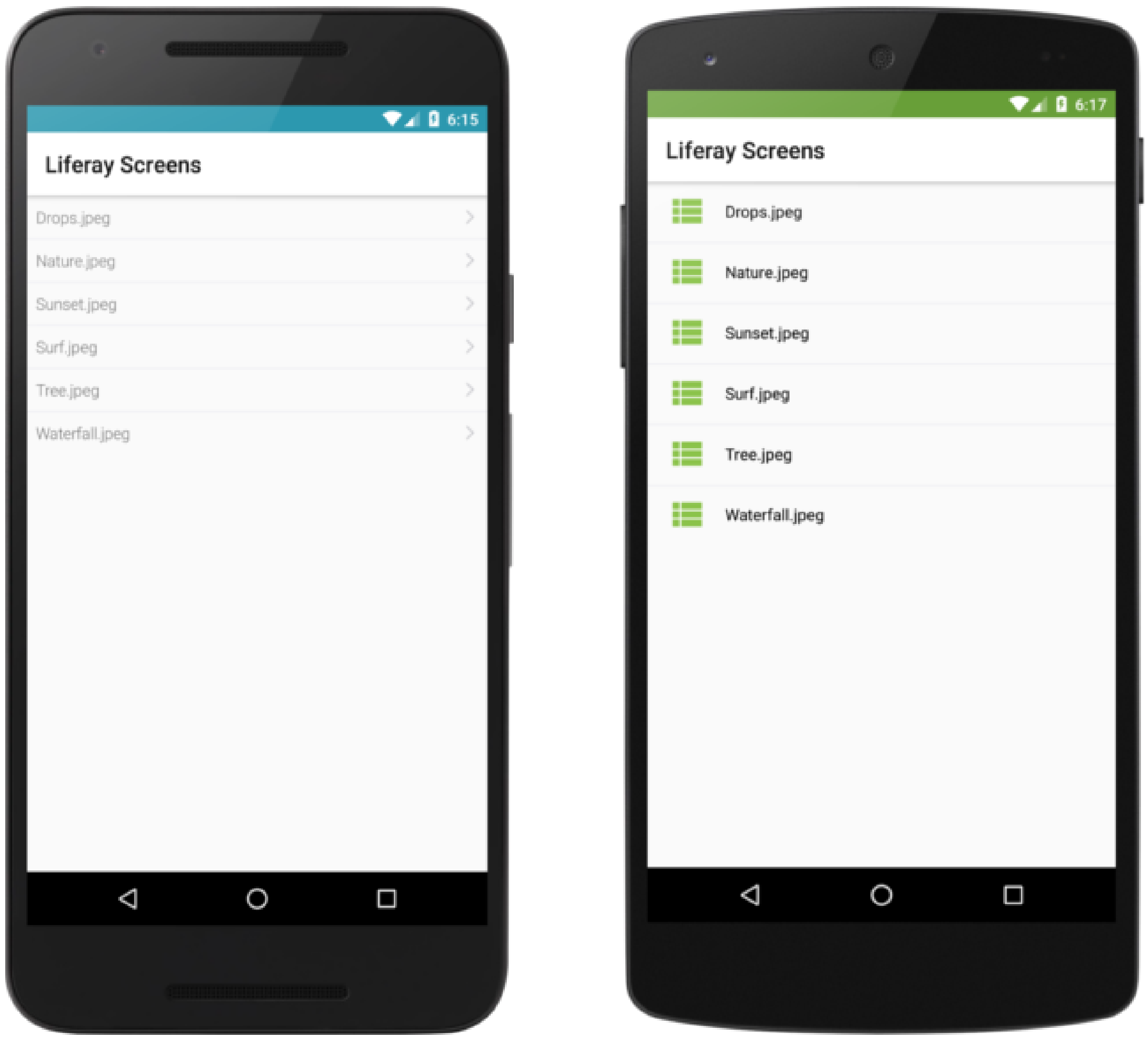


Figure 128.12: Asset List Screenlet using the Default (left) and Material (right) Views.

Portal Configuration

Dynamic Data Lists (DDL) and Data Types should be configured properly in the portal. Refer to the Creating Data Definitions
and Creating Data Lists sections of the User Guide for more details.

Also, to allow remote calls without the `userId`, the Liferay Screens Compatibility app must be installed in your Liferay instance. You can find this app on Liferay Marketplace.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture.

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the list from the portal. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the list, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the list from the portal. If this succeeds, the Screenlet shows the list to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the list from the local cache. If the list doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet requests it from the portal and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `classNameId`

If you don't set `classNameId`, you must set this attribute instead:

- `portletItemName`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View.| `autoLoad` | `boolean` | Whether the list should be loaded when it's presented on the screen. The default value is `true`. | `groupId` | `number` | The asset's group (site) ID. If this value is `0`, the `groupId` specified in `LiferayServerContext` is used. The default value is `0`. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. | `portletItemName` | `string` | The name of the configuration template you used in the Asset Publisher. To use this feature, add an Asset Publisher to one of your site's pages (it may be a hidden page), configure the Asset Publisher's filter (in *Configuration → Setup → Asset Selection*), and then use the Asset Publisher's *Configuration Templates* option to save this configuration with a name. Use this name in this attribute. | `classNameId` | `number` | The asset class name's ID. Use values from the portal's `classname_` database table. | `firstPageSize` | `number` | The number of items to retrieve from the server for display on the list's first page. The default value is `50`. | `pageSize` | `number` | The number of items to retrieve from the server for display on the second and subsequent pages. The default value is `25`. | `labelFields` | `string` | The comma-separated names of the DDL fields to show. Refer to the list's data definition to find the field names. For more information on this, see Creating Data Definitions. Note that the appearance of these values in your app depends on the layoutId set. | customEntryQuery | HashMap | The set of keys (string) and values (string or number) to be used in the AssetEntryQuery object. These values filter the assets returned by the Liferay instance. |

Methods

Method | Return | Explanation | loadPage(pageNumber) | void | Starts the request to load the specified page of assets. The page is shown when the response is received. |

Listener

Asset List Screenlet delegates some events to an object or a class that implements the BaseListListener interface. This interface lets you implement the following methods:

- `onListPageFailed(int startRow, Exception e)`: Called when the server call to retrieve a page of items fails. This method's arguments include the `Exception` generated when the server call fails.

- `onListPageReceived(int startRow, int endRow, List<Model> entries, int rowCount)`: Called when the server call to retrieve a page of items succeeds. Note that this method may be called more than once; once for each page received. Because `startRow` and `endRow` change for each page, a `startRow` of `0` corresponds to the first item on the first page.

- `onListItemSelected(Model entries, View view)`: Called when an item is selected in the list. This method's arguments include the selected list item (`Model`).

- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.8 Web Content Display Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Web Content Display Screenlet shows web content elements in your app, rendering the web content's inner HTML. The Screenlet also supports i18n, rendering contents differently depending on the device's locale.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`DDMStructureService`	`getStructure`	
`JournalArticleService`	`getArticle`	
`JournalArticleService`	`getArticleContent`	
`ScreensddlrecordService` (Screens compatibility plugin)	`getJournalArticleContent`	With `entryQuery`

Module

- None

Views

- Default

  The Default View uses a standard `WebView` to render the HTML.

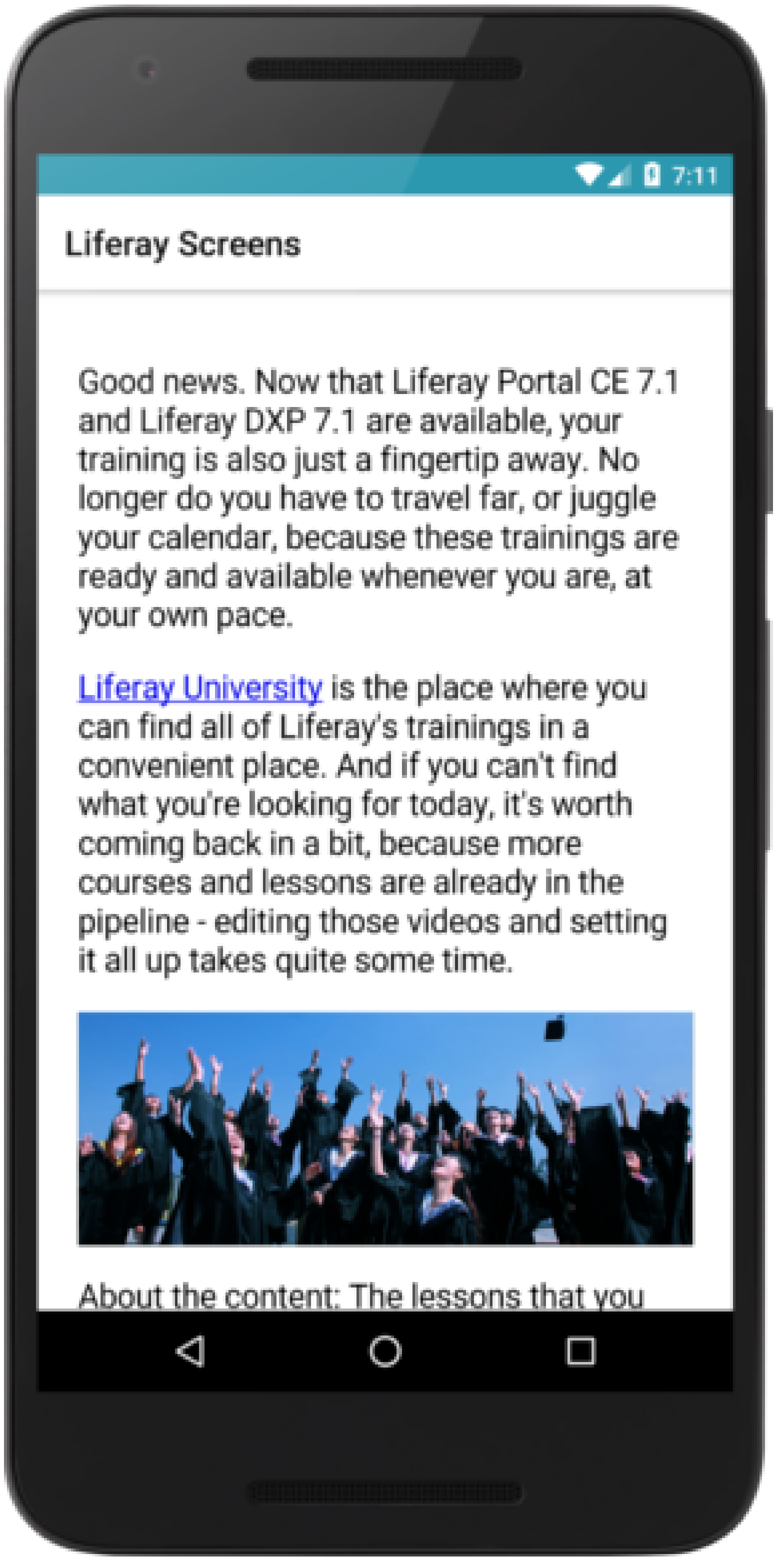
Portal Configuration

For the Web Content Display Screenlet to function properly, there should be web content in the Liferay instance your app connects to. For more details on web content, see the Creating Web Content section of the Liferay User Guide.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the content from the portal. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the content, it stores the data in the local cache for later use. | Use this policy when you always need to show updated content, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the content from the local cache. If the content isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local content, without



Good news. Now that Liferay Portal CE 7.1 and Liferay DXP 7.1 are available, your training is also just a fingertip away. No longer do you have to travel far, or juggle your calendar, because these trainings are ready and available whenever you are, at your own pace.

Liferay University is the place where you can find all of Liferay's trainings in a convenient place. And if you can't find what you're looking for today, it's worth coming back in a bit, because more courses and lessons are already in the pipeline - editing those videos and setting it all up takes quite some time.

About the content: The lessons that you

Figure 128.13: Web Content Display Screenlet using the Default View.

retrieving remote content under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the content from the portal. If this succeeds, the Screenlet shows the content to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the content from the local cache. If the content doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the content when connected, but show a possibly outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the content from the local cache. If the content isn't there, the Screenlet requests it from the portal and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) content. |

Required Attributes

- `articleId`

Note that if your web content uses structures and templates, you can use `templateId` or `structureId` in conjunction with `articleId`.

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout used to show the View. | `groupId` | `number` | The site (group) identifier where the asset is stored. If this value is `0`, the `groupId` specified in `LiferayServerContext` is used. | `articleId` | `string` | The identifier of the web content to display. You can find the identifier by clicking *Edit* on the web content in the portal. | `classPK` | `number` | The corresponding asset's class primary key. If the web content is an asset (from Asset List Screenlet, for example), this is the asset's identifier. This attribute is used only if `articleId` is empty. | `templateId` | `number` | The identifier of the template used to render the web content. This only applies to structured web content. | `structureId` | `number` | The identifier of the `DDMStructure` used to model the web content. This parameter lets the Screenlet retrieve and parse the structure. | `labelFields` | `string` | A comma-delimited list of `DDMStructure` fields to display in the Screenlet. | `autoLoad` | `boolean` | Whether the content should be retrieved from the portal as soon as the screenlet appears. Default value is `true`. | `javascriptEnabled` | `boolean` | Enables support for JavaScript. This is disabled by default. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Methods

Method | Return | Explanation | `load()` | `void` | Starts the request to load the web content. The HTML is rendered when the response is received. | `getLocalized(String name)` | `String` | Returns the value, according to the device locale, of a field of the `DDMStructure` used to render the web content.

Listener

The Web Content Display Screenlet delegates some events to an object that implements the `WebContentDisplayListener` interface. This interface lets you implement the following methods:

- `onWebContentReceived(WebContent webContent)`: Called when the web content's HTML or `DDMStructure` is received. The HTML is available by calling the `getHtml` method. To make some adaptations, the listener may return a modified version of the HTML. The original HTML is rendered if the listener returns `null`.
- `onUrlClicked(String url)`: Called when a URL is clicked. Return `true` to replace the default behavior, or `false` to load the url.
- `onWebContentTouched(View view, MotionEvent event)`: Called when something is touched in the web content. Return `true` to replace the default behavior, or `false` to keep processing the event.
- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.9 Web Content List Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Web Content List Screenlet has the following features:

- Shows a scrollable collection of web content articles.
- Implements fluent pagination with configurable page size.
- Supports i18n in web content values.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`JournalArticleService`	`getJournalArticles`	
`JournalArticleService`	`getJournalArticlesCount`	

Module

- None

Views

- Default

The Default View uses a standard `RecyclerView` to show the scrollable list. Other Views may use a different component, such as `ViewPager` or others, to show the items.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the list from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the list, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the list from the Liferay instance. If this succeeds, the Screenlet shows the list to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the list from the local cache. If the list doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show a possibly outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but possibly outdated) data. |

Required Attributes

- `folderId`
- `labelFields`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The ID of the layout to use to show the View. | `autoLoad` | `boolean` | Whether the list loads automatically when the Screenlet appears in the app's UI. The default value is `true`. | `folderId` | `number` | The ID of the web content folder to retrieve content from. | `groupId` | `number` | The ID of the site (group) where the asset is stored. If set to `0`, the `groupId` specified in `LiferayServerContext` is used. The default value is `0`. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. | `firstPageSize` | `number` | The number of items to retrieve from the server for display on the first page. The default value is `50`. | `pageSize` | `number` | The number of items to retrieve from the server for display on the second and subsequent pages. The default value is `25`. | `labelFields` |

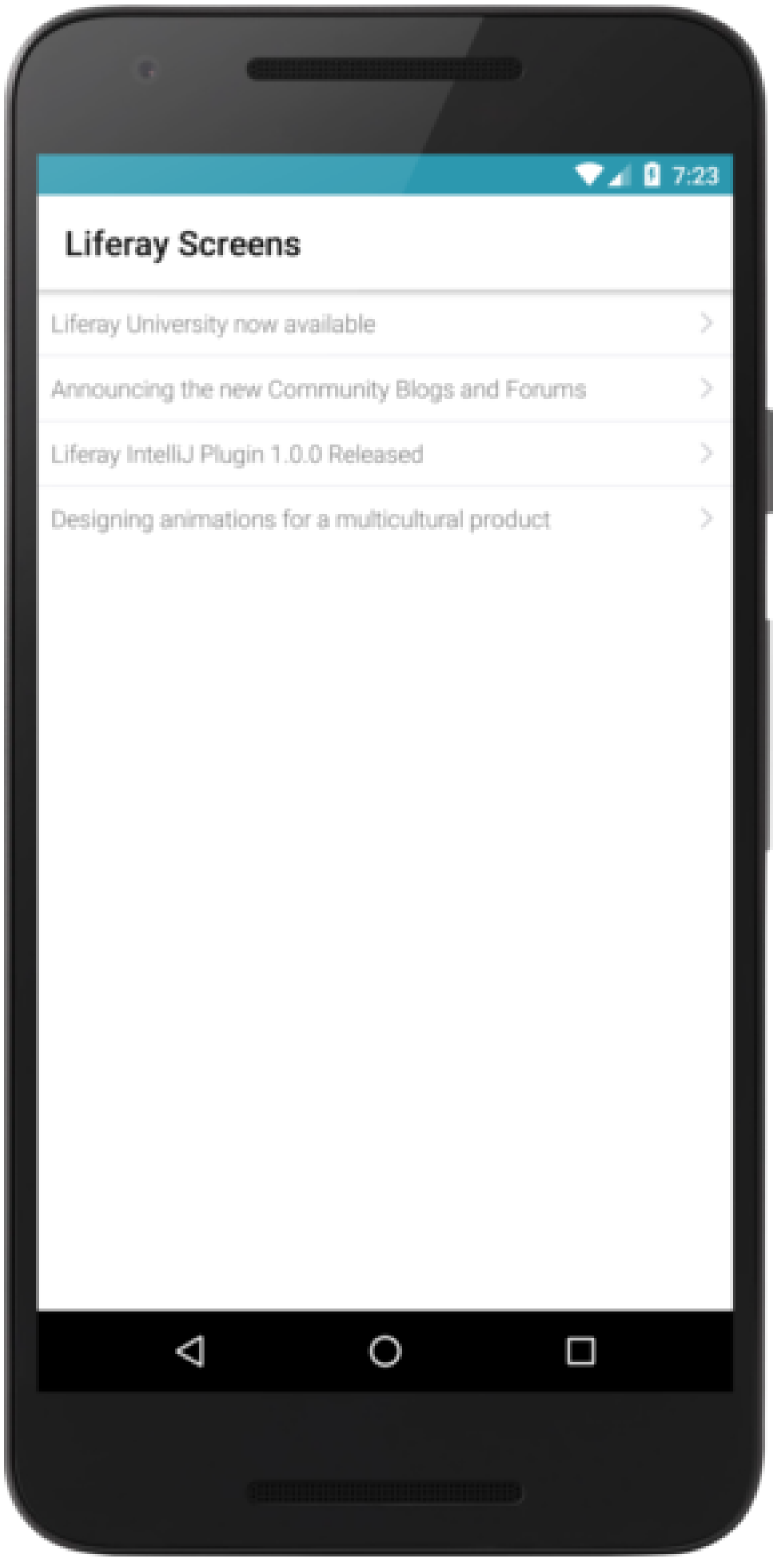

7:23
Liferay Screens
Liferay University now available
Announcing the new Community Blogs and Forums
Liferay IntelliJ Plugin 1.0.0 Released
Designing animations for a multicultural product

Figure 128.14: The Web Content List Screenlet using the Default View.

`string` | The comma-separated names of the DDM fields to show. Refer to the list's data definition to find the field names. For more information on this, see the article on structured web content. Note that the appearance of data from a structure's fields depends on the `layoutId`. | `obcClassName` | `string` | The name of the `OrderByComparator` class to use to sort the results. Omit this property if you don't want to sort the results. Click here to see some comparator classes. Note, however, that not all of these classes can be used with `obcClassName`. You can only use comparator classes that extend `OrderByComparator<JournalArticle>`. You can also create your own comparator classes that extend `OrderByComparator<JournalArticle>`. |

Methods

Method | Return | Explanation | `loadPage(pageNumber)` | `void` | Starts the request to load the specified page of records. The page is shown when the response is received. |

Listener

Web Content List Screenlet delegates some events to an object or a class that implements the `BaseListListener` interface. This interface lets you implement the following methods:

- `onListPageFailed(int startRow, Exception e)`: Called when the server call to retrieve a page of items fails. This method's arguments include the `Exception` generated when the server call fails.

- `onListPageReceived(int startRow, int endRow, List<Record> records, int rowCount)`: Called when the server call to retrieve a page of items succeeds. Note that this method may be called more than once; once for each page received. Because `startRow` and `endRow` change for each page, a `startRow` of `0` corresponds to the first item on the first page.

- `onListItemSelected(Record records, View view)`: Called when an item is selected in the list. This method's arguments include the selected list item (`Record`).

128.10 Image Gallery Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Image Gallery Screenlet shows a list of images from a Documents and Media folder in a Liferay instance. You can also use Image Gallery Screenlet to upload images to and delete images from the same folder. The Screenlet implements fluent pagination with configurable page size, and supports i18n in asset values.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`DLAppService`	`getFileEntries`	Load
`DLAppService`	`getFileEntriesCount`	
`DLAppService`	`addFileEntry`	Upload
`DLAppService`	`deleteFileEntry`	Delete

Module

- None

Views

The included Views use a standard Android `RecyclerView` to show the scrollable list. Other custom Views may use a different component, such as `ViewPager` or others, to show the items.

This Screenlet has three different Views:

1. Grid (default)
2. Slideshow
3. List

Offline

This Screenlet supports offline mode so it can function without a network connection when loading or uploading images (deleting images while offline is unsupported). For more information on how offline mode works, see the tutorial on its architecture. This Screenlet supports the `REMOTE_ONLY`, `CACHE_ONLY`, `REMOTE_FIRST`, and `CACHE_FIRST` offline mode policies.

These policies take the following actions when loading images from a Liferay instance:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the list from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the list, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving

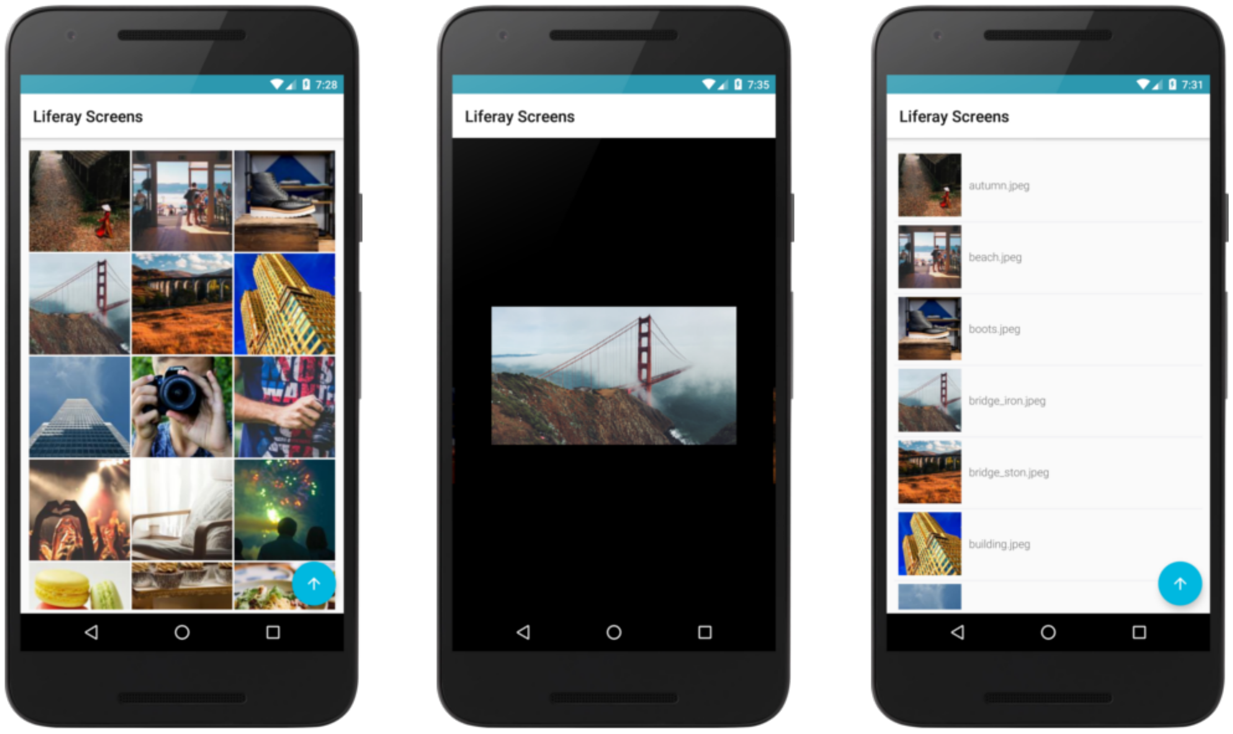


Figure 128.15: Image Gallery Screenlet using the Grid, Slideshow, and List Views.

remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the list from the Liferay instance. If this succeeds, the Screenlet shows the list to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the list from the local cache. If the list doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

These policies take the following actions when uploading an image to a Liferay instance:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet sends the image to the Liferay instance. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error, but it also discards the image. | Use this policy to make sure the Liferay instance always has the most recent version of the image. | `CACHE_ONLY` | The Screenlet stores the image in the local cache. | Use this policy when you need to save the image locally, but don't want to update the image in the Liferay instance (delete or add image). | `REMOTE_FIRST` | The Screenlet sends the image to the Liferay instance. If this succeeds, it also stores the image in the local cache for later use. If a connection issue occurs, the Screenlet stores the image in the local cache and sends it to the Liferay instance when the connection is re-established. | Use this policy when you need to make sure the Screenlet sends the image to the Liferay instance as soon as the connection is restored. | `CACHE_FIRST` | The Screenlet stores the image in the local cache and then attempts to send it to the Liferay instance. If a connection issue occurs, the Screenlet sends the image to the Liferay instance when the connection is re-established. | Use this policy when you need to make sure the Screenlet sends the image to the Liferay instance as soon as the connection is restored. Compared to `REMOTE_FIRST`, this policy always stores the image in the cache. The `REMOTE_FIRST` policy only stores the image in the event of a network error. |

Required Attributes

- `folderId`
- `repositoryId`

Attributes

Attribute | Data type | Explanation | `repositoryId` | `number` | The ID of the Liferay instance's Documents and Media repository that contains the image gallery. If you're using a site's default Documents and Media repository, then the `repositoryId` matches the site ID (`groupId`). | `folderId` | `number` | The ID of the Documents and Media repository folder that contains the image gallery. When accessing the folder in your browser, the `folderId` is at the end of the URL. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. | `firstPageSize` | `number` | The number of items to display on the first page. The default value is `50`. | `pageSize` | `number` | The number of items to display on second and subsequent pages. The default value is `25`. | `mimeTypes` | `string` | The comma-separated list of MIME types for the Screenlet to support. | `autoLoad` | `boolean` | Whether the list automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `layoutId` | `@layout` | The layout to use to show the View. | `obcClassName` | `string` | The name of the `OrderByComparator` class to use to sort the results. Omit this property if you don't want to sort the results. Note that you can only use comparator classes that extend `OrderByComparator<DLFileEntry>`. Liferay contains no such comparator classes. You must therefore create your own by extending `OrderByComparator<DLFileEntry>`. To see examples of some comparator classes that extend other Document Library classes, click here. |

Methods

Method | Return | Explanation | `loadPage(pageNumber)` | `void` | Starts the request to load the specified page of images. The page is shown when the response is received. |

Listener

Image Gallery Screenlet delegates some events to an object or class that implements its `ImageGalleryListener` interface. This interface extends the `BaseListListener` interface. Therefore, Image Gallery Screenlet's listener methods are as follows:

- `onListPageFailed(int startRow, Exception e)`: Called when the server call to retrieve a page of items fails. This method's arguments include the `Exception` generated when the server call fails.

- `onListPageReceived(int startRow, int endRow, List<Record> records, int rowCount)`: Called when the server call to retrieve a page of items succeeds. Note that this method may be called more than once; once for each page received. Because `startRow` and `endRow` change for each page, a `startRow` of `0` corresponds to the first item on the first page.

- `onListItemSelected(Record records, View view)`: Called when an item is selected in the list. This method's arguments include the selected list item (`Record`).
- `onImageEntryDeleted(long imageEntryId)`: Called when an item in the list is deleted.
- `onImageUploadStarted(String picturePath, String title, String description, String changelog)`: Called when an item is prepared for upload.
- `onImageUploadProgress(int totalBytes, int totalBytesSent)`: Called when an item is uploading.
- `onImageUploadEnd(ImageEntry entry)`: Called when an item finishes uploading.
- `showUploadImageView(String actionName, String picturePath, int screenletId)`: Called when the View for uploading an image is instantiated. The default behavior is to show the default View in a dialog. To retain this behavior, all this method needs to do is return `false`. To change the default behavior, use the `initializeUploadView` method to initialize a custom View that extends `BaseDetailUploadView`. Then return `true` to prevent the Screenlet from executing the default behavior. For example, the following sample implementation uses `initializeUploadView` to initialize the custom View instance `uploadDetailView`. It then performs a custom UI action (`uploadImageCard.goRight()`) and returns `true`:

  ```
  @Override
  public boolean showUploadImageView(String actionName, String picturePath, int screenletId) {
      uploadDetailView.initializeUploadView(actionName, picturePath, screenletId);
      uploadImageCard.goRight();

      return true;
  }
  ```

- `provideImageUploadDetailView()`: Called when the Screenlet provides the image upload View. To inflate the default View, return `0` in this method. Alternatively, display this View with a custom layout by returning its layout ID. Such a layout must have `DefaultUploadDetailView` as its root class.
- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.11 Rating Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Rating Screenlet shows an asset's rating. It also lets users update or delete the rating. This Screenlet comes with different Views that display ratings as thumbs, stars, and emojis.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensratingsentryService` (Screens compatibility plugin)	`getRatingsEntries`	With `entryId`
`ScreensratingsentryService` (Screens compatibility plugin)	`getRatingsEntries`	With `classPK` and `className`
`ScreensratingsentryService` (Screens compatibility plugin)	`updateRatingsEntry`	
`ScreensratingsentryService` (Screens compatibility plugin)	`deleteRatingsEntry`	

Module

- None

Views

The default View uses an Android `RatingBar` to show an asset's rating. Other custom Views may show the rating with a different Android component such as `Button`, `ImageButton`, or others.

This Screenlet has five different Views:

1. Like
2. Thumbs (default)
3. Stars
4. Reactions
5. Emojis

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when

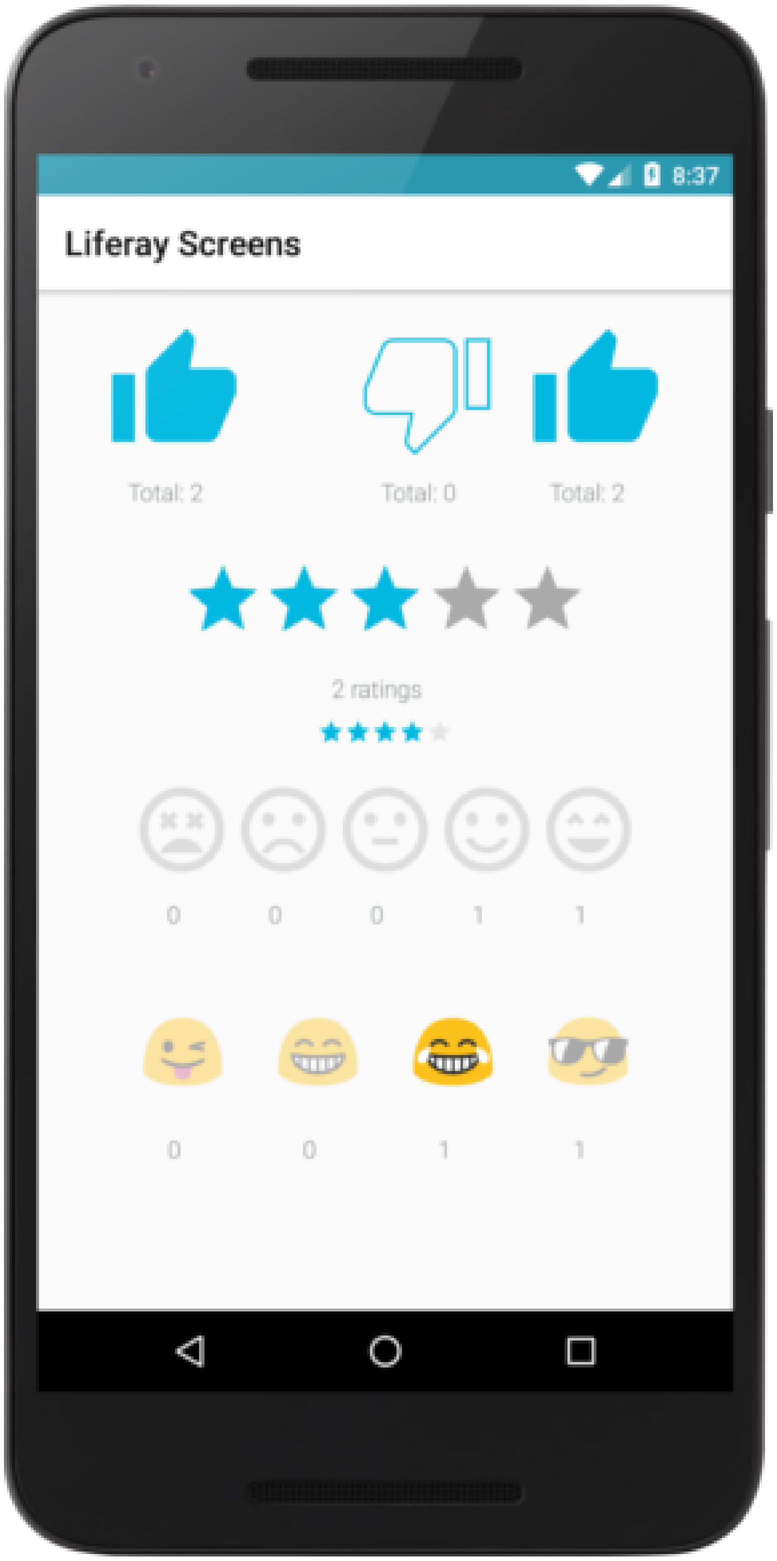


Figure 128.16: Rating Screenlet's different Views.

you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `entryId`

If you don't use `entryId`, you must use both of the following attributes:

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The ID of the layout to use to show the View. | `autoLoad` | `boolean` | Whether the rating loads automatically when the Screenlet appears in the app's UI. The default value is `true`. | `editable` | `boolean` | Whether the user can change the rating. | `entryId` | `number` | The primary key of the asset with the rating to display. | `className` | `string` | The asset's fully qualified class name. For example, a blog entry's `className` is `com.liferay.blogs.kernel.model.BlogsEntry`. The `className` attribute is required when using it with `classPK` to instantiate the Screenlet. | `classPK` | `number` | The asset's unique identifier. Only use this attribute when also using `className` to instantiate the Screenlet. | `groupId` | `number` | The ID of the site (group) containing the asset. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Methods

Method | Return | Explanation | `load()` | `void` | Starts the request to load the asset's ratings. |

Listener

Rating Screenlet delegates some events to an object or class that implements its `RatingListener` interface. Therefore, Rating Screenlet's listener methods are as follows:

- `onRatingOperationSuccess(AssetRating assetRating)`: Called when the operation finishes successfully and the rating is loaded.

128.12 Comment List Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Comment List Screenlet can list all the comments of an asset in a Liferay instance. It also lets the user update or delete comments.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreenscommentService` (Screens compatibility plugin)	`getComments`	
`ScreenscommentService` (Screens compatibility plugin)	`getCommentsCount`	

Module

- None

Views

- Default

The Default View uses an Android `RecyclerView` to show an asset's comments. Other Views may use a different component, such as `TableView` or others, to show the items.

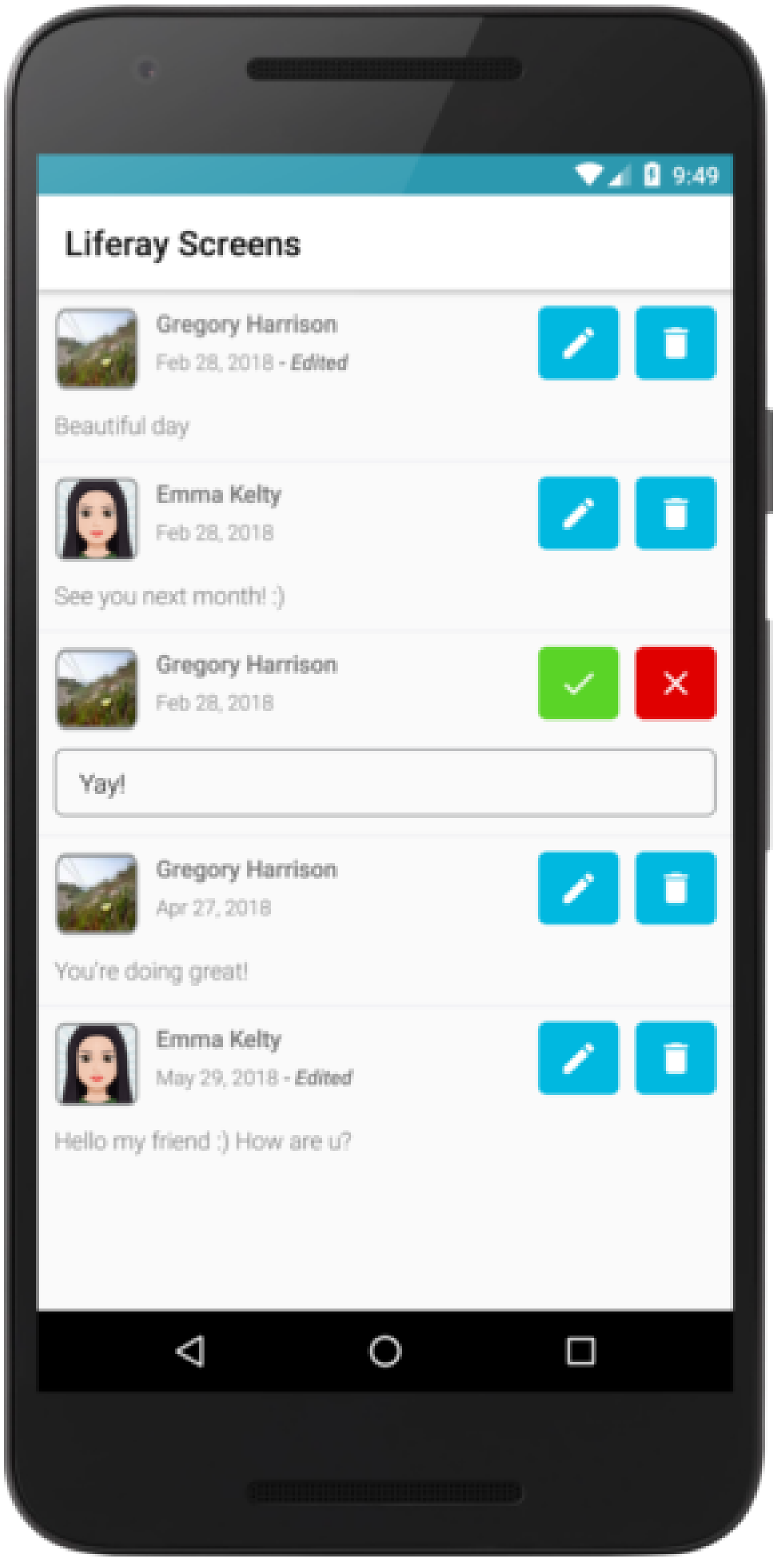


Figure 128.17: Comment List Screenlet using the Default View.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the comments from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the comments, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the comments from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the comments from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the comments from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View. | `autoLoad` | `boolean` | Whether the list should automatically load when the Screenlet appears in the app's UI. The default value is `true`. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. | `className` | `string` | The asset's fully qualified class name. For example, a blog entry's `className` is `com.liferay.blogs.kernel.model.BlogsEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The asset's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `firstPageSize` | `number` | The number of items to retrieve from the server for display on the first page. The default value is `50`. | `pageSize` | `number` | The number of items to retrieve from the server for display on the second and subsequent pages. The default value is `25`. | `labelFields` | `string` | The comma-separated names of the DDL fields to show. Refer to the list's data definition to find the field names. For more information on this, see the article on structured web content. Note that the appearance of data from a structure's fields depends on the `layoutId`. | `editable` | `boolean` | Whether the user can edit the comment. |

Methods

Method | Return | Explanation | `loadPage(pageNumber)` | `void` | Starts the request to load the specified page of records. The page is shown when the response is received. |

Listener

Comment List Screenlet delegates some events to a class that implements `CommentListListener`. This interface lets you implement the following methods:

- `onDeleteCommentSuccess(CommentEntry commentEntry)`: Called when the Screenlet successfully deletes the comment.
- `onUpdateCommentSuccess(CommentEntry commentEntry)`: Called when the Screenlet successfully updates the comment.
- `onListPageFailed(int startRow, Exception e)`: Called when the server call to retrieve a page of items fails. This method's arguments include the `Exception` generated when the server call fails.
- `onListPageReceived(int startRow, int endRow, List<CommentEntry> entries, int rowCount)`: Called when the server call to retrieve a page of items succeeds. Note that this method may be called more than once; once for each page received. Because `startRow` and `endRow` change for each page, a `startRow` of `0` corresponds to the first item on the first page.
- `onListItemSelected(CommentEntry element, View view)`: Called when an item is selected in the list. This method's arguments include the selected list item (`CommentEntry`).
- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.13 Comment Display Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Comment Display Screenlet can show one comment of an asset in a Liferay instance. It also lets the user update or delete the comment.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreenscommentService` (Screens compatibility plugin)	`getComment`	
`ScreenscommentService` (Screens compatibility plugin)	`updateComment`	
`CommentmanagerjsonwsService`	`deleteComment`	

Module

- None

Views

- Default

The Default View uses User Portrait Screenlet, and `TextView` and `ImageButton` elements to show an asset's comment. Other Views may different components to show the comment.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

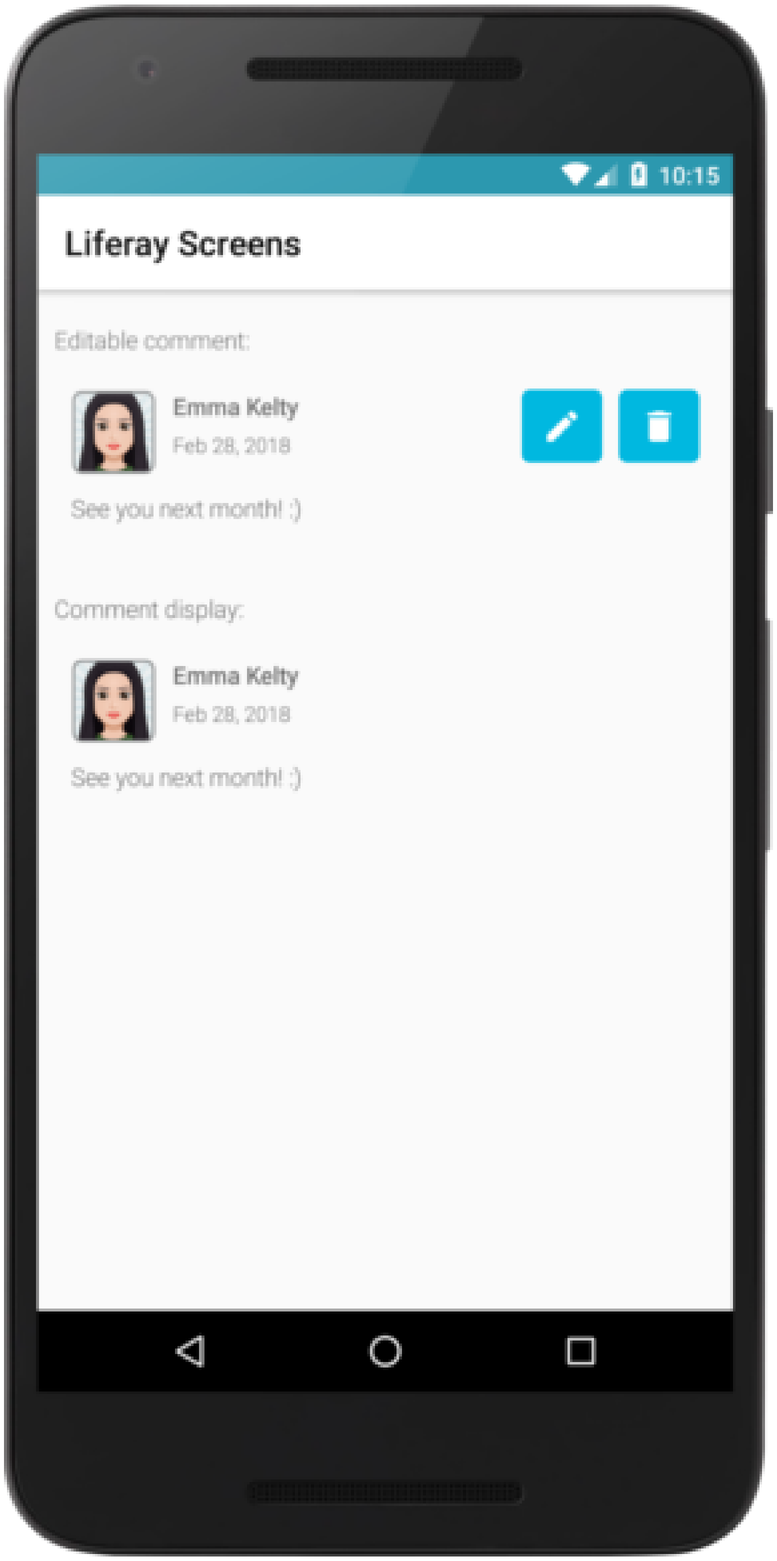


Figure 128.18: Comment Display Screenlet using the Default View.

Required Attributes

- `commentId`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View.| `autoLoad` | `boolean` | Whether the list should automatically load when the Screenlet appears in the app's UI. The default value is `true`. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. | `commentId` | `number` | The primary key of the comment to display. | `editable` | `boolean` | Whether the user can edit the comment. |

Methods

Method | Return | Explanation | `load()` | `void` | Starts the request to load the comment. |

Listener

Comment Display Screenlet delegates some events to a class that implements `CommentDisplayListener`. This interface lets you implement the following methods:

- `onLoadCommentSuccess(CommentEntry commentEntry)`: Called when the Screenlet successfully loads the comment.

- `onDeleteCommentSuccess(CommentEntry commentEntry)`: Called when the Screenlet successfully deletes the comment.

- `onUpdateCommentSuccess(CommentEntry commentEntry)`: Called when the Screenlet successfully updates the comment.

- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.14 Comment Add Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Comment Add Screenlet can add a comment to an asset in a Liferay instance.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreenscommentService` (Screens compatibility plugin)	`addComment`	

Module

- None

Views

- Default

The Default View uses Android's `EditText` and `Button` elements to show an add comment dialog. Other Views may use different components to show this dialog.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet sends the data to the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully sends the data, it also stores it in the local cache. | Use this policy when you always need to send updated data, and send nothing when there's no connection. | `CACHE_ONLY` | The Screenlet sends the data to the local cache. If an error occurs, the Screenlet uses the listener to notify the developer. | Use this policy when you always need to store local data without sending remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet sends the data to the Liferay instance. If this succeeds, the Screenlet also stores the data in the local cache. If a connection issue occurs, the Screenlet stores the data to the local cache and sends it to the Liferay instance when the connection is restored. If an error occurs, the Screenlet uses the listener to notify the developer. | Use this policy to send the most recent version of the data when connected, and store the data for later synchronization when there's no connection. | `CACHE_FIRST` | The Screenlet sends

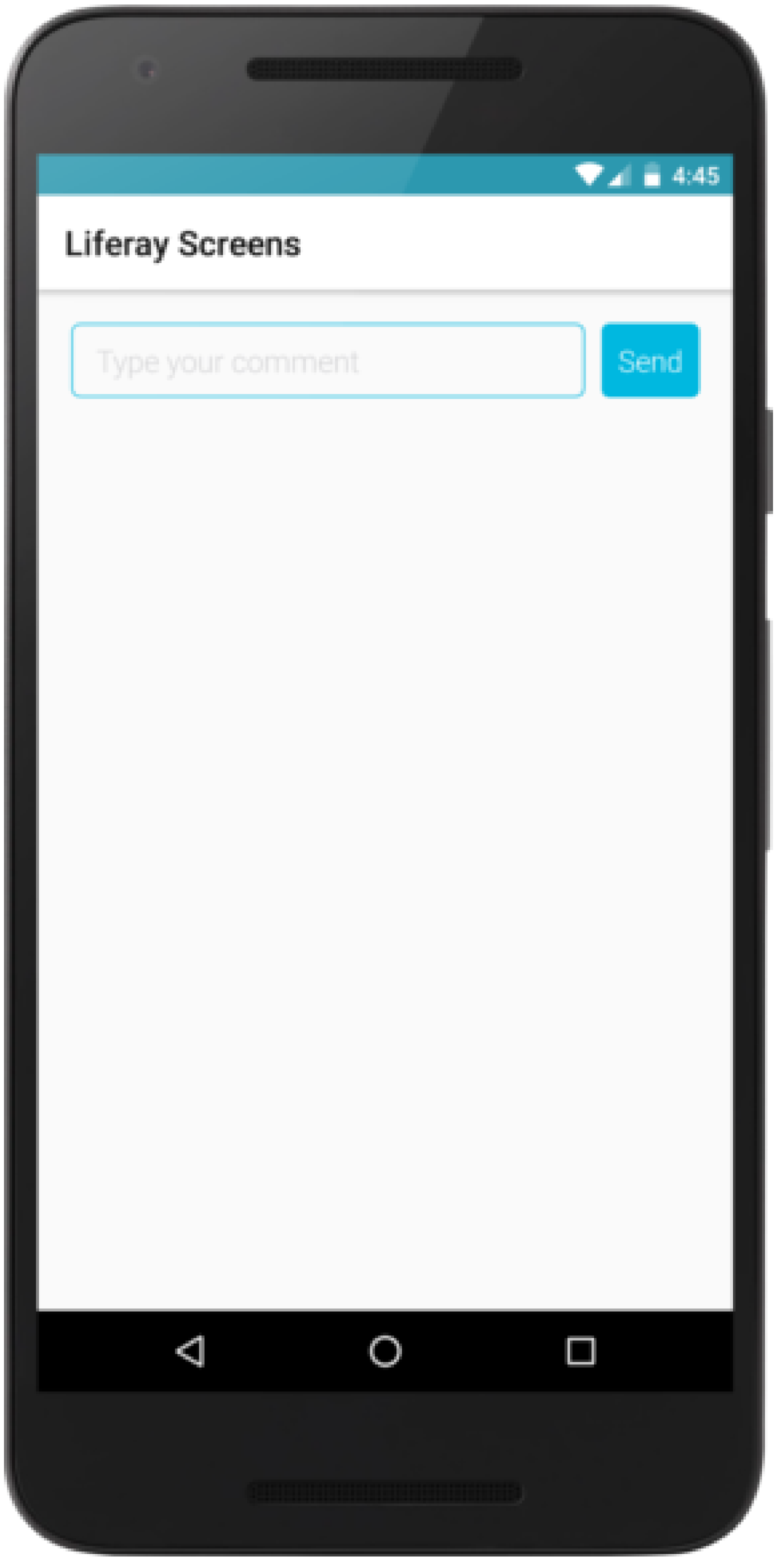


Figure 128.19: Comment Add Screenlet using the Default View.

the data to the local cache, then sends it to the Liferay instance. If sending the data to the Liferay instance fails, the Screenlet still stores the data locally and then notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and store local (but possibly outdated) data. |

Required Attributes

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View.| `className` | `string` | The asset's fully qualified class name. For example, a blog entry's `className` is `com.liferay.blogs.kernel.model.BlogsEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The asset's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Listener

Comment Add Screenlet delegates some events to a class that implements `CommentAddListener`. This interface lets you implement the following methods:

- `onAddCommentSuccess(CommentEntry commentEntry)`: Called when the Screenlet successfully adds a comment to the asset.

- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.15 Asset Display Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Asset Display Screenlet can display an asset from a Liferay instance. The Screenlet can currently display Documents and Media files (`DLFileEntry` images, videos, audio files, and PDFs), blogs entries (`BlogsEntry`) and web content articles (`WebContent`).

Asset Display Screenlet can also display your custom asset types. See the Listener section of this document for details.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Views

- Default

The Default View uses different UI elements to show each asset type. For example, it displays images with `ImageView` and blogs with `TextView`. Note that other Views may use different UI elements.

This Screenlet can also render other Screenlets as inner Screenlets:

- Images: Image Display Screenlet
- Videos: Video Display Screenlet
- Audio: Audio Display Screenlet
- PDFs: PDF Display Screenlet
- Blog entries: Blogs Entry Display Screenlet
- Web content: Web Content Display Screenlet

These Screenlets can also be used alone without Asset Display Screenlet.

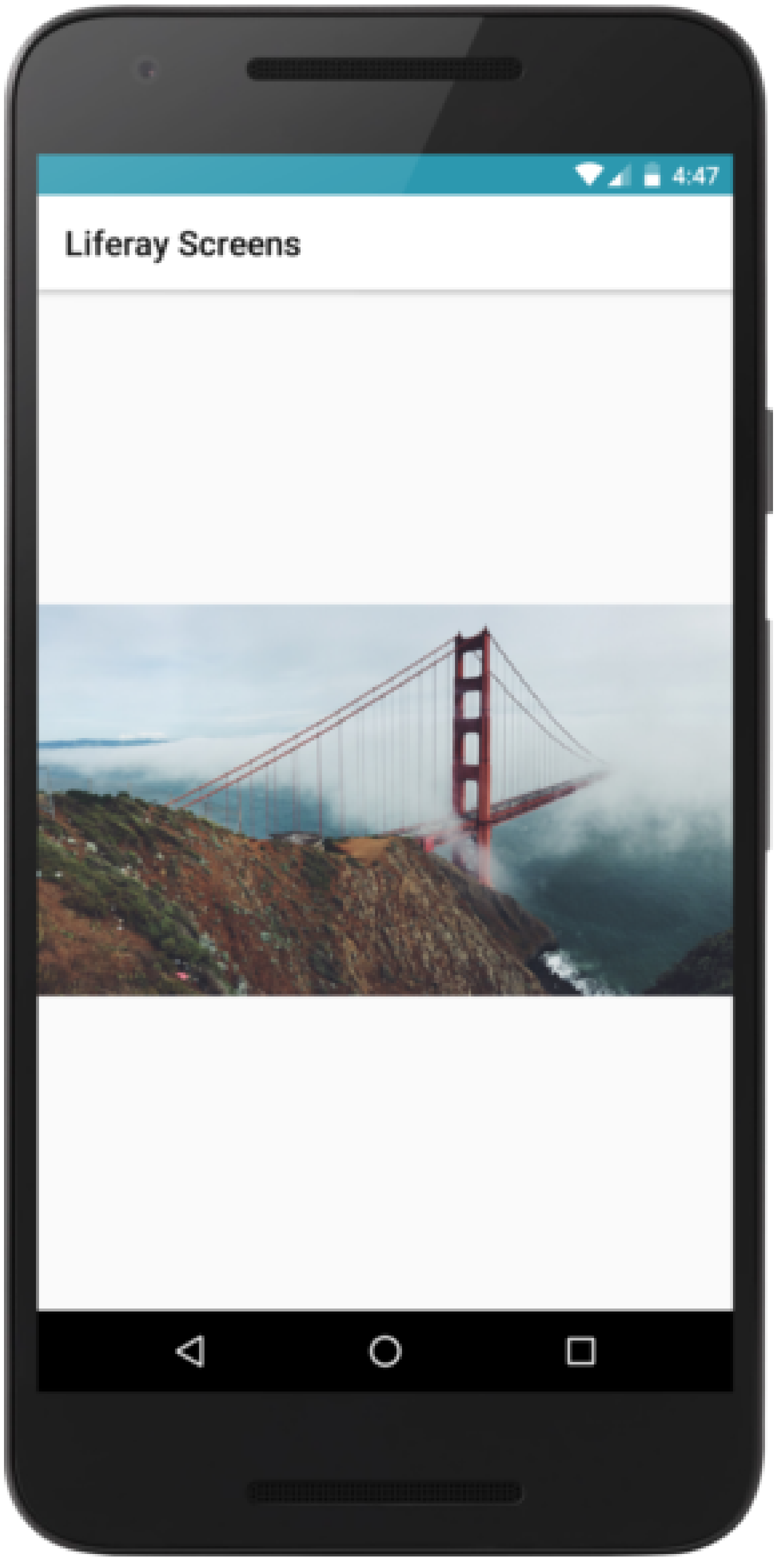


Figure 128.20: Asset Display Screenlet using the Default View.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `entryId`

Instead of `entryId`, you can use both of the following attributes:

- `className`
- `classPK`

If you don't use `entryId`, `className`, or `classPK`, you must use this attribute:

- `portletItemName`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View. | `autoLoad` | `boolean` | Whether the asset automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `entryId` | `number` | The primary key of the asset. | `className` | `string` | The asset's fully qualified class name. For example, a blog entry's `className` is `com.liferay.blogs.kernel.model.BlogsEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The asset's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `portletItemName` | `string` | The name of the configuration template you used in the Asset Publisher. To use this feature, add an Asset Publisher to one of your site's pages (it may be a hidden page), configure the Asset Publisher's filter (in *Configuration → Setup → Asset Selection*), and then use the Asset Publisher's *Configuration Templates* option to save this configuration with a name. Use this name in this attribute. If there is more than one asset in the configuration, the Screenlet displays only the first one. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. | `imageLayoutId` | `@layout` | The layout to use to show an image (`DLFileEntry`). | `videoLayoutId` | `@layout` | The layout to use to show a video (`DLFileEntry`). | `audioLayoutId` | `@layout` | The layout to use to show an audio file (`DLFileEntry`). | `pdfLayoutId` | `@layout` | The layout to use to show a PDF (`DLFileEntry`). | `blogsLayoutId` | `@layout` | The layout to use to show a blog entry (`BlogsEntry`). | `webDisplayLayoutId` | `@layout` | The layout to use to show a web content article (`WebContent`). |

Methods

Method | Return | Explanation | `load(AssetEntry assetEntry)` | `void` | Loads the given `AssetEntry` in the Screenlet. If no inner Screenlet is instantiated, the method tries to load the asset with a custom asset listener method. If this returns `null`, a new `Intent` is called to display the asset. | `loadAsset()` | `void` | Searches for the `AssetEntry` defined by the required attributes and loads it in the Screenlet. |

Listener

Asset Display Screenlet delegates some events to a class that implements `AssetDisplayListener`. This interface contains the following methods:

- `onRetrieveAssetSuccess(AssetEntry assetEntry)`: Called when the Screenlet successfully loads the asset.

A second listener, `AssetDisplayInnerScreenletListener`, also exists for configuring a child Screenlet (the Screenlet rendered inside Asset Display Screenlet) or rendering a custom asset.

- `onConfigureChildScreenlet(AssetDisplayScreenlet screenlet, BaseScreenlet innerScreenlet, AssetEntry assetEntry)`: Called when the child Screenlet has been successfully initialized. Use this method to configure or customize the child Screenlet. The example implementation here sets the child Screenlet's background color to light gray if the asset is a blog entry entity (`BlogsEntry`):

```
@Override
public void onConfigureChildScreenlet(AssetDisplayScreenlet screenlet,
    BaseScreenlet innerScreenlet, AssetEntry assetEntry) {
        if ("blogsEntry".equals(assetEntry.getObjectType())) {
            innerScreenlet.setBackgroundColor(ContextCompat.getColor(this,
            R.color.light_gray));
        }
}
```

- `onRenderCustomAsset(AssetEntry assetEntry)`: Called to render a custom asset. The following example implementation inflates and returns the custom View necessary to render a user from a Liferay instance (`User`):

```
@Override
public View onRenderCustomAsset(AssetEntry assetEntry) {
    if (assetEntry instanceof User) {
        View view = getLayoutInflater().inflate(R.layout.user_display, null);
        User user = (User) assetEntry;

        TextView username = (TextView) view.findViewById(R.id.liferay_username);

        username(user.getUsername());
```

```
            return view;
        }
        return null;
    }
```

128.16 Blogs Entry Display Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Blogs Entry Display Screenlet displays a single blog entry. Image Display Screenlet renders any header image the blogs entry may have.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Views

- Default

The Default View uses different components to show a blogs entry (`BlogsEntry`). For example, it uses an Android `TextView` to show the blog's text, and User Portrait Screenlet to show the profile picture of the Liferay user who posted it. Note that other custom Views may use different components.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote data under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

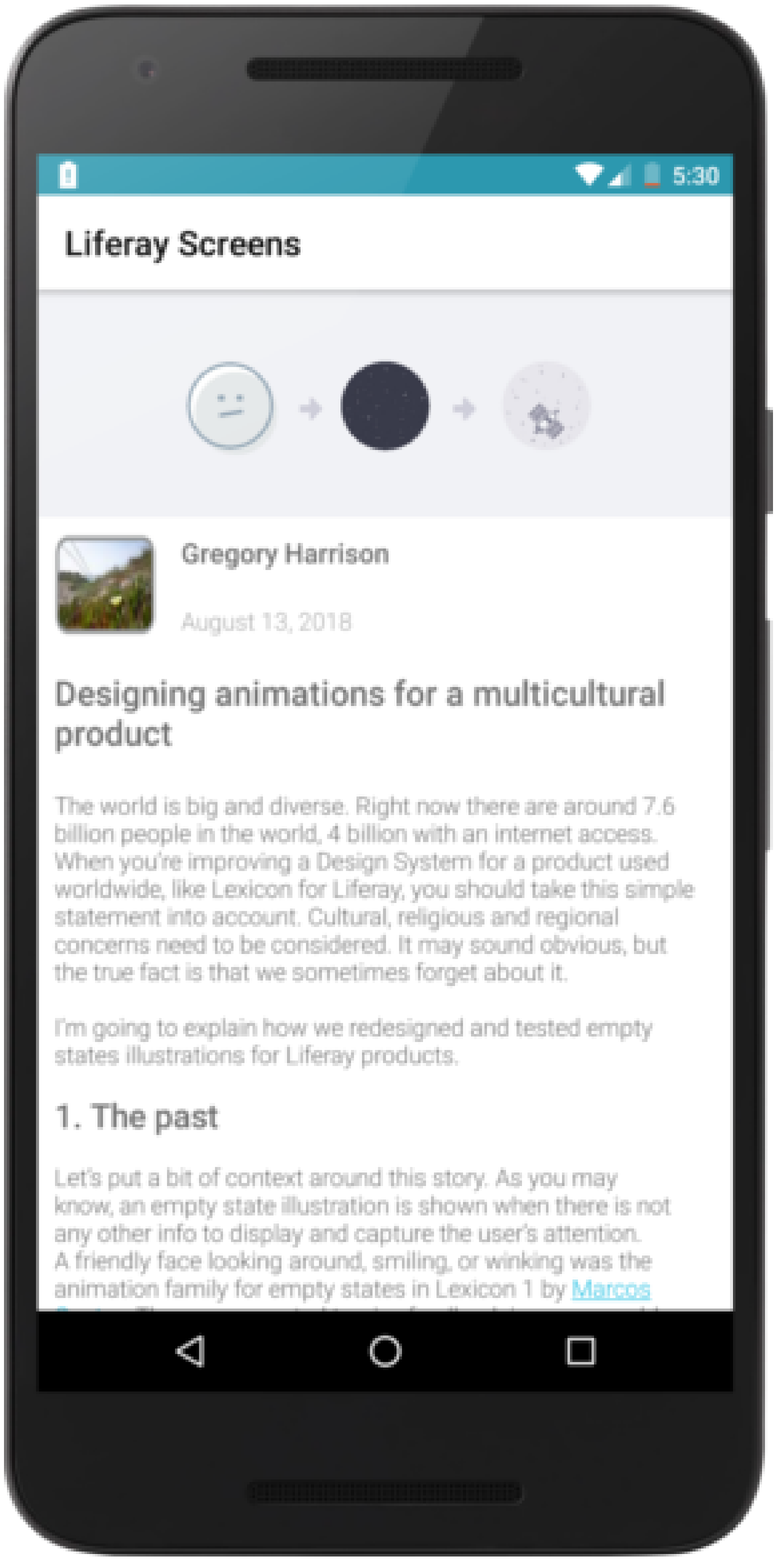
- `entryId`

If you don't use `entryId`, you must use both of the following attributes:

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View.| `autoLoad` | `boolean` | Whether the blog entry automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `entryId` | `number` | The primary key of the blog entry (`BlogsEntry`). | `className` | `string` | The `BlogsEntry` object's fully qualified class name. This is `com.liferay.blogs.kernel.model.BlogsEntry`. If you don't use `entryId`, the `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The `BlogsEntry` object's unique identifier. If you don't use `entryId`, the `className` and `classPK`



Gregory Harrison

August 13, 2018

Designing animations for a multicultural product

The world is big and diverse. Right now there are around 7.6 billion people in the world, 4 billion with an internet access. When you're improving a Design System for a product used worldwide, like Lexicon for Liferay, you should take this simple statement into account. Cultural, religious and regional concerns need to be considered. It may sound obvious, but the true fact is that we sometimes forget about it.

I'm going to explain how we redesigned and tested empty states illustrations for Liferay products.

1. The past

Let's put a bit of context around this story. As you may know, an empty state illustration is shown when there is not any other info to display and capture the user's attention. A friendly face looking around, smiling, or winking was the animation family for empty states in Lexicon 1 by Marcos

Figure 128.21: Blogs Entry Display Screenlet using the Default View.

attributes are required to instantiate the Screenlet. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Listener

Because a blog entry is an asset, Blogs Entry Display Screenlet delegates its events to a class that implements `AssetDisplayListener`. This interface lets you implement the following method:

- `onRetrieveAssetSuccess(AssetEntry assetEntry)`: Called when the Screenlet successfully loads the blog entry.
- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.17 Image Display Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Image Display Screenlet displays an image file from a Liferay instance's Documents and Media Library.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Views

- Default

The Default View uses an Android `ImageView` to display the image.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `entryId` or `classPK`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View. | `autoLoad` | `boolean` | Whether the image automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `entryId` | `number` | The primary key of the image. | `classPK` | `number` | The image's unique identifier. |

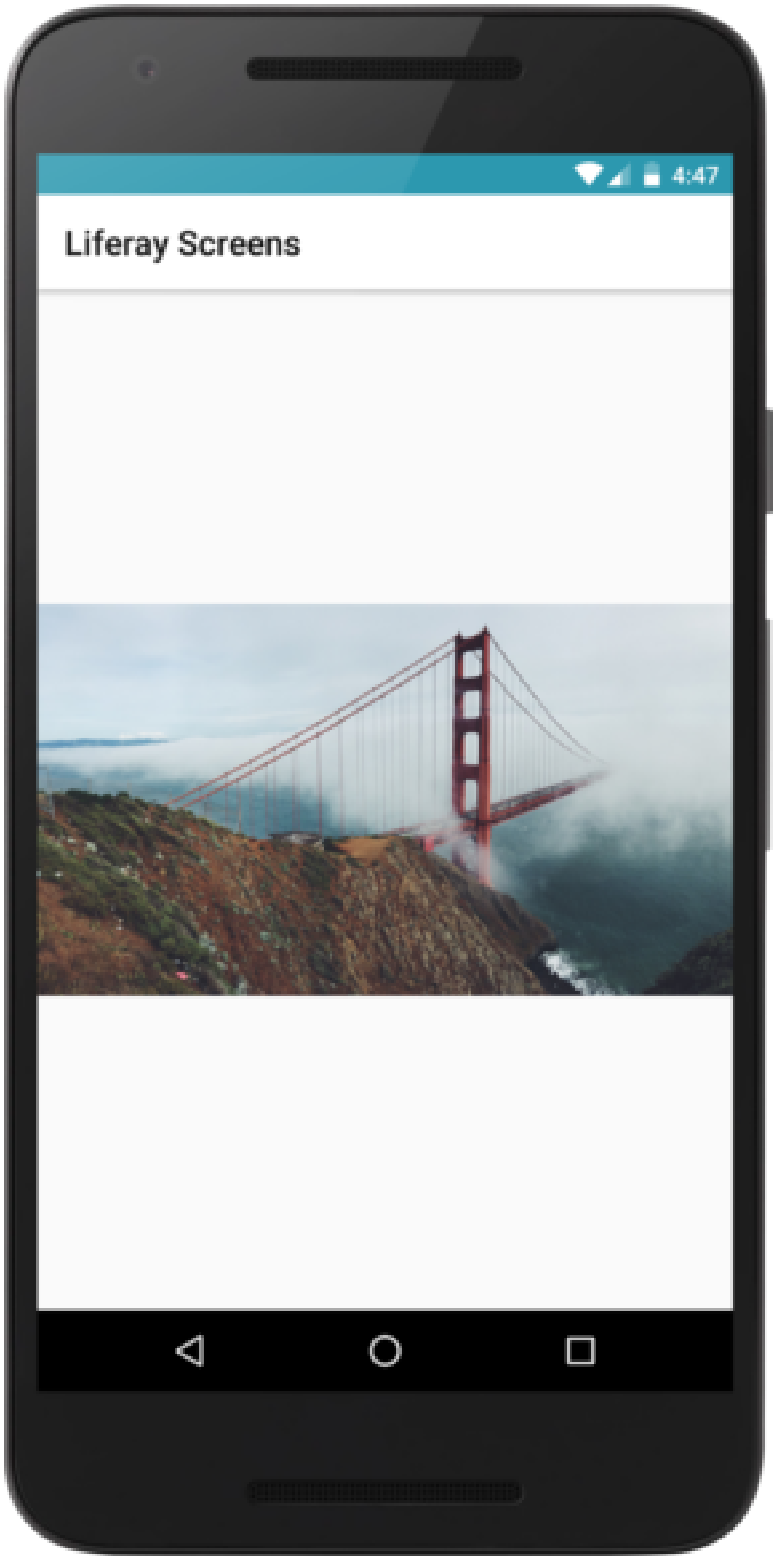


Figure 128.22: Image Display Screenlet using the Default View.

`cachePolicy` | `string` | The offline mode setting. See the Offline section for details. | `imageScaleType` | `number` | Lets you set a scale image type like `CENTER`, `CENTER_CROP`, `CENTER_INSIDE`, `FIT_CENTER`, `FIT_END`, `FIT_START`, `FIT_XY`, `MATRIX`. | `placeHolder` | `@resource` | Image to load until the final image loads. | `placeHolderScaleType` | `number` | Lets you set a scale image type for the placeholder like `CENTER`, `CENTER_CROP`, `CENTER_INSIDE`, `FIT_CENTER`, `FIT_END`, `FIT_START`, `FIT_XY`, `MATRIX`. |

Note that the values for `imageScaleType` and `placeHolderScaleType` match those described in Android's `ImageView.ScaleType`.

Listener

Because images are assets, Image Display Screenlet delegates its events to a class that implements `AssetDisplayListener`. This interface lets you implement the following methods:

- `onRetrieveAssetSuccess(AssetEntry assetEntry)`: Called when the Screenlet successfully loads the image.
- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.18 Video Display Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Video Display Screenlet displays a video file from a Liferay instance's Documents and Media Library.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Views

- Default

The Default View uses an Android `VideoView` to display the video.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `entryId` or `classPK`

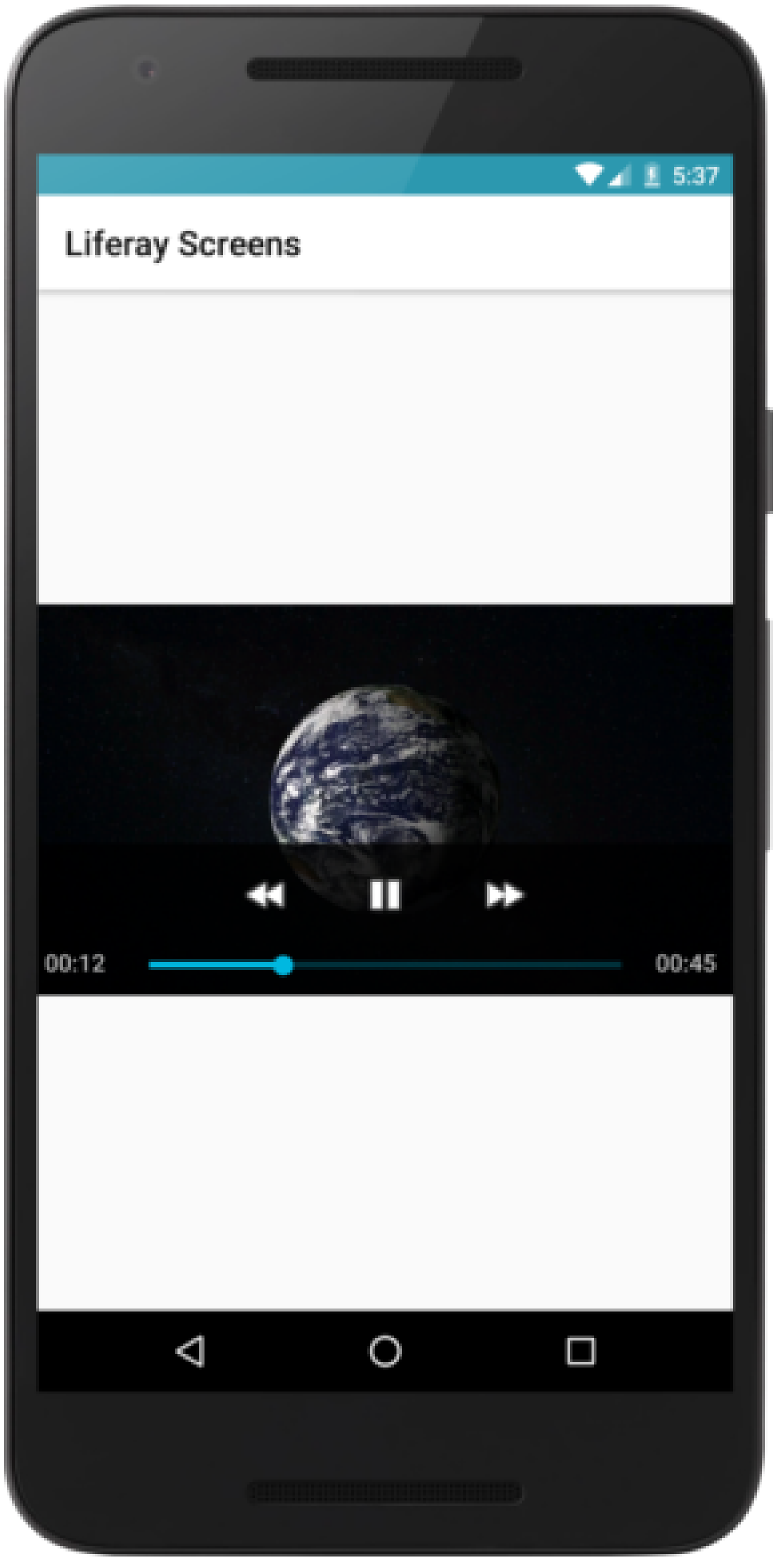


Figure 128.23: Video Display Screenlet using the Default View.

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View. | `autoLoad` | `boolean` | Whether the video automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `entryId` | `number` | The primary key of the video file. | `classPK` | `number` | The video file's unique identifier. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Listener

Video Display Screenlet delegates its events to a class that implements `VideoDisplayListener`. This interface lets you implement these methods:

- `onVideoPrepared()`: Called when the video is ready for display.
- `onVideoCompleted()`: Called when the video is completed.
- `onVideoError(Exception e)`: Called when an error occurs displaying the video.
- `onRetrieveAssetSuccess(AssetEntry assetEntry)`: Called when the Screenlet successfully loads the video.
- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.19 Audio Display Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Audio Display Screenlet displays an audio file from a Liferay instance's Documents and Media Library.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Views

- Default

The Default View uses an Android `VideoView` to display the audio file.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

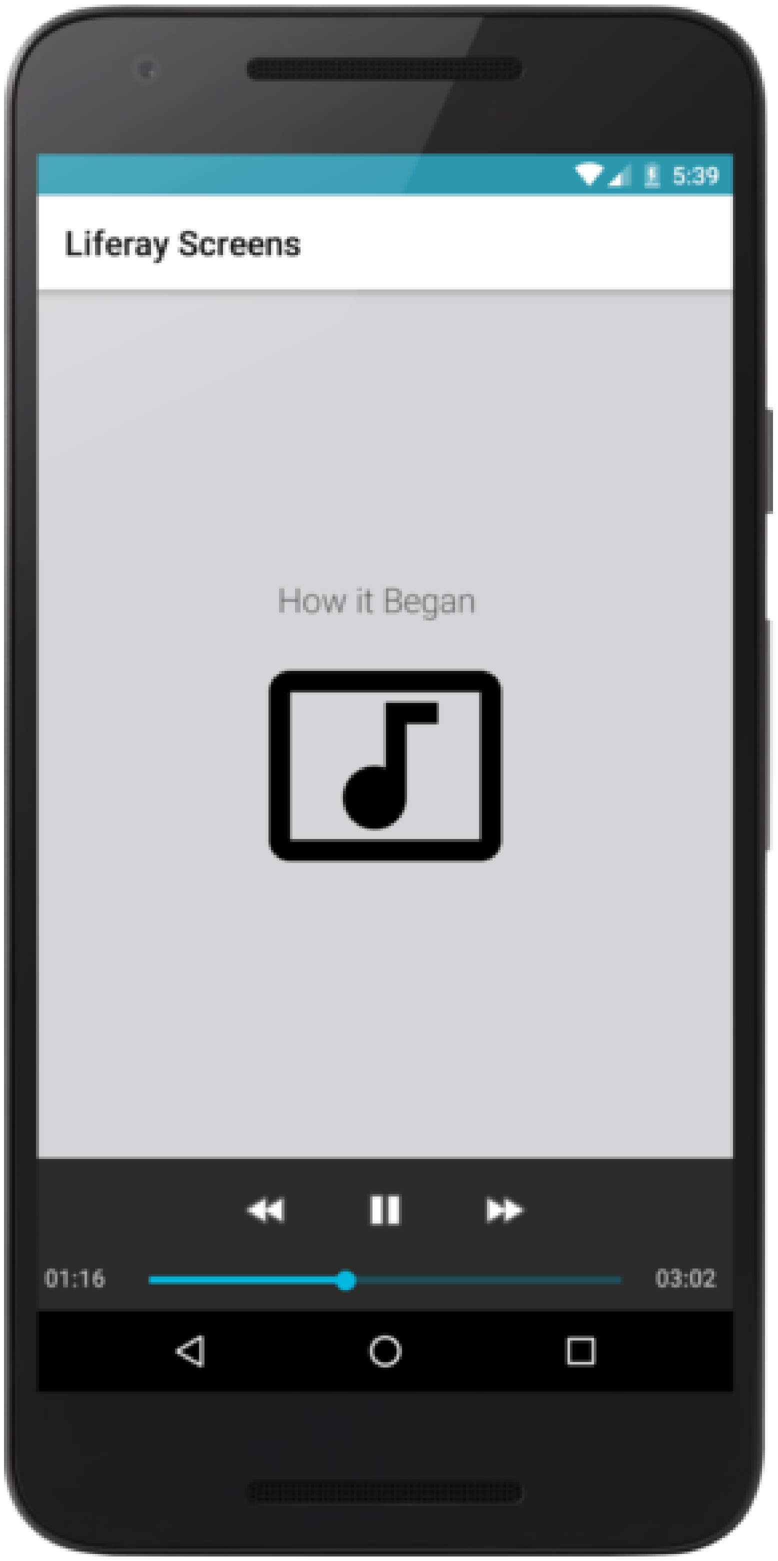


Figure 128.24: Audio Display Screenlet using the Default View.

Required Attributes

- `entryId` or `classPK`

Attributes

Attribute	Data type	Explanation
`layoutId`	`@layout`	The layout to use to show the View.
`autoLoad`	`boolean`	Whether the audio file automatically loads when the Screenlet appears in the app's UI. The default value is `true`.
`entryId`	`number`	The primary key of the audio file.
`classPK`	`number`	The audio file's unique identifier.
`cachePolicy`	`string`	The offline mode setting. See the Offline section for details.

Listener

Because audio files are assets, Audio Display Screenlet delegates its events to a class that implements `AssetDisplayListener`. This interface lets you implement the following methods:

- `onRetrieveAssetSuccess(AssetEntry assetEntry)`: Called when the Screenlet successfully loads the audio file.

- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.20 PDF Display Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

PDF Display Screenlet displays a PDF file from a Liferay Instance's Documents and Media Library.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Views

- Default

The Default View uses Android's `PdfRenderer` to display the PDF. Note that `PdfRenderer` requires an Android API level of 21 or higher.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `REMOTE_ONLY` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `CACHE_ONLY` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `REMOTE_FIRST` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `CACHE_FIRST` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity

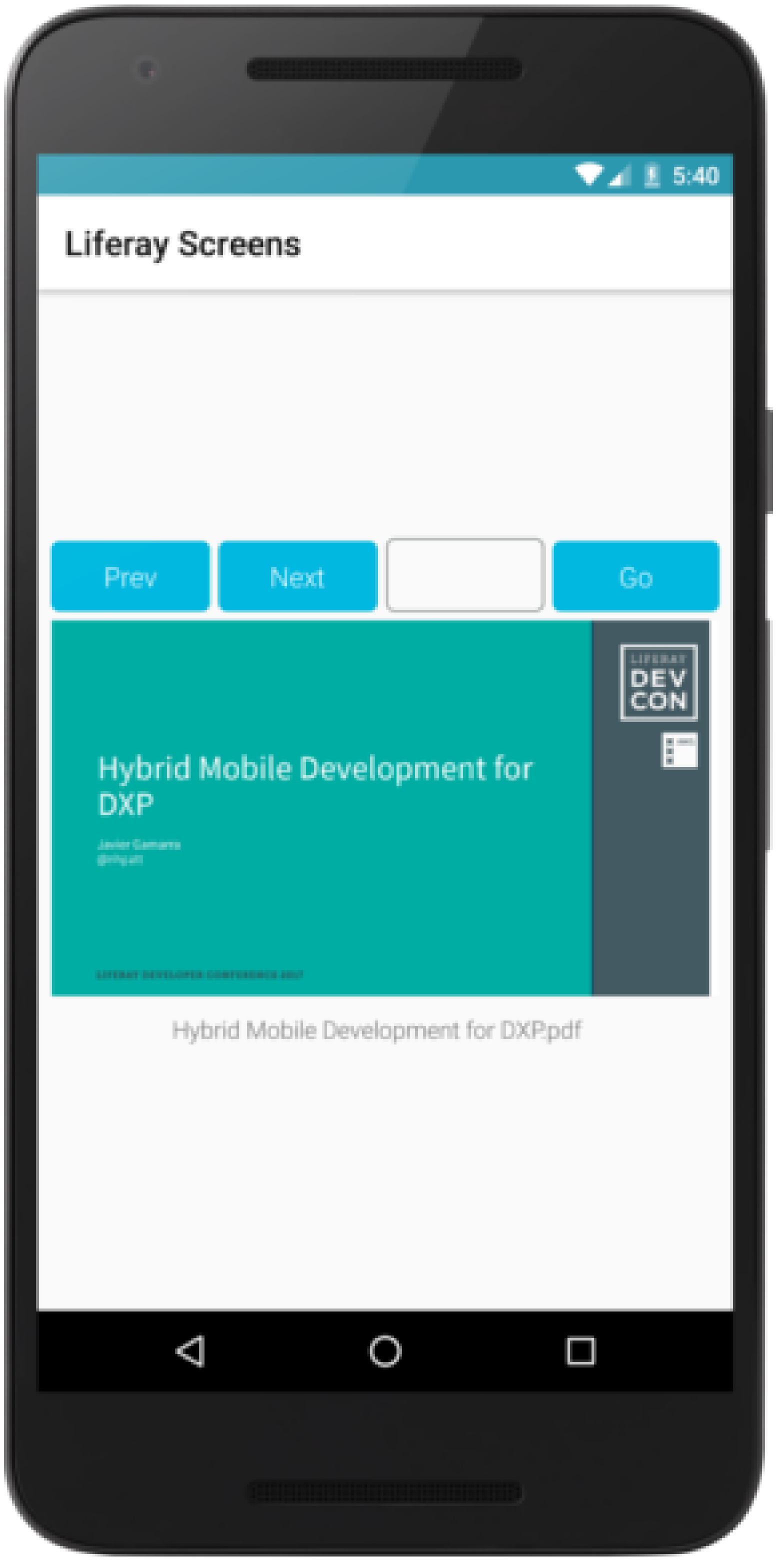

5:40
Liferay Screens
Prev
Next
Go
LIFERAY DEV CON
Hybrid Mobile Development for DXP
Hybrid Mobile Development for DXP.pdf

Figure 128.25: PDF Display Screenlet using the Default View.

errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `entryId` or `classPK`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The layout to use to show the View. | `autoLoad` | `boolean` | Whether the PDF automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `entryId` | `number` | The primary key of the PDF file. | `classPK` | `number` | The PDF file's unique identifier. | `cachePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Listener

Because PDF files are assets, PDF Display Screenlet delegates its events to a class that implements `AssetDisplayListener`. This interface lets you implement the following methods:

- `onRetrieveAssetSuccess(AssetEntry assetEntry)`: Called when the Screenlet successfully loads the PDF file.

- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.21 Web Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 16) or above
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- Android SDK 4.1 (API Level 16) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Web Screenlet lets you display any web page. It also lets you customize the web page through injection of local and remote JavaScript and CSS files. If you're using Liferay DXP as backend, you can use Application Display Templates in your page to customize its content from the server side.

Module

- None

Views

- Default

Configuration

To learn how to use Web Screenlet, follow the steps in the tutorial Rendering Web Pages in Your Android App. That tutorial gives detailed instructions for using the configuration items described here.

Web Screenlet has `WebScreenletConfiguration` and `WebScreenletConfiguration.Builder` classes that you can use together to supply the parameters that the Screenlet needs to work. `WebScreenletConfiguration.Builder` has the following methods, which let you supply the described configuration parameters:

Method	Return	Explanation
`addLocalJs(fileName)`	`WebScreenletConfiguration.Builder`	Adds a local JavaScript file with the supplied filename. The JavaScript files must be in the first level of your app's `assets` folder. Create this folder at the same level of the `res` folder.
`addLocalCss(fileName)`	`WebScreenletConfiguration.Builder`	Adds a local CSS file with the supplied filename. The CSS files must be in the first level of your app's `assets` folder. Create this folder at the same level of the `res` folder.
`addRawJs(rawJs, name)`	`WebScreenletConfiguration.Builder`	Adds a JavaScript file from your app's `res/raw` folder. Create this folder if it doesn't exist. Reference the file using `R.raw.yourfilename`. This method also takes a second parameter called `name`, which is only for debugging purposes. If there's an error, the console displays it with this `name` value.
`addRawCss(rawCss, name)`	`WebScreenletConfiguration.Builder`	Adds a CSS file from your app's `res/raw` folder. Create this folder if it doesn't exist. Reference the file using `R.raw.yourfilename`. This method also takes a second parameter called `name`, which is only for debugging purposes. If there's an error, the console displays it with this `name` value.
`addRemoteJs(url)`	`WebScreenletConfiguration.Builder`	Adds a JavaScript file from the supplied URL.
`addRemoteCss(url)`	`WebScreenletConfiguration.Builder`	Adds a CSS file from the supplied URL.
`setWebType(webType)`	`WebScreenletConfiguration.Builder`	Sets the `WebType`.
`enableCordova(observer)`	`WebScreenletConfiguration.Builder`	Enables Cordova inside the Web Screenlet.
`load()`	`WebScreenletConfiguration`	Returns the `WebScreenletConfiguration` object that you can set to the Screenlet instance.

Note: If you want to add comments in the scripts, use the `/**/` notation.

WebType

- **WebType.LIFERAY_AUTHENTICATED** (default): Displays a Liferay DXP page that requires authentication. The user must therefore be logged in with Screens via Login Screenlet or a `SessionContext` method. For this `WebType`, the URL you must pass to the `WebScreenletConfiguration.Builder` constructor is a relative URL. For example, if the full URL is `http://screens.liferay.org.es/web/guest/blog`, then the URL you must supply to the constructor is `/web/guest/blog`.

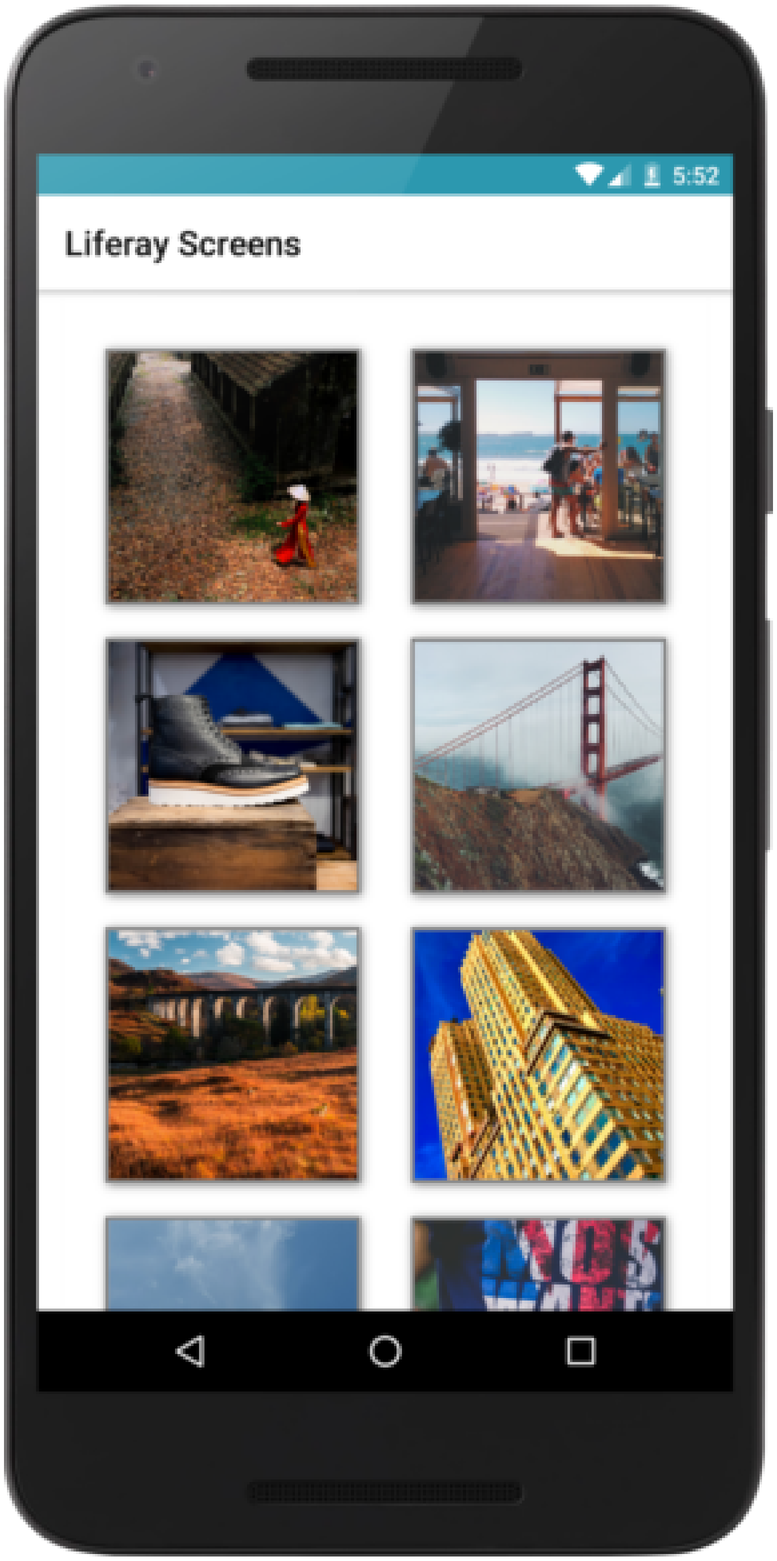


Figure 128.26: The Web Screenlet with the Default View Set.

- **WebType.OTHER**: Displays any other page. For this `WebType`, the URL you must pass to the `WebScreenletConfiguration.Builder` constructor is a full URL. For example, if the full URL is `http://screens.liferay.org.es/web/guest/blog`, then you must supply that URL to the constructor.

Attributes

Attribute | Data type | Explanation | `autoLoad` | `boolean` | Whether to load the page automatically when the Screenlet appears in the app's UI. The default value is true. | `layoutId` | `@layout` | The layout to use to show the View. | `isLoggingEnabled` | `boolean` | Whether logging is enabled. | `isScrollEnabled` | `boolean` | Whether to enable scrolling on the page inside the Screenlet. |

Methods

Method | Return | Explanation | `load()` | `void` | Checks if the page's URL is valid, and then loads it. The operation fails if the URL is invalid. | `clearCache()` | `void` | Clears the Web Screenlet's cache. | `injectScript(script)` | `void` | Injects a script when the page is already loaded. |

Listener

Web Screenlet delegates some events to an object or class that implements its `WebListener` interface. This interface extends the `BaseCacheListener` interface. Therefore, Web Screenlet's listener methods are as follows:

- `onPageLoaded(String url)`: Called when the Screenlet loads the page correctly.
- `onScriptMessageHandler(String namespace, String body)`: Called when the `WebView` in the Screenlet sends a message. The `namespace` parameter is the source namespace key, and `body` is the source namespace body.
- `error(Exception e, String userAction)`: Called when an error occurs in the process. The `userAction` argument distinguishes the specific action in which the error occurred.

128.22 Android Breaking Changes

This document presents a list of changes in Liferay Screens for Android 2.0 that break preceding functionality. We try our best to minimize these disruptions, but sometimes they are unavoidable.

Breaking Changes List

Interactors Now Run in a Background Process

What changed? Interactors now run in a background process, so you don't need to create or set callback classes manually. This means you can write what appear to be synchronous server calls, and Liferay Screens handles the background threading for you. The Interactor's `execute` method makes the server call. Invoking the `start` method in your Screenlet class causes `execute` to run in a background thread.

Note that you no longer have to handle the exception when making the server call. The Screenlet framework does this for you and propagates any error via the listeners. Also note that the `screenletId` is no longer required. The Screenlet framework automatically decorates the event with a `screenletId` that it generates.

Who is affected? This affects all Screenlet Interactors.

How should I update my code? You must rewrite your Interactors. See the tutorial Creating Android Screenlets for the most recent instructions on creating an Interactor.

Why was this change made? Asynchronous calls can be difficult to develop and work with. By handling them for you, Liferay Screens removes this potential source of error and frees you to focus on other parts of your Screenlet.

Changes to View Set Inheritance

What changed? To use a View Set, your app or activity's theme must also inherit that View Set's styles. For example, to use the Default View Set, your app or activity's theme must inherit `default_theme`.

Who is affected? This affects any apps or activities that use a View Set without inheriting that View Set's styles. For example, if you use the Default View for a Screenlet by setting the Screenlet XML's `layoutId` attribute, your app or activity's theme must now inherit `default_theme` as well. Likewise, your app or activity's theme must inherit `westeros_theme` or `material_theme` to use the Westeros or Material View Set, respectively.

How should I update my code? Change your app or activity's theme to inherit the styles of the View Set you want to use.

Example

This code snippet from an app's `res/values/styles.xml` tells `AppTheme.NoActionBar` to inherit the Default View Set's styles:

```
<resources>

    <style name="AppTheme.NoActionBar" parent="default_theme">
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>

        <item name="windowActionBar">false</item>
        <item name="windowNoTitle">true</item>
    </style>
    ...
</resources>
```

Why was this change made? This lets you change an Android theme's colors and styles according to Android conventions. Before, the Android themes were hardcoded inside the Screenlets.

The Screenlet Attribute offlinePolicy is now cachePolicy

What changed? The Screenlet attribute `offlinePolicy` is now `cachePolicy`.

Who is affected? This affects any Screenlets that used the `offlinePolicy` attribute to set that Screenlet's offline mode policy.

How should I update my code? In the app layouts that contain the Screenlet, change the `offlinePolicy` attribute to `cachePolicy`.

Example

Old way:

```
<com.liferay.mobile.screens.assetlist.AssetListScreenlet
    android:id="@+id/asset_list_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    liferay:autoLoad="false"
    liferay:offlinePolicy="REMOTE_FIRST" />
```

New way:

```
<com.liferay.mobile.screens.asset.list.AssetListScreenlet
    android:id="@+id/asset_list_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    liferay:autoLoad="false"
    liferay:cachePolicy="REMOTE_FIRST"
    />
```

Why was this change made? This change was made for consistency throughout Liferay Screens. The method and class names in the offline mode APIs contain *cache*, as do the offline policies `CACHE_ONLY` and `CACHE_FIRST`.

Some Listener Methods in DDL Form Screenlet Have Changed

What changed? The following error listener methods in DDL Form Screenlet's `DDLFormListener` have been removed:

- `void onDDLFormLoadFailed(Exception e)`: Called when an error occurs in the load form definition request.
- `void onDDLFormRecordLoadFailed(Exception e)`: Called when an error occurs in the load form record request.
- `void onDDLFormRecordAddFailed(Exception e)`: Called when an error occurs in the request to add a new record.
- `void onDDLFormUpdateRecordFailed(Exception e)`: Called when an error occurs in the request to update an existing record.

Also in `DDLFormListener`, the method `onDDLFormRecordLoaded` now takes an additional parameter for the attribute map received from the server.

Who is affected? This affects any classes that implement `DDLFormListener`.

How should I update my code? In place of the removed error listeners, use `BaseCacheListener`'s generic error listener:

```
void error(Exception e, String userAction)
```

You must also change any `onDDLFormRecordLoaded` implementations to account for the method's new signature:

```
public void onDDLFormRecordLoaded(Record record, Map<String, Object> valuesAndAttributes)
```

Why was this change made? The old error listener methods were usually implemented the same way: by logging the exception. Multiple error listener methods aren't needed for this. You can use the new error listener method to log the exception and take any other action that depends on the user action.

Cache Listener Methods Moved into Their Own Listener

What changed? The cache listener methods `loadingFromCache`, `retrievingOnline`, and `storingToCache` have been moved to their own listener, `CacheListener`. Note, this change was introduced in Liferay Screens 1.4.0.

Who is affected? All activity classes that implement a listener.

How should I update my code? If you don't have special behavior in your old cache listener method implementations, you can remove them. Otherwise, you must implement the new `CacheListener`. When implementing `CacheListener` (in an activity or fragment, for example), you should also register a Screenlet instance as the cache listener:

```
    screenlet.setCacheListener(this);
```

Example
In the Liferay Screens test app, the activity `UserPortraitActivity` implements `CacheListener`:

```
public class UserPortraitActivity extends ThemeActivity implements UserPortraitListener,
    CacheListener {

    private UserPortraitScreenlet screenlet;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);

            setContentView(R.layout.user_portrait);

            screenlet = (UserPortraitScreenlet) findViewById(R.id.user_portrait_screenlet);
            screenlet.setListener(this);
            screenlet.setCacheListener(this);
    }

    …

    @Override
    public void loadingFromCache(boolean success) {
        View content = findViewById(android.R.id.content);
        Snackbar.make(content, "Trying to load from cache: " + success,
            Snackbar.LENGTH_SHORT).show();
    }

    @Override
    public void retrievingOnline(boolean triedInCache, Exception e) {

    }

    @Override
    public void storingToCache(Object object) {
        View content = findViewById(android.R.id.content);
        Snackbar.make(content, "Storing to cache...", Snackbar.LENGTH_SHORT).show();
    }

    …

}
```

Why was this change made? Reacting to cache errors via the cache listener methods is a niche use case. Because the old cache listener methods were part of the normal listener, developers were forced to implement them whether they needed them or not. Putting them in their own listener makes their implementation optional.

Changed BaseListListener Methods

What changed? The `BaseListListener` methods `onListPageFailed` and `onListPageReceived` no longer have the `BaseListScreenlet` argument `source`. These methods also now account for a page's start and end row instead of the page number.

Who is affected? This affects any classes or interfaces that extend or implement `BaseListListener`.

How should I update my code? Remove the `BaseListScreenlet` argument from your `onListPageFailed` and `onListPageReceived` implementations. You must also replace the `int page` argument in `onListPageFailed` with an `int` argument representing the page's start row. Likewise, replace the `int page` argument in `onListPageReceived` with two `int` arguments that represent the page's start row and end row, respectively.

Example

Old signatures:

```
void onListPageFailed(BaseListScreenlet source, int page, Exception e)
void onListPageReceived(BaseListScreenlet source, int page, List<E> entries, int rowCount)
```

New signatures:

```
void onListPageFailed(int startRow, Exception e)
void onListPageReceived(int startRow, int endRow, List<E> entries, int rowCount)
```

Why was this change made? The `BaseListScreenlet` argument served to disambiguate two instances of the same Screenlet in a single activity. This is a very rare use case. Therefore, forcing the argument on all `BaseListListener` implementations was unnecessary. If you still need this use case, create a Screenlet instance and listener for each Screenlet instead of relying on the `BaseListScreenlet` argument in a single listener. The start row and end row change was made for consistency with other listeners that also use start row and end row arguments.

Changed Asset List Screenlet Package

What changed? Asset List Screenlet's package is now `com.liferay.mobile.screens.asset.list` instead of `com.liferay.mobile.screens.assetlist`.

Who is affected? This affects any activities or fragments that use Asset List Screenlet.

How should I update my code? Change your `com.liferay.mobile.screens.assetlist` imports to `com.liferay.mobile.screens.asset.list`.

Why was this change made? This allows for other Screenlets that work with assets, like Asset Display Screenlet. For example, the package `com.liferay.mobile.screens.asset` now contains Asset List Screenlet, Asset Display Screenlet, and classes common to both.

Changed Return Type for a DDL Record Method

What changed? The `getModelValues()` method for DDL records now returns a `Map` instead of a `HashMap`.

Who is affected? This affects any code that expects `getModelValues()` to return a `HashMap`.

How should I update my code? Change any code that uses `getModelValues()` to expect a `Map` instead of a `HashMap`.

Why was this change made? This follows general Java conventions.

Changed Code Conventions for Private and Protected Fields

What changed? Private and protected fields in Screenlets are no longer prefixed by `_`.

Who is affected? This affects any code that directly accesses protected fields.

How should I update my code? Change your code to use the new variable name. For example, if your code directly accesses a protected Screenlet variable named `_fields`, change it to use `fields` instead.

Why was this change made? This follows general Java naming conventions.

Changes to Using a Screenlet without a View

What changed? If you're using a Screenlet without View (like you might be if you need to log a user in programmatically), you no longer have to call `LiferayScreensContext.init(this)` to initialise the library. This is now called automatically.

Who is affected? This affects any apps that use a Screenlet without a View.

How should I update my code? Remove your manual call to `LiferayScreensContext.init(this)`.

Why was this change made? This removes the possibility of an error if you forget to call `LiferayScreensContext.init(this)` when using a Screenlet without a View.

Chapter 129

Screenlets in Liferay Screens for iOS

Liferay Screens for iOS contains several Screenlets that you can use in your iOS apps. This section contains the reference documentation for each. If you're looking for instructions on using Screens, see the Screens tutorials. The Screens tutorials contain instructions on using Screenlets and using Themes in Screenlets. Each Screenlet reference document here lists the Screenlet's features, compatibility, its module (if any), available Themes, attributes, delegate methods, and more. The available Screenlets are listed here with links to their reference documents:

- **Login Screenlet:** Signs users in to a Liferay DXP instance.
- **Sign Up Screenlet:** Registers new users in a Liferay DXP instance.
- **Forgot Password Screenlet:** Sends emails containing a new password or password reset link to users.
- **User Portrait Screenlet:** Shows the user's portrait picture.
- **DDL Form Screenlet:** Presents dynamic forms to be filled out by users and submitted back to the server.
- **DDL List Screenlet:** Shows a list of records based on a pre-existing DDL in a Liferay DXP instance.
- **Asset List Screenlet:** Shows a list of assets managed by Liferay DXP's Asset Framework. This includes web content, blog entries, documents, and more.
- **Web Content Display Screenlet:** Shows the web content's HTML or structured content. This Screenlet uses the features available in Web Content Management.
- **Web Content List Screenlet:** Shows a list of web contents from a folder, usually based on a pre-existing `DDMStructure`.
- **Image Gallery Screenlet:** Shows a list of images from a folder. This Screenlet also lets users upload and delete images.
- **Rating Screenlet:** Shows the rating for an asset. This Screenlet also lets the user update or delete the rating.
- **Comment List Screenlet:** Shows a list of comments for an asset.

- **Comment Display Screenlet:** Shows a single comment for an asset.
- **Comment Add Screenlet:** Lets the user comment on an asset.
- **Asset Display Screenlet:** Displays an asset. Currently, this Screenlet can display Documents and Media Library files (`DLFileEntry` entities), blog articles (`BlogsEntry` entities), and web content articles (`WebContent` entities). You can also use it to display custom assets.
- **Blogs Entry Display Screenlet:** Shows a single blogs entry.
- **Image Display Screenlet:** Shows a single image file from a Liferay DXP instance's Documents and Media Library.
- **Video Display Screenlet:** Shows a single video file from a Liferay DXP instance's Documents and Media Library.
- **Audio Display Screenlet:** Shows a single audio file from a Liferay DXP instance's Documents and Media Library.
- **PDF Display Screenlet:** Shows a single PDF file from a Liferay DXP instance's Documents and Media Library.
- **File Display Screenlet:** Shows a single file from a Liferay DXP instance's Documents and Media Library. Use this Screenlet to display file types not covered by the other display Screenlets (e.g., DOC, PPT, XLS).
- **Web Screenlet:** Displays any web page. You can also customize the web page through injection of local and remote JavaScript and CSS files.

129.1 Login Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Login Screenlet authenticates portal users in your iOS app. The following authentication methods are supported:

- **Basic:** uses user login and password according to HTTP Basic Access Authentication specification. Depending on the authentication method used by your Liferay instance, you need to provide the user's email address, screen name, or user ID. You also need to provide the user's password.
- **OAuth:** implements the OAuth 1.0a specification.
- **Cookie:** uses a cookie to log in. This lets you access documents and images in the portal's document library without the guest view permission in the portal. The other authentication types require this permission to access such files.

Note: Cookie authentication with Login Screenlet is broken in fix packs 14 through 18 of Liferay Digital Enterprise 7.0. This issue is fixed in newer fix packs for Liferay Digital Enterprise 7.0.

For instructions on configuring the Screenlet to use these authentication types, see the below Portal Configuration and Screenlet Attributes sections.

When a user successfully authenticates, their attributes are retrieved for use in the app. You can use the `SessionContext` class to get the current user's attributes.

Note that user credentials and attributes can be stored securely in the keychain (see the `saveCredentials` attribute). Stored user credentials can be used to automatically log the user in to subsequent sessions. To do this, you can use the method `SessionContext.loadStoredCredentials()` method.

JSON Services Used

Screenlets in Liferay Screens call the portal's JSON web services. This Screenlet calls the following services and methods.

Service	Method	Notes
`UserService`	`getUserByEmailAddress`	Basic login
`UserService`	`getUserByScreenName`	Basic login
`UserService`	`getUserById`	Basic login
`UserService`	`getCurrentUser`	Cookie and OAuth login

Module

- Auth

Themes

- Default (`default`)
- Flat7 (`flat7`)

For instructions on using Themes, click here.

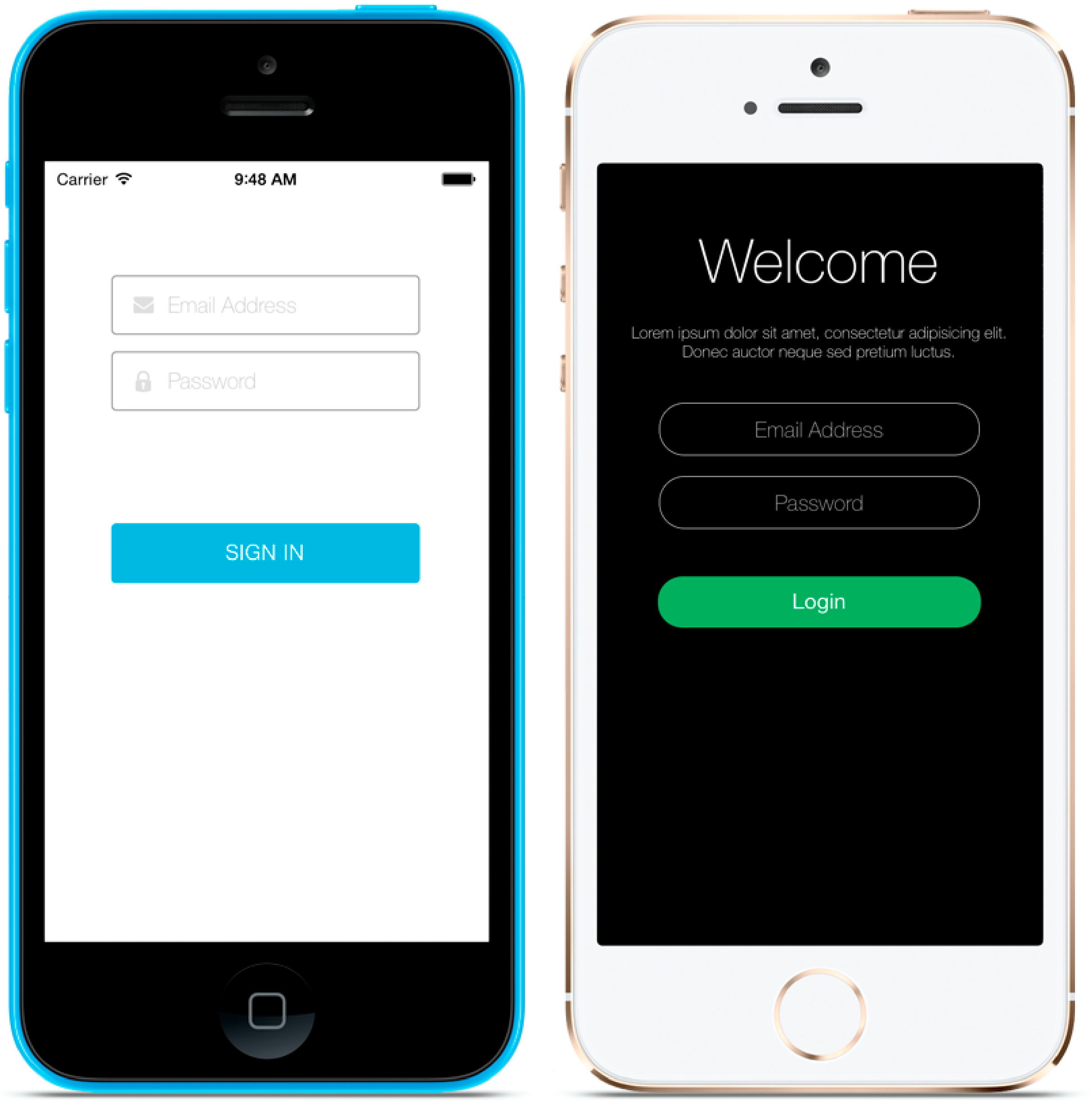
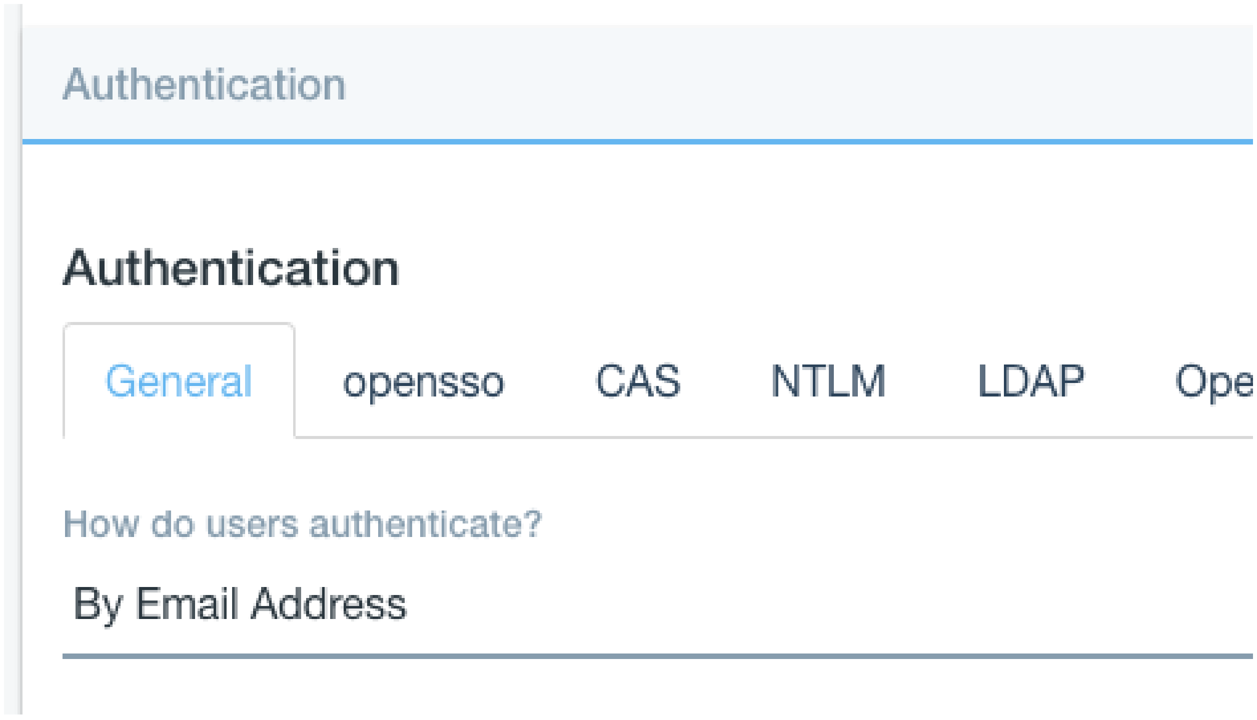

Carrier
9:48 AM
Email Address
Password
SIGN IN
Welcome
Lorem ipsum dolor sit amet, consectetur adipisicing elit. Donec auctor neque sed pretium luctus.
Email Address
Password
Login

Figure 129.1: The Login Screenlet using the Default and Flat7 Themes.

Portal Configuration

Basic Authentication

Before using Login Screenlet, you should make sure your portal is configured with the authentication option you want to use. You can choose email address, screen name, or user ID. You can set this in the Control Panel by selecting *Configuration* → *Instance Settings*, and then selecting the *Authentication* section. The authentication options are in the *How do users authenticate?* selector menu.



How do users authenticate?
By Email Address

Figure 129.2: Setting the authentication method in your Liferay instance.

For more details, please refer to the Setting up a Liferay Instance section of the User Guide.

OAuth

Note: OAuth authentication is only available in Liferay DXP instances.

To use OAuth authentication, first install the OAuth provider app from the Liferay Marketplace. Click here to get this app. Once it's installed, go to *Control Panel* → *Users* → *OAuth Admin*, and add a new application to be used from Liferay Screens. Once the application exists, copy the *Consumer Key* and *Consumer Secret* values for later use in Login Screenlet.

Offline

This Screenlet doesn't support offline mode. It requires network connectivity. If you need to log in users automatically, even when there's no network connection, you can use the `saveCredentials` attribute together with the `SessionContext.loadStoredCredentials()` method.

Attributes

Attribute | Data type | Explanation | `companyId` | `number` | The ID of the portal instance to authenticate to. If you don't set this attribute or set it to `0`, the Screenlet uses the `companyId` setting in `LiferayServerContext`. | `loginMode` | `string` | The Screenlet's authentication type. You can set this attribute to `basic`, `oauth`, or `cookie`. If you don't set this attribute, the Screenlet defaults to basic authentication. | `basicAuthMethod` | `string` | Specifies the basic authentication option to use. You can set this attribute to `email`, `screenName` or `userId`. This must match the server's authentication option. If you don't set this attribute, and don't set the `loginMode` attribute to `oauth` or `cookie`, the Screenlet defaults to basic authentication with the `email` option. | `OAuthConsumerKey` | `string` | Specifies the *Consumer Key* to use in OAuth authentication. | `OAuthConsumerSecret` | `string` | Specifies the *Consumer Secret* to use in OAuth authentication. | `saveCredentials` | `boolean` | When set, the user credentials and attributes are stored securely in the keychain. This information can then be loaded in subsequent sessions by calling the `SessionContext.loadStoredCredentials()` method. | `shouldHandleCookieExpiration` | `bool` | Whether to refresh the cookie automatically when using cookie login. When set to `true` (the default value), the cookie refreshes as it's about to expire. | `cookieExpirationTime` | `int` | How long the cookie lasts, in seconds. This value depends on your portal instance's configuration. The default value is `900`. |

Delegate

The Login Screenlet delegates some events to an object that conforms to the `LoginScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onLoginResponseUserAttributes:`: Called when login successfully completes. The user attributes are passed as a dictionary of keys (`String` or `NSStrings`) and values (`AnyObject` or `NSObject`). The supported keys are the same as the portal's User entity.

- `- screenlet:onLoginError:`: Called when an error occurs during login. The `NSError` object describes the error.

- `- screenlet:onCredentialsSavedUserAttributes:`: Called when the user credentials are stored after a successful login.

- `- screenlet:onCredentialsLoadedUserAttributes:`: Called when the user credentials are retrieved. Note that this only occurs when the Screenlet is used and stored credentials are available.

Challenge-Response Authentication

To support challenge-response authentication when using a cookie to log in to the portal, the `SessionContext` class has a `challengeResolver` attribute. For more information about how iOS handles challenge-response authentication, see the article Authentication Challenges and TLS Chain Validation.

The challenge resolver type is a closure or block that receives two parameters:

1. `URLAuthenticationChallenge`
2. Another closure or block. You must call this to resolve the challenge (e.g., by passing credentials, canceling the challenge, etc.). You can do this by passing a `URLSession.AuthChallengeDisposition`.

Here's an example that sends a basic authorization in response to an authentication challenge:

```
SessionContext.challengeResolver = resolver

func resolver(challenge: URLAuthenticationChallenge,
    decisionCallback: (URLSession.AuthChallengeDisposition, URLCredential) -> Void) {

    // Use the challenge variable to get information about the challenge itself
    if challenge.previousFailureCount == 0 {
        // To solve the challenge, call the decision callback with your decision
        // Pass the credentials to the server
        decisionCallback(.useCredential, URLCredential(user: "user", password: "password",
            persistence: .forSession))
    }
    else {
        // Something went wrong, so let the system handle the challenge
        decisionCallback(.performDefaultHandling, URLCredential(user: "these credentials",
            password: "are ignored", persistence: .none))
    }

}
```

129.2 Sign Up Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Sign Up Screenlet creates a new user in your Liferay instance: a new user of your app can become a new user in your portal. You can also use this Screenlet to save the credentials of the new user in their keychain. This enables auto login for future sessions. The Screenlet also supports navigation of form fields from the keyboard of the user's device.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`UserService`	`addUser`	

Module

- Auth

Themes

- Default (`default`)
- Flat7 (`flat7`)

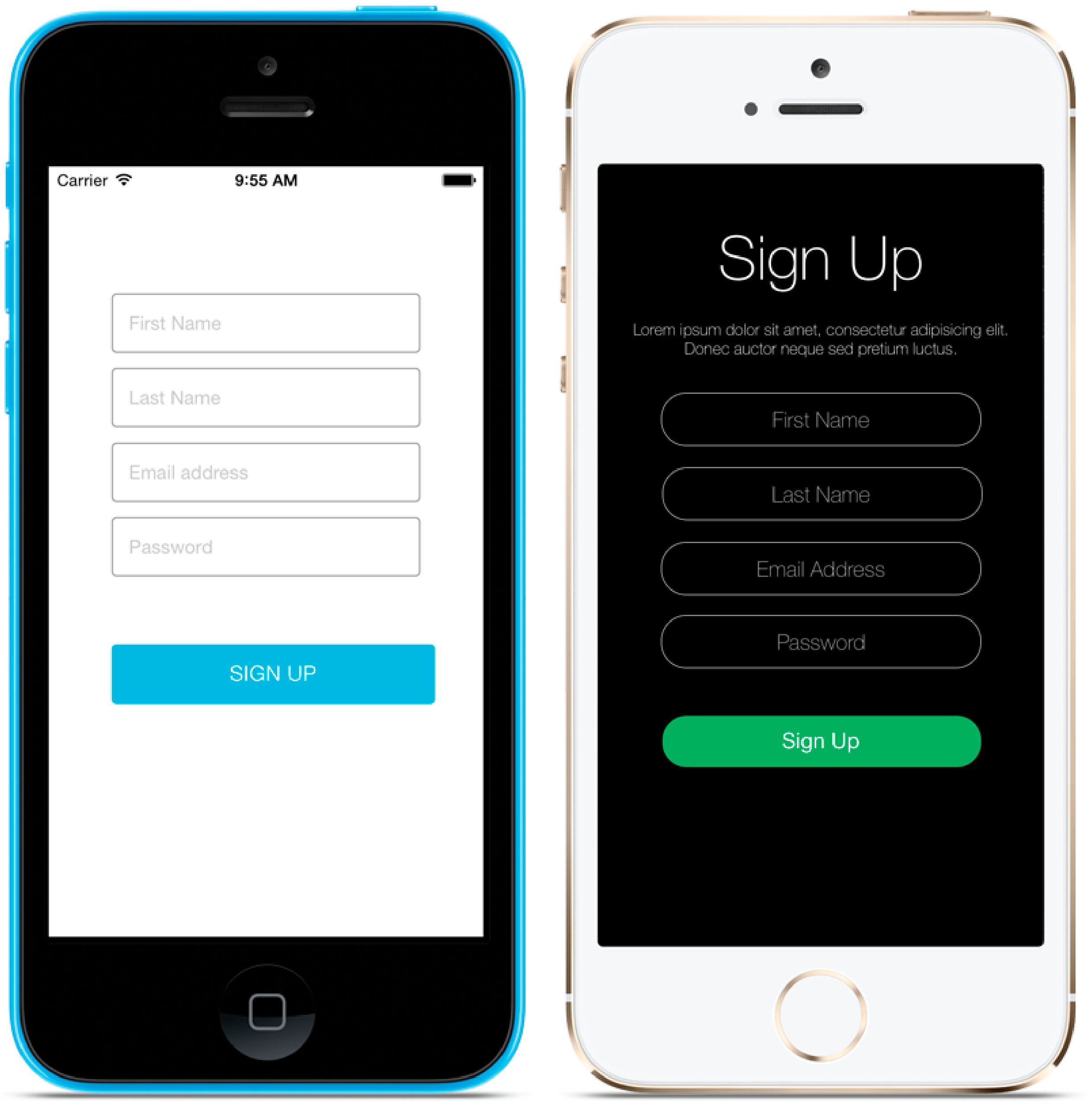
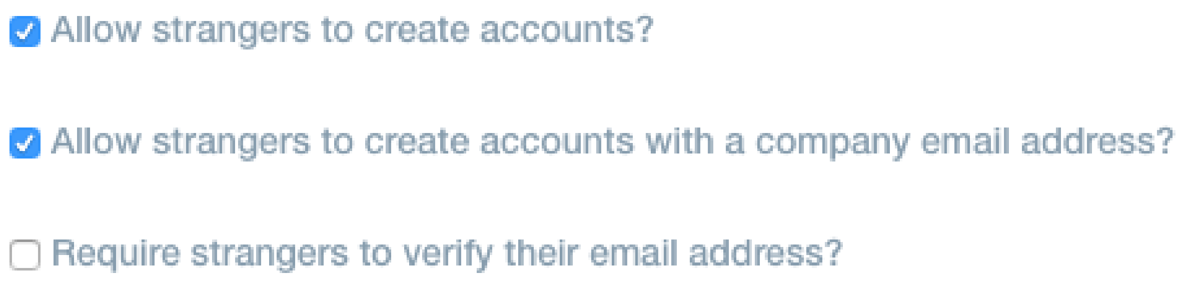


Figure 129.3: The Sign Up Screenlet with the Default and Flat7 Themes.

Portal Configuration

Sign Up Screenlet's corresponding configuration in the Liferay instance can be set in the Control Panel by selecting *Configuration* → *Instance Settings*, and then selecting the *Authentication* section.



☑ Allow strangers to create accounts?

☑ Allow strangers to create accounts with a company email address?

☐ Require strangers to verify their email address?

Figure 129.4: The Liferay instance's authentication settings.

For more details, please refer to the Setting up a Liferay Instance section of the User Guide.

Anonymous Request

Anonymous requests are unauthenticated requests. Authentication is needed, however, to call the API. To allow this operation, the portal administrator should create a specific user with minimal permissions.

Offline

This Screenlet doesn't support offline mode. It requires network connectivity.

Attributes

Attribute | Data type | Explanation | `anonymousApiUserName` | `string` | The user name, email address, or user ID (depending on the portal's authentication method) to use for authenticating the request. | `anoymousApiPassword` | `string` | The password for use in authenticating the request. | `companyId` | `number` | When set, authentication is done for a user in the specified company. If the value is `0`, the company specified in `LiferayServerContext` is used. | `autoLogin` | `boolean` | Whether the user is logged in automatically after a successful sign up. | `saveCredentials` | `boolean` | Sets whether or not the user's credentials and attributes are stored in the keychain after a successful log in. This attribute is ignored if `autologin` is disabled. |

Delegate

The Sign Up Screenlet delegates some events to an object that conforms to the `SignUpScreenletDelegate` protocol. If the `autologin` attribute is enabled, login events are delegated to an object conforming to the `LoginScreenletDelegate` protocol. Refer to the `LoginScreenlet` documentation for more details.

The `SignUpScreenletDelegate` protocol lets you implement the following methods:

- `- screenlet:onSignUpResponseUserAttributes::` Called when sign up successfully completes. The user attributes are passed as a dictionary of keys (`String` or `NSStrings`) and values (`AnyObject` or `NSObject`). The supported keys are the same as Liferay Portal's User entity.
- `- screenlet:onSignUpError::` Called when an error occurs in the process. The `NSError` object describes the error.

129.3 Forgot Password Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Forgot Password Screenlet sends emails to registered users with their new passwords or password reset links, depending on the server configuration. The available authentication methods are:

- Email address
- Screen name
- User id

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`UserService`	`sendPasswordByEmailAddress`	
`UserService`	`sendPasswordByUserId`	
`UserService`	`sendPasswordByScreenName`	

Module

- Auth

Themes

- Default (`default`)
- Flat7 (`flat7`)

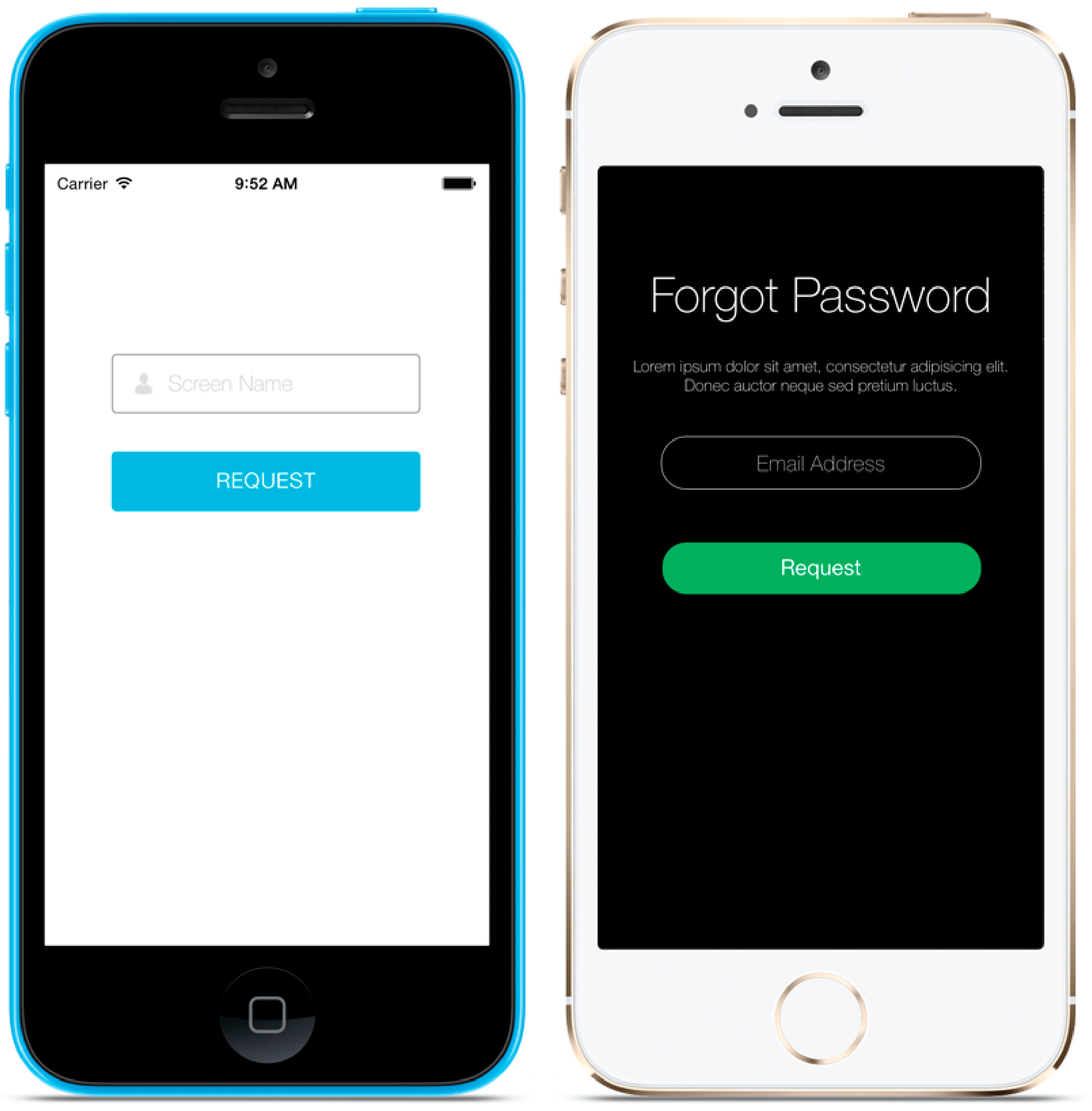

Carrier
9:52 AM
Screen Name
REQUEST
Forgot Password
Lorem ipsum dolor sit amet, consectetur adipisicing elit.
Donec auctor neque sed pretium luctus.
Email Address
Request

Figure 129.5: The Forgot Password Screenlet with the Default and Flat7 Themes.

Portal Configuration

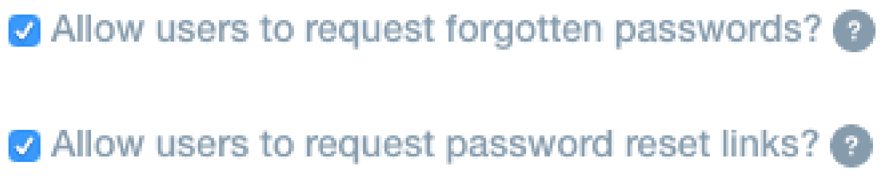
To use the Forgot Password Screenlet, you must allow users to request new passwords in the portal. The next sections show you how to do this.

Authentication Method

Note that the authentication method configured in the portal can be different from the one used by this Screenlet. For example, it's *perfectly fine* to use `screenName` for sign in authentication, but allow users to recover their password using the `email` authentication method.

Password Reset

You can set the Liferay instance's corresponding password reset options in the Control Panel by selecting *Configuration* → *Instance Settings*, and then selecting the *Authentication* section. The Screenlet's password functionality depends on the authentication settings in the portal:



✓ Allow users to request forgotten passwords?

✓ Allow users to request password reset links?

Figure 129.6: Checkboxes for the password recovery features in Liferay Portal.

If both of these options are unchecked, password recovery is disabled. If both options are checked, an email containing a password reset link is sent when a user requests it. If only the first option is checked, an email containing a new password is sent when a user requests it.

For more details on authentication in Liferay Portal, please refer to the Setting up a Liferay Instance section of the User Guide.

Anonymous Request

An anonymous request can be made without the user being logged in. However, authentication is needed to call the API. To allow this operation, the portal administrator should create a specific user with minimal permissions.

Offline

This Screenlet doesn't support offline mode. It requires network connectivity.

Attributes

Attribute | Data type | Explanation | `anonymousApiUserName` | `string` | The user name, email address, or userId (depending on the portal's authentication method) to use for authenticating the request. | `anonymousApiPassword` | `string` | The password to use to authenticate the request. | `companyId` | `number` | When set, the authentication is done for a user within the specified company. If the value is `0`, the company specified in `LiferayServerContext` is used. | `basicAuthMethod` | `string` | The authentication method that is presented to the user. This can be `email`, `screenName`, or `userId`. |

Delegate

The Forgot Password Screenlet delegates some events to an object that conforms to the `ForgotPasswordScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onForgotPasswordSent:`: Called when a password reset email is successfully sent. The Boolean parameter indicates whether the email contains the new password or a password reset link.
- `- screenlet:onForgotPasswordError:`: Called when an error occurs in the process. The `NSError` object describes the error.

129.4 User Portrait Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The User Portrait Screenlet shows the user's portrait from Liferay Portal. If the user doesn't have a portrait configured, a placeholder image is shown.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`UserService`	`getUserById`	
`UserService`	`getUserByEmailAddress`	
`UserService`	`getUserByScreenName`	

Module

- None

Themes

- Default (`default`)
- Flat7 (`flat7`)

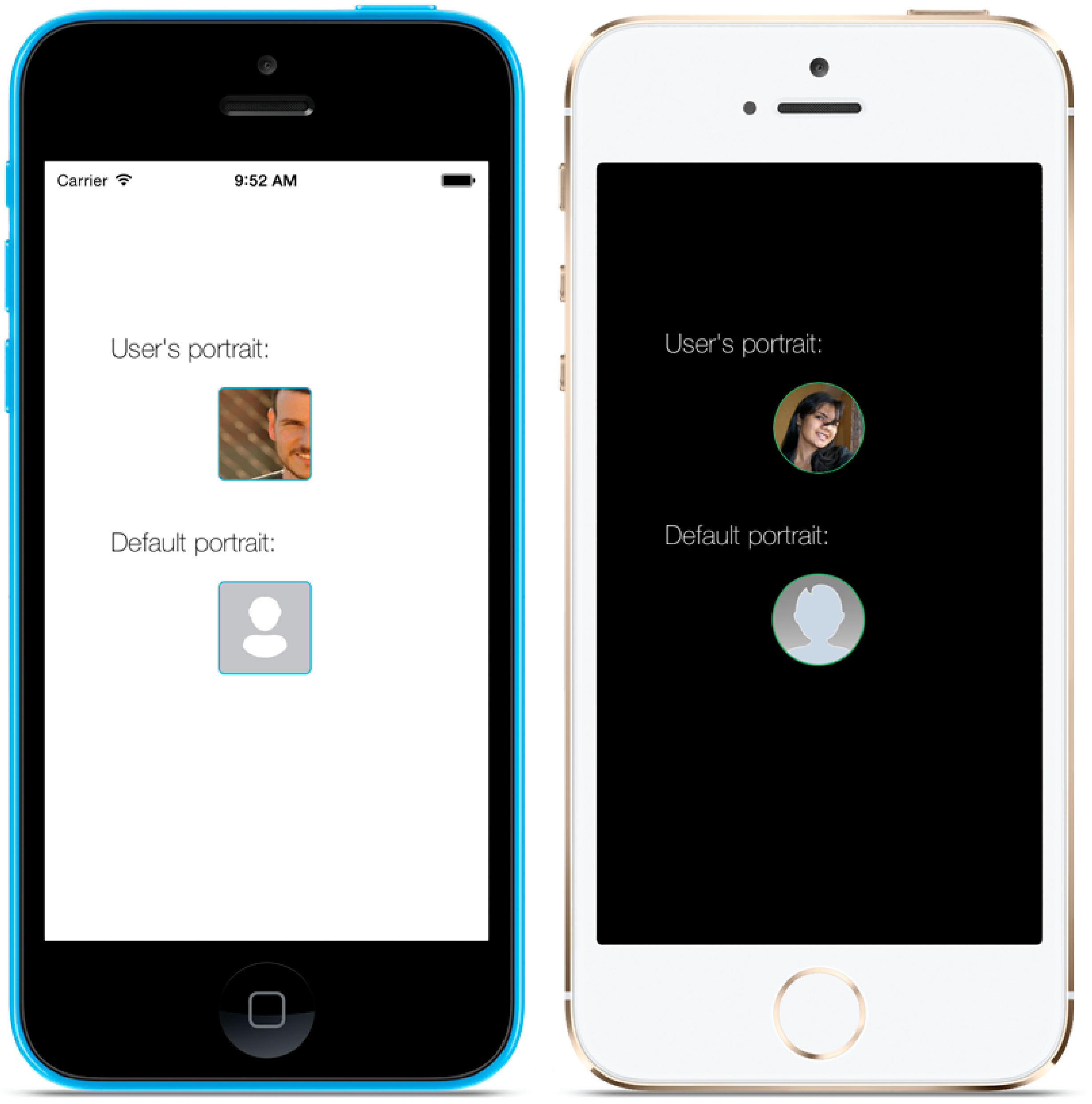


Figure 129.7: The User Portrait Screenlet using the Default and Flat7 Themes.

Portal Configuration

None

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture.

When loading the portrait, the Screenlet supports the following offline mode policies:

Policy	What happens	When to use
`remote-only`	The Screenlet loads the user portrait from the portal. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet loads the portrait, it stores the received image in the local cache for later use.	Use this policy when you always need to show updated portraits, and show the default placeholder when there's no connection.
`cache-only`	The Screenlet loads the user portrait from the local cache. If the portrait isn't there, the Screenlet uses the delegate to notify the developer about the error.	Use this policy to show local portraits, without retrieving remote information under any circumstance.
`remote-first`	The Screenlet loads the user portrait from the portal. The Screenlet displays the portrait to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the portrait from the local cache. If the portrait doesn't exist there, the Screenlet uses the delegate to notify the developer about the error.	Use this policy to show the most recent portrait when connected, but show a potentially outdated version when there's no connection.
`cache-first`	If the portrait exists in the local cache, the Screenlet loads it from there. If it doesn't exist there, the Screenlet requests the portrait from the portal and uses the delegate to notify the developer about any connection errors.	Use this policy to save bandwidth and loading time in the event a local (but probably outdated) portrait exists.

When editing the portrait, the Screenlet supports the following offline mode policies:

Policy	What happens	When to use
`remote-only`	The Screenlet sends the user portrait to the portal. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error, but it also discards the new portrait.	Use this policy when you need to make sure portal always has the most recent version of the portrait.
`cache-only`	The Screenlet stores the user portrait in the local cache.	Use this policy when you need to save the portrait locally, but don't want to change the portrait in the portal.
`remote-first`	The Screenlet sends the user portrait to the portal. If this succeeds, the Screenlet also stores the portrait in the local cache for later usage. If a connection issue occurs, the Screenlet stores the portrait in the local cache with the *dirty flag* enabled. This causes the portrait to be sent to the portal when the synchronization process runs.	Use this policy when you need to make sure the Screenlet sends the new portrait to the portal as soon as the connection is restored.
`cache-first`	The Screenlet stores the user portrait in the local cache and then sends it to the portal. If a connection issue occurs, the Screenlet stores the portrait in the local cache with the *dirty flag* enabled. This causes the portrait to be sent to the portal when the synchronization process runs.	Use this policy when you need to make sure the Screenlet sends the new portrait to the portal as soon as the connection is restored. Compared to `remote-first`, this policy always stores the portrait in the cache. The `remote-first` policy only stores the new image in the event of a network error.

Attributes

Attribute | Data type | Explanation | `borderWidth` | `number` | The size in pixels for the portrait's border. The default value is 1. Set this to `0` if you want to hide the border.| `borderColor` | `UIColor` | The border's color. Use the system's transparent color to hide the border. | `editable` | `boolean` | Lets the user change the portrait image by taking a photo or selecting a gallery picture. The default value is `false`. Portraits loaded with the `load(portraitId, uuid, male)` method aren't editable. | `offlinePolicy` | `string` | Configure the loading and saving behavior in case of connectivity issues. For more details, read the "Offline" section below. |

Methods

Method | Return | Explanation | `loadLoggedUserPortrait()` | `boolean` | Starts the request to load the currently logged in user's portrait image (see the `SessionContext` class). | `load(userId)` | `boolean` | Starts the request to load the specified user's portrait image. | `load(portraitId, uuid, male)` | `boolean` | Starts the request to load the portrait image using the specified user's data. The parameters `portraitId` and `uuid` can be retrieved by using the `SessionContext.userAttributes()` method. | `load(companyId, emailAddress)` | `boolean` | Starts the request to load the portrait image using the user's email address. | `load(companyId, screenName)` | `boolean` | Starts the request to load the portrait image using the user's screen name. |

Delegate

The User Portrait Screenlet delegates some events to an object that conforms to the `UserPortraitScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onUserPortraitResponseImage:`: Called when an image is received from the server. You can then apply image filters (grayscale, for example) and return the new image. You can return the original image supplied as the argument if you don't want to modify it.
- `- screenlet:onUserPortraitError:`: Called when an error occurs in the process. The `NSError` object describes the error.
- `- screenlet:onUserPortraitUploaded:`: Called when a new portrait is uploaded to the server. You receive the user attributes as a parameter.
- `- screenlet:onUserPortraitUploadError:`: Called when an error occurs in the upload process. The `NSError` object describes the error.

129.5 DDL Form Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

DDL Form Screenlet can be used to show a collection of fields so that a user can fill in their values. Initial or existing values may be shown in the fields. Fields of the following data types are supported:

- *Boolean*: A two state value typically shown using a checkbox.
- *Date*: A formatted date value. The format depends on the device's locale.
- *Decimal, Integer, and Number*: A numeric value.
- *Document and Media*: A file stored on the current device. It can be uploaded to a specific portal repository.
- *Radio*: A set of options to choose from. A single option must be chosen.
- *Select*: A selection box of options to choose from. A single option must be chosen.
- *Text*: A single line of text.
- *Text Box*: Supports multiple lines of text.

DDL Form Screenlet also supports the following features:

- Stored records can support a specific workflow.
- A Submit button can be shown at the end of the form.
- Required values and validation for fields can be used.
- Users can traverse the form fields from the keyboard.
- Supports i18n in record values and labels.

There are also a few limitations you should be aware of when using DDL Form Screenlet. They are listed here:

- Nested fields in the data definition aren't supported.
- Selection of multiple items in the Radio and Select data types isn't supported yet.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`DDMStructureService`	`getStructureWithStructureId`	Load form
`ScreensddlrecordService` (Screens compatibility plugin)	`getDdlRecord`	Load record
`DLAppService`	`addFileEntry`	Upload document
`DDLRecordService`	`addRecord`	Submit form

Service	Method	Notes
`DDLRecordService`	`updateRecord`	Update form

Module

- DDL

Themes

- Default

The Default Theme uses a standard `UITableView` to show a scrollable list of fields. Other Themes may use a different component, such as `UICollectionView` or others, to show the fields.

Custom Cells

A Theme needs to define a cell view for each field type. For instance, the `xib` file `DDLFieldDateTableCell_default` is used to render `Date` fields in the Default Theme.

If you want a specific field to have a unique appearance, you can customize your field's display by using the following filename pattern, where `XXX` is your field's name: `DDLCustomFieldXXXTableCell_default`. For example, the "Are you a subscriber?" field in screenshot above shows how text fields appear in the Default Theme. If you want to customize this, you don't need to create an entire Theme. You just need to create an `xib` file for the field `subscriberName`. The filename is therefore `DDLCustomFieldSubscriberNameTableCell_default`. Be careful to keep the same components and `IBOutlet` defined in the custom file.

Portal Configuration

Before using DDL Form Screenlet, you should make sure that Dynamic Data Lists and Data Types are configured properly in the portal. Refer to the Creating Data Definitions and Creating Data Lists sections of the User Guide for more details. If Workflow is required, it must also be configured. See the Using Workflow section of the User Guide for details.

Permissions

To use DDL Form Screenlet to add new records, you must grant the Add Record permission in the Dynamic Data List in the portal. If you want to use DDL Form Screenlet to view or edit record values, you must also grant the View and Update permissions, respectively. The Add Record, View, and Update permissions are highlighted by the red boxes in the following screenshot:

Also, if your form includes at least one Documents and Media field, you must grant permissions in the target repository and folder. For more details, see the `repositoryId` and `folderId` attributes below.

For more details, please see the User Guide sections Creating Data Definitions, Creating Data Lists, and Using Workflow.

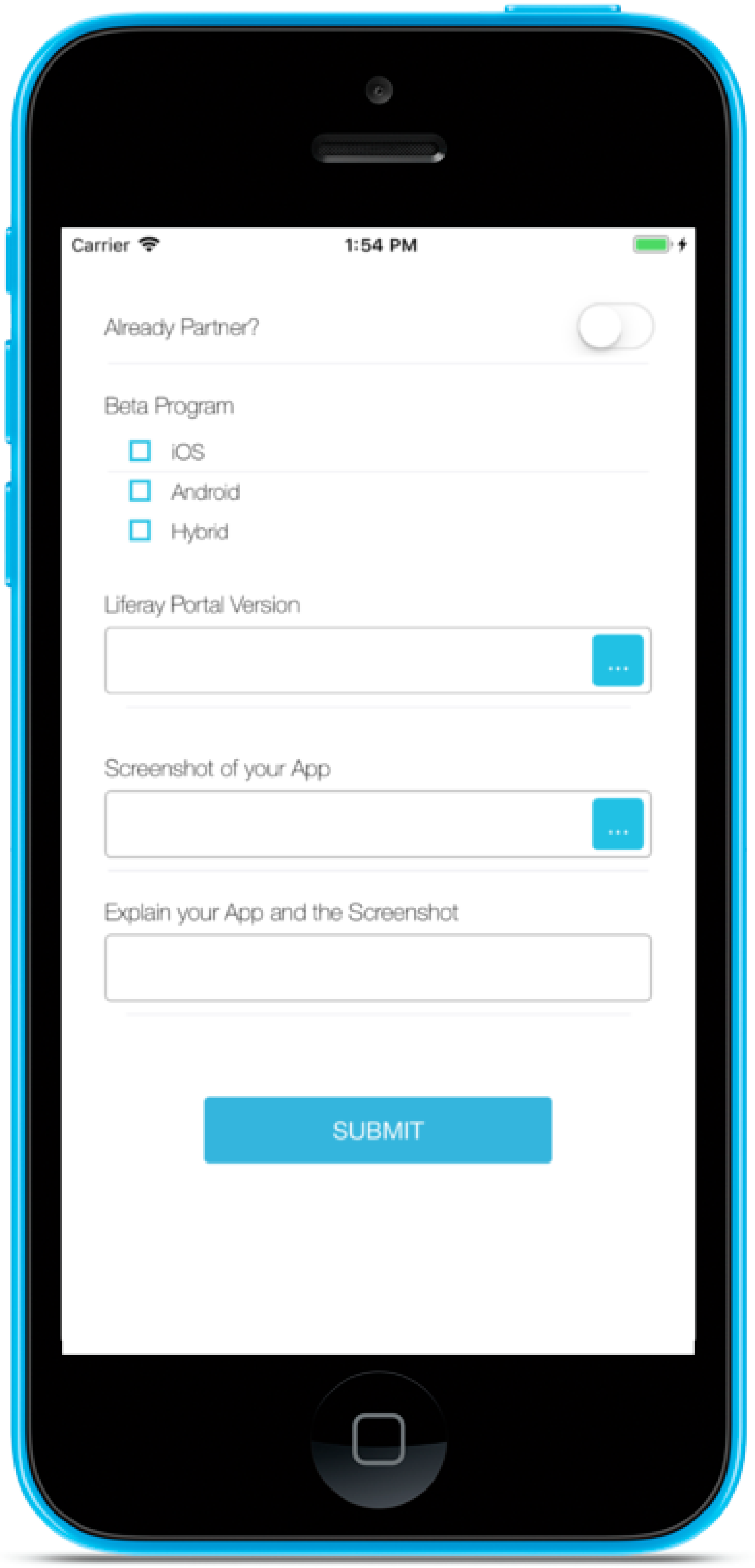
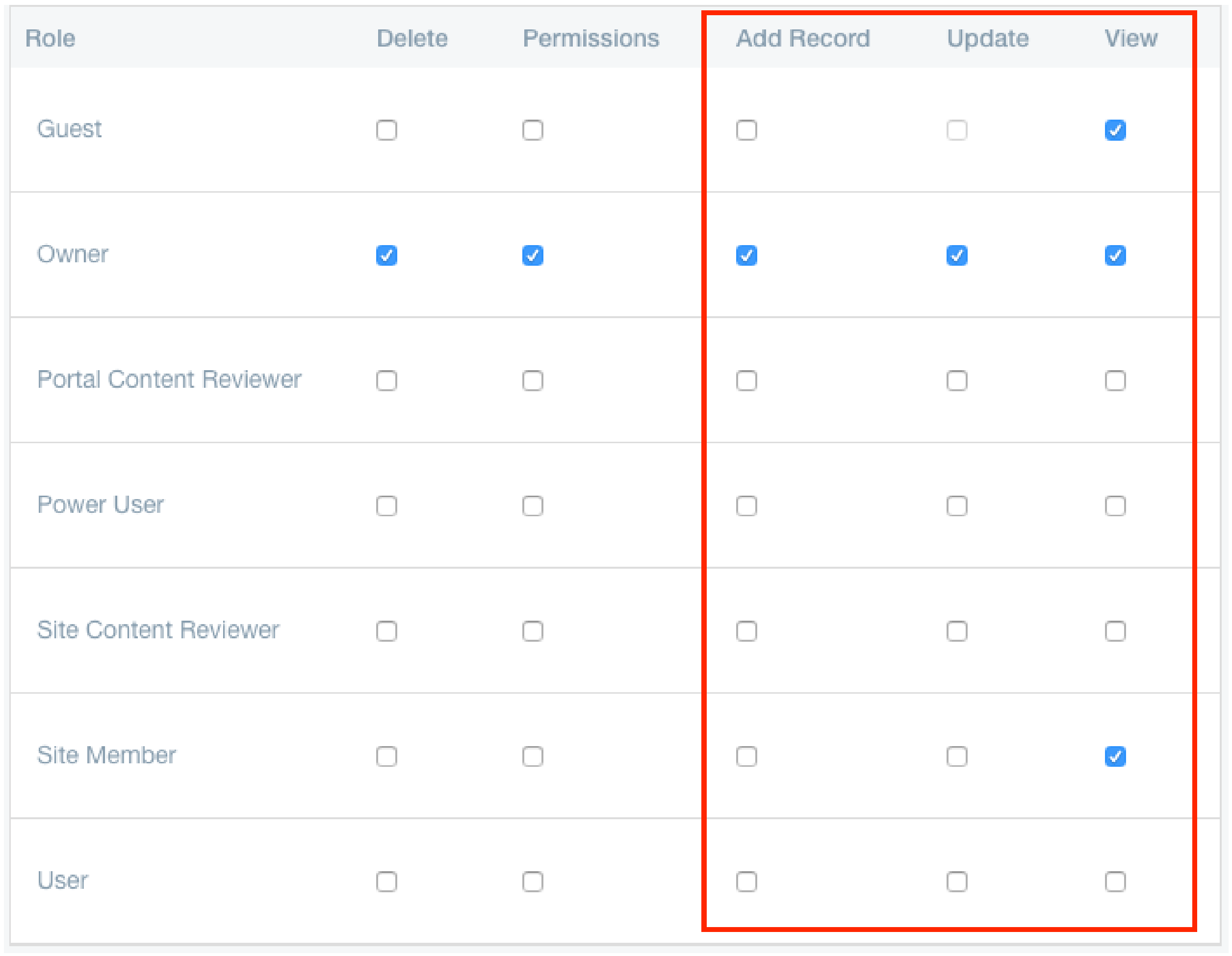


Figure 129.8: DDL Form Screenlet using the Default (`default`) Theme.



Role	Delete	Permissions	Add Record	Update	View
Guest	☐	☐	☐	☐	✓
Owner	✓	✓	✓	✓	✓
Portal Content Reviewer	☐	☐	☐	☐	☐
Power User	☐	☐	☐	☐	☐
Site Content Reviewer	☐	☐	☐	☐	☐
Site Member	☐	☐	☐	☐	✓
User	☐	☐	☐	☐	☐

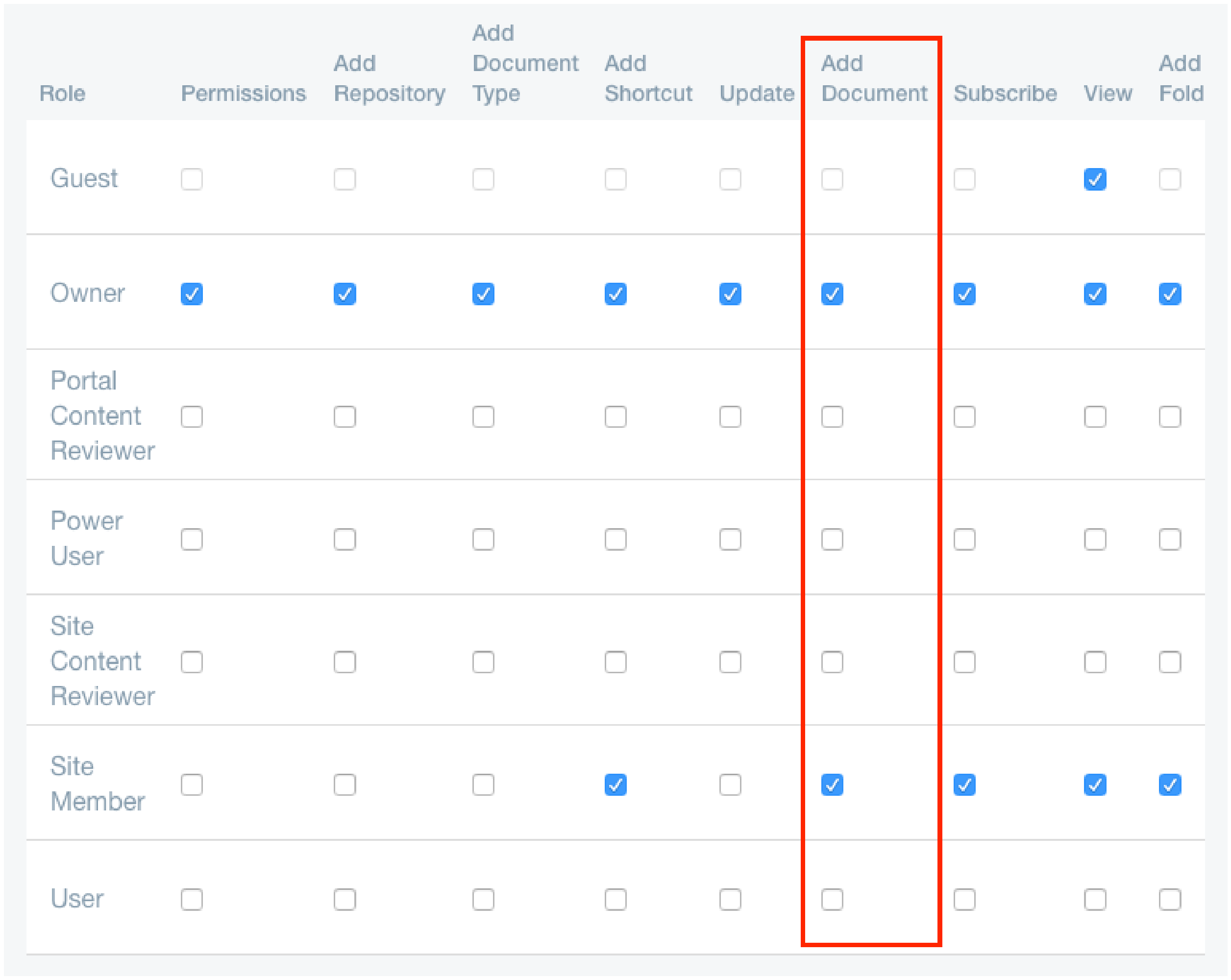
Figure 129.9: The permissions for adding, viewing, and editing DDL records.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture.

When loading the form or record, the Screenlet supports the following offline mode policies:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the form or record from the portal. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet loads the form or record, it stores the received data (record structure and data) in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection.| `cache-only` | The Screenlet loads the form or record from the local cache. If the form or record isn't there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance.| `remote-first` | The Screenlet requests the form or record from the portal. The Screenlet shows the record or form to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the form or record from the local cache. If the form or record doesn't exist there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy to show the most recent version of the



Role	Permissions	Add Repository	Add Document Type	Add Shortcut	Update	Add Document	Subscribe	View	Add Fold
Guest	☐	☐	☐	☐	☐	☐	☐	✓	☐
Owner	✓	✓	✓	✓	✓	✓	✓	✓	✓
Portal Content Reviewer	☐	☐	☐	☐	☐	☐	☐	☐	☐
Power User	☐	☐	☐	☐	☐	☐	☐	☐	☐
Site Content Reviewer	☐	☐	☐	☐	☐	☐	☐	☐	☐
Site Member	☐	☐	☐	✓	☐	✓	✓	✓	✓
User	☐	☐	☐	☐	☐	☐	☐	☐	☐

Figure 129.10: The permission for adding a document to a Documents and Media folder.

data when connected, but show an outdated version when there's no connection. | `cache-first` | If the form or record exists in the local cache, the Screenlet loads it from there. If it doesn't exist there, the Screenlet requests it from the portal and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

When editing the record, the Screenlet supports the following offline mode policies:

Policy | What happens | When to use | `remote-only` | The Screenlet sends the record to the portal. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error, but it also discards the record. | Use this policy to make sure the portal always has the most recent version of the record. | `cache-only` | The Screenlet stores the record in the local cache. | Use this policy when you need to save the data locally, but don't want to update the data in the portal (update or add record). | `remote-first` | The Screenlet sends the record to the portal. If this succeeds, it also stores the record in the local cache for later usage. If a connection issue occurs, then Screenlet stores the record in the local cache with the *dirty flag* enabled. This causes the synchronization process to send the record to the portal when it runs. | Use this policy when you need to make sure the Screenlet sends the record to the portal as soon as the connection is restored. | `cache-first` | The Screenlet stores the record in the local cache and then sends it to the remote portal. If a connection issue occurs, then Screenlet stores the record in the local cache with the *dirty flag* enabled. This causes the the synchronization process to send the record to the portal when it runs. | Use this policy when you need to make sure the Screenlet sends the record to the portal as soon as the connection is restored. Compared to `remote-first`, this policy always stores the record in the cache. The `remote-first` policy only stores the record in the event of a network error. |

Required Attributes

- `structureId`
- `recordSetId`

Attributes

Attribute | Data Type | Explanation | `structureId` | `number` | This is the identifier of a data definition for your site in Liferay. To find the identifiers for your data definitions, click *Admin* from the Dockbar and select *Content*. Then click *Dynamic Data Lists* and click the *Manage Data Definitions* button. The identifier of each data definition is in the ID column of the table that appears. | `groupId` | `number` | The site (group) identifier where the record is stored. If this value is `0`, the `groupId` specified in `LiferayServerContext` is used. | `recordSetId` | `number` | The identifier of a dynamic data list. To find the identifiers for your dynamic data lists, click *Admin* from the Dockbar and select *Content*. Then click *Dynamic Data Lists*. The identifier of each dynamic data list is in the ID column of the table that appears. | `recordId` | `number` | The identifier of the record you want to show. Setting the `editable` attribute to `true` allows editing of the record's values. The `recordId` can be obtained from other methods or delegates. | `repositoryId` | `number` | The identifier of the Documents and Media repository to upload to. If this value is `0`, the default repository for the site specified in `groupId` is used. | `folderId` | `number` | The identifier of the folder where Documents and Media files are uploaded. If this value is `0`, the root folder is used. | `filePrefix` | `string` | The prefix to attach to the names of files uploaded to a Documents and Media repository. A random GUID string is appended following the prefix. | `autoLoad` | `boolean` | Sets whether or not the form is loaded when the Screenlet is shown. If `recordId` is set, the record value is loaded together with the form definition. | `autoscrollOnValidation` | `boolean` | Sets whether or not the form automatically scrolls to the first failed field when validation is used. | `showSubmitButton` | `boolean` | Sets whether or not the form shows a submit button at the bottom. If this is set to `false`, you should call the `submitForm()` method. | `editable` | `boolean` | Sets whether the values can be changed by the user. The default is `true`. |

Methods

Method | Return Type | Explanation | `loadForm()` | `boolean` | Starts the request to load the form definition. The form fields are shown when the response is received. This method returns `true` if the request is sent. | `loadRecord()` | `boolean` | Starts the request to load the record specified in `recordId`. If needed, the form definition is also loaded. The form fields are shown filled with record values when the response is received. This method returns `true` if the request is sent. | `submitForm()` | `boolean` | Starts the request to submit form values to the dynamic data list specified in `recordSetId`. All fields are validated prior to submission. Validation errors stop the submit process. |

Delegate

DDL Form Screenlet delegates some events to an object that conforms with the `DDLFormScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onFormLoaded:`: Called when the form is loaded. The second parameter (`record`) contains only field definitions.
- `- screenlet:onFormLoadError:`: Called when an error occurs while loading the form. The `NSError` object describes the error.
- `- screenlet:onRecordLoaded:`: Called when a form with values loads. The second parameter (`record`) contains field definitions and values. The method `onFormLoadResult` is called before `onRecordLoaded`.
- `- screenlet:onRecordLoadError:`: Called when an error occurs while loading a record. The `NSError` object describes the error.
- `- screenlet:onFormSubmitted:`: Called when the form values are successfully submitted to the server.
- `- screenlet:onFormSubmitError:`: Called when an error occurs while submitting the form. The `NSError` object describes the error.
- `- screenlet:onDocumentFieldUploadStarted:`: Called when the upload of a Documents and Media field begins.
- `- screenlet:onDocumentField:uploadedBytes:totalBytes:`: Called when a block of bytes in a Documents and Media field is uploaded. This method is intended to track progress of the uploads.
- `- screenlet:onDocumentField:uploadResult:`: Called when a Documents and Media field upload is completed.
- `- screenlet:onDocumentField:uploadError:`: Called when an error occurs in the Documents and Media upload process. The `NSError` object describes the error.

129.6 DDL List Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The DDL List Screenlet enables the following features:

- Shows a scrollable collection of DDL records.
- Implements fluent pagination with configurable page size.
- Allows filtering of records by creator.
- Supports i18n in record values.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensddlrecordService` (Screens compatibility plugin)	`getDdlRecords`	With `ddlRecordSetId`, or `ddlRecordSetId` and `userId`
`ScreensddlrecordService` (Screens compatibility plugin)	`getDdlRecordsCount`	

Module

- DDL

Themes

- The Default Theme uses a standard `UITableView` to show the scrollable list. Other Themes may use a different component, such as `UICollectionView` or others, to show the items.

Portal Configuration

Dynamic Data Lists (DDL) and Data Types should be configured in the portal. For more details, please refer to the Liferay User Guide sections Creating Data Definitions and Creating Data Lists.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

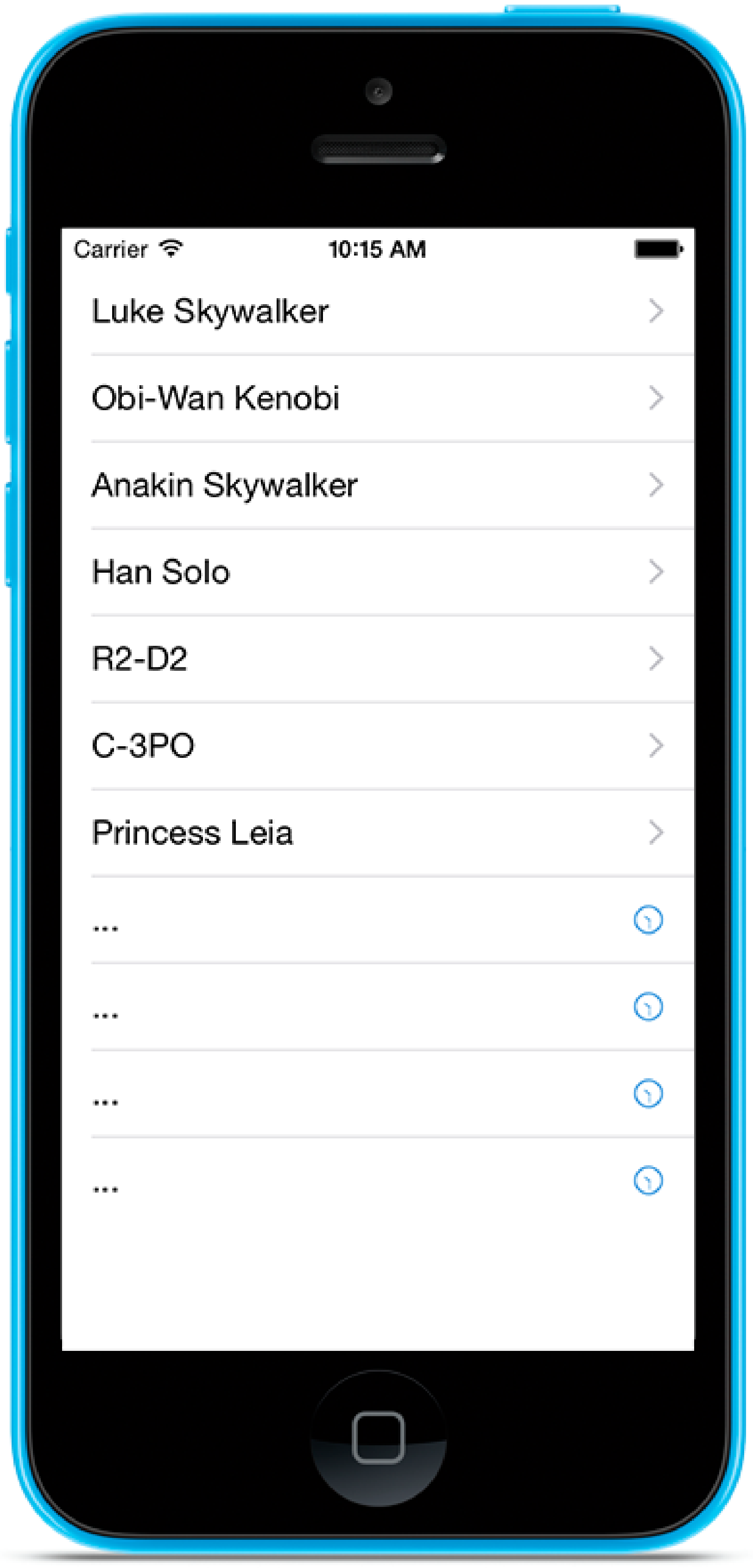


Figure 129.11: The DDL List Screenlet using the Default (`default`) Theme.

Policy | What happens | When to use | `remote-only` | The Screenlet loads the list from the portal. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet successfully loads the list, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the list from the portal. If this succeeds, the Screenlet shows the list to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the list from the local cache. If the list doesn't exist there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet requests it from the portal and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `recordSetId`
- `labelFields`

Attributes

Attribute | Data type | Explanation | `recordSetId` | `number` | The ID of the DDL being called. To find the IDs for your DDLs, first open the Product Menu and select the site that contains your DDLs. Then click *Content* → *Dynamic Data Lists*. Each DDL's ID is in the table's ID column. | `userId` | `number` | The ID of the user to filter records on. Records aren't filtered if the `userId` is `0`. The default value is `0`. | `labelFields` | `string` | The comma-separated names of the DDL fields to show. Refer to the list's data definition to find the field names. To do so, first open the Product Menu and select the site that contains your DDLs. Then click *Content* → *Dynamic Data Lists*, and find the find the icon (⋮) for the Dynamic Data List configuration menu at the upper right. Click this icon and select *Manage Data Definitions*. You can view the fields by clicking on any of the data definitions in the table that appears. Note that the appearance of these values in your app depends on the Theme selected by the user. | `offlinePolicy` | `string` | The offline mode setting. The default value is `remote-first`. See the Offline section for details. | `autoLoad` | `boolean` | Whether the list loads automatically when the Screenlet appears in the app's UI. The default value is `true`. | `refreshControl` | `boolean` | Whether a standard iOS `UIRefreshControl` appears when the user performs the pull to refresh gesture. The default value is `true`. | `firstPageSize` | `number` | The number of items retrieved from the server for display on the first page. The default value is `50`. | `pageSize` | `number` | The number of items retrieved from the server for display on the second and subsequent pages. The default value is `25`. | `obcClassName` | `string` | The name of the `OrderByComparator` class to use to sort the results. Omit this property if you don't want to sort the results. Click here to see some comparator classes. Note, however, that not all of these classes can be used with `obcClassName`. You can only use comparator classes that extend `OrderByComparator<DDLRecord>`. You can also create your own comparator classes that extend `OrderByComparator<DDLRecord>`. |

Methods

Method | Return | Explanation | `loadList()` | `boolean` | Starts the request to load the list of records. The list is shown when the response is received. This method returns true if the request is sent. |

Delegate

The DDL List Screenlet delegates some events in an object that conforms to the `DDLListScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onDDLListResponseRecords:`: Called when a page of contents is received. Note that this method may be called more than once; once for each retrieved page.
- `- screenlet:onDDLListError:`: Called when an error occurs in the process. The `NSError` object describes the error.
- `- screenlet:onDDLSelectedRecord:`: Called when an item in the list is selected.

129.7 Asset List Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Asset List Screenlet can be used to show lists of assets from a Liferay instance. For example, you can use the Screenlet to show a scrollable collection of assets. It also implements fluent pagination with configurable page size. The Asset List Screenlet can show assets of the following classes:

- `BlogsEntry`
- `BookmarksEntry`
- `BookmarksFolder`
- `CalendarEvent`
- `DLFileEntry`
- `DDLRecord`
- `DDLRecordSet`
- `Group`

- `JournalArticle` (Web Content)
- `JournalFolder`
- `Layout`
- `LayoutRevision`
- `MBThread`
- `MBCategory`
- `MBDiscussion`
- `MBMailingList`
- `Organization`
- `User`
- `WikiPage`
- `WikiPageResource`
- `WikiNode`

The Asset List Screenlet also supports i18n in asset values.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensddlrecordService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensddlrecordService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`
`AssetEntryService`	`getEntriesCount`	

Module

- None

Themes

- Default

The Default Theme uses a standard `UITableView` to show the scrollable list. Other Themes may use a different component, such as `UICollectionView` or others, to show the items.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

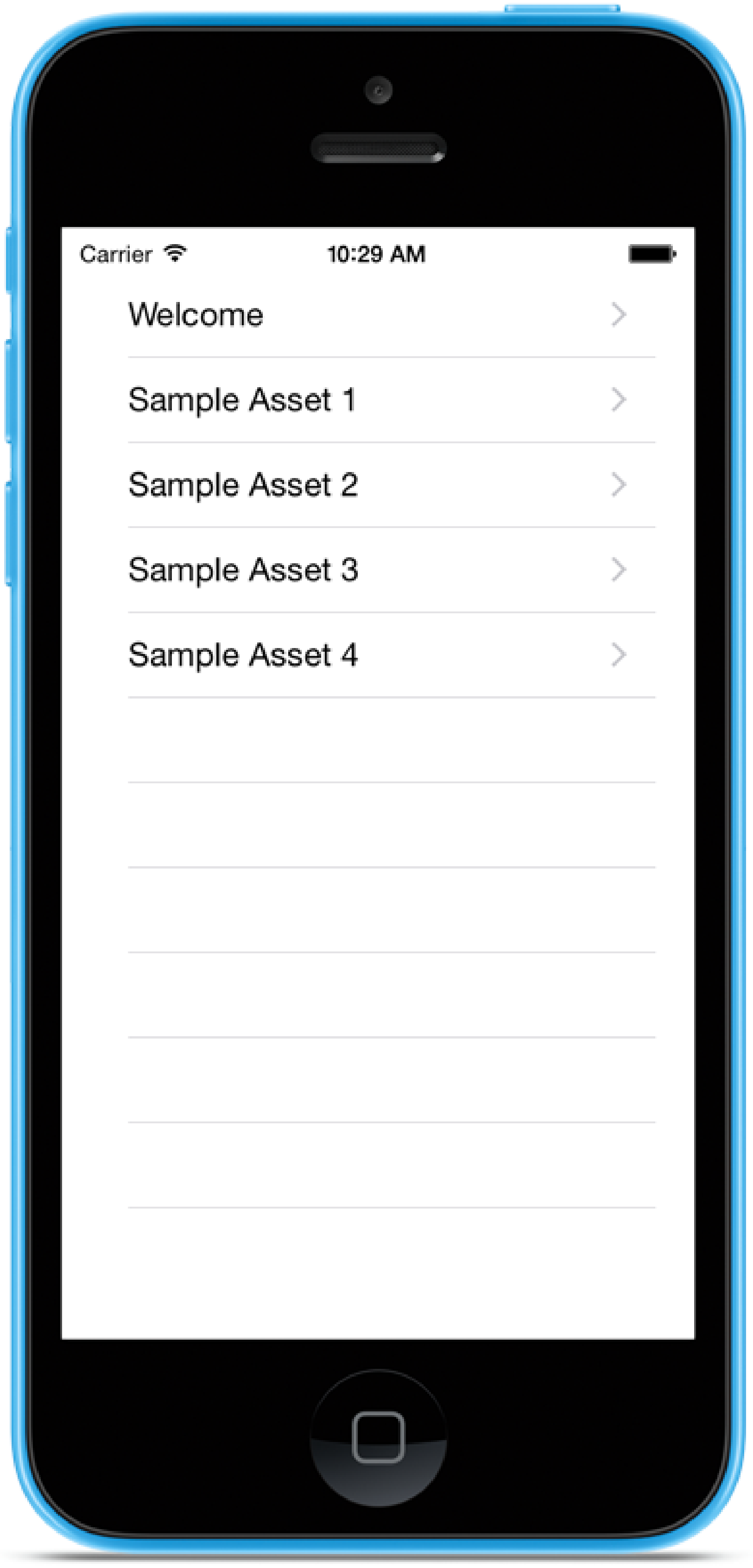


Figure 129.12: Asset List Screenlet using the Default (`default`) Theme.

Policy | What happens | When to use | `remote-only` | The Screenlet loads the list from the portal. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet successfully loads the list, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the list from the portal. If this succeeds, the Screenlet shows the list to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the list from the local cache. If the list doesn't exist there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet requests it from the portal and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `classNameId`

If you don't use `classNameId`, you must use this attribute:

- `portletItemName`

Attributes

Attribute | Data type | Explanation | `groupId` | `number` | The ID of the site (group) where the asset is stored. If set to `0`, the `groupId` specified in `LiferayServerContext` is used. The default value is `0`. | `classNameId` | `number` | The ID of the asset's class name. Use values from the `AssetClassNameId` class or the Liferay Instance's `classname_` database table. | `portletItemName` | `string` | The name of the configuration template you used in the Asset Publisher. To use this feature, add an Asset Publisher to one of your site's pages (it may be a hidden page), configure the Asset Publisher's filter (in *Configuration → Setup → Asset Selection*), and then use the Asset Publisher's *Configuration Templates* option to save this configuration with a name. Use this name as this attribute's value. | `offlinePolicy` | `string` | The offline mode setting. The default value is `remote-first`. See the Offline section for details. | `autoLoad` | `boolean` | Whether the list loads automatically when the Screenlet appears in the app's UI. The default value is `true`. | `refreshControl` | `boolean` | Defines whether a standard ios `UIRefreshControl` appears when the user does the pull to refresh gesture. The default value is `true`. | `firstPageSize` | `number` | The number of items retrieved from the server for display on the first page. The default value is `50`. | `pageSize` | `number` | The number of items retrieved from the server for display on the second and subsequent pages. The default value is `25`. | `customEntryQuery` | `Dictionary` | The set of keys (string) and values (string or number) to be used in the `AssetEntryQuery` object. These values filter the assets returned by the Liferay instance. |

Methods

Method | Return | Explanation | `loadList()` | `boolean` | Starts the request to load the list of assets. This list is shown when the response is received. Returns `true` if the request is sent. |

Delegate

The Asset List Screenlet delegates some events to an object that conforms to the `AssetListScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onAssetListResponse:`: Called when a page of assets is received. Note that this method may be called more than once; one call for each page received.
- `- screenlet:onAssetListError:`: Called when an error occurs in the process. The `NSError` object describes the error.
- `- screenlet:onAssetSelected:`: Called when an item in the list is selected.

129.8 Web Content Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

The Web Content Display Screenlet shows web content elements in your app, rendering the inner HTML of the web content. The Screenlet also supports i18n, rendering contents differently depending on the device's current locale.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`DDMStructureService`	`getStructureWithStructureId`	
`JournalArticleService`	`getArticleWithGroupId`	
`JournalArticleService`	`getArticleContent`	
`ScreensddlrecordService` (Screens compatibility plugin)	`getJournalArticleContent`	With `entryQuery`

Module

- WebContent

Themes

- Default

The Default Theme uses a standard `UIWebView` to render the HTML. Other Themes may use a different component, such as iOS 8's.

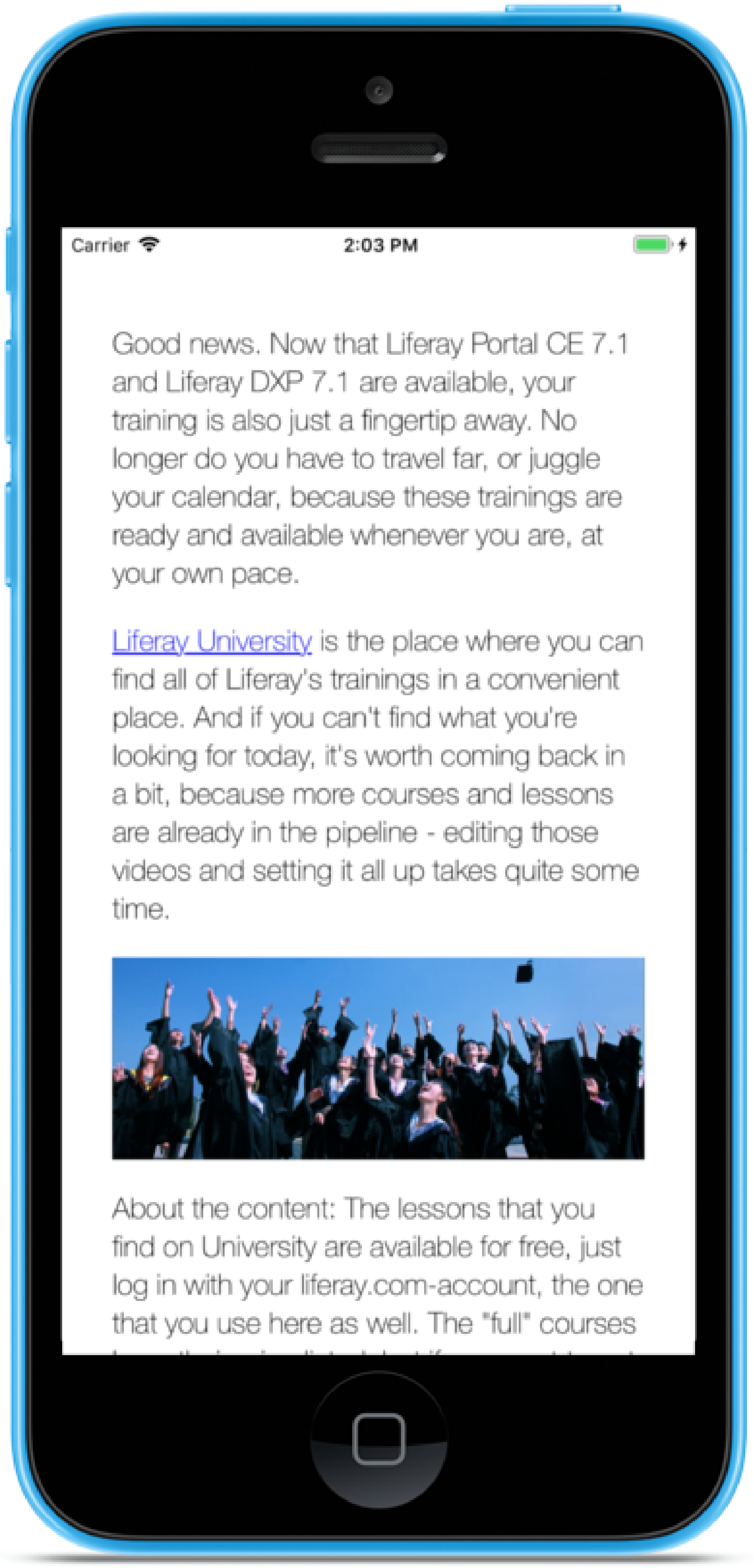
Portal Configuration

For the Web Content Display Screenlet to function properly, there should be web content in the Liferay instance your app connects to. For more details on web content, please refer to the Creating Web Content section of the Liferay User Guide.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the content from the portal. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet successfully loads the content, it stores the data in the local cache for later use. | Use this policy when you always need to show updated content, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the content from the local cache. If the content isn't there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy when you always need to show local content, without retrieving remote content under any circumstance. | `remote-first` | The Screenlet loads the content from the portal. If this succeeds, the Screenlet shows the content to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the content from the local cache. If the content doesn't exist there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy to show the most recent version of the content when connected, but show a possibly outdated version when there's no connection. | `cache-first` | The Screenlet loads the content from the local cache. If the content isn't there, the Screenlet requests it from the portal and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) content. |


Carrier 2:03 PM

Good news. Now that Liferay Portal CE 7.1 and Liferay DXP 7.1 are available, your training is also just a fingertip away. No longer do you have to travel far, or juggle your calendar, because these trainings are ready and available whenever you are, at your own pace.

Liferay University is the place where you can find all of Liferay's trainings in a convenient place. And if you can't find what you're looking for today, it's worth coming back in a bit, because more courses and lessons are already in the pipeline - editing those videos and setting it all up takes quite some time.

About the content: The lessons that you find on University are available for free, just log in with your liferay.com-account, the one that you use here as well. The "full" courses

Figure 129.13: The Web Content Display Screenlet using the Default (`default`) Theme

Required Attributes

- `articleId`

If you have structured web content, you can alternatively use `templateId` or `structureId` with `articleId`.

Attributes

Attribute | Data type | Explanation | `groupId` | `number` | The site (group) identifier where the asset is stored. If this value is `0`, the `groupId` specified in `LiferayServerContext` is used. | `articleId` | `string` | The identifier of the web content to display. You can find the identifier by clicking *Edit* on the web content in the portal. | `templateId` | `number` | The identifier of the template used to render the web content. This is applicable only with structured web content. | `structureId` | `number` | The identifier of the `DDMStructure` used to model the web content. This parameter lets the Screenlet retrieve and parse the structure. | `autoLoad` | `boolean` | Whether the content should be retrieved from the portal as soon as the Screenlet appears. The default value is `true`. |

Methods

Method | Return | Explanation | `loadWebContent()` | `boolean` | Starts the request to load the web content. The HTML is rendered when the response is received. Returns `true` if the request is sent. |

Delegate

The Web Content Display Screenlet delegates some events to an object that conforms to the `WebContentDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onWebContentResponse:`: Called when the web content's HTML is received.
- `- screenlet:onWebContentError:`: Called when an error occurs in the process. The `NSError` object describes the error.
- `- screenlet:onRecordContentResponse:`: Called when a web content record is received.
- `- screenlet:onUrlClicked:`: Called when a URL is clicked in the web content. Return `true` to handle the navigation, or `false` to cancel it.

129.9 Web Content List Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Web Content List Screenlet can show lists of web content from a Liferay instance. It can show both basic and structured web content. The Screenlet also implements fluent pagination with configurable page size, and supports i18n in asset values.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`JournalArticleService`	`getArticlesWithGroupId`	
`JournalArticleService`	`getArticlesCount`	

Module

- WebContent

Themes

- Default

The Default Theme uses a standard `UITableView` to show the scrollable list. Other Themes may use a different component, such as `UICollectionView` or others, to show the contents.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the list from the Liferay instance. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet successfully loads the list, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving

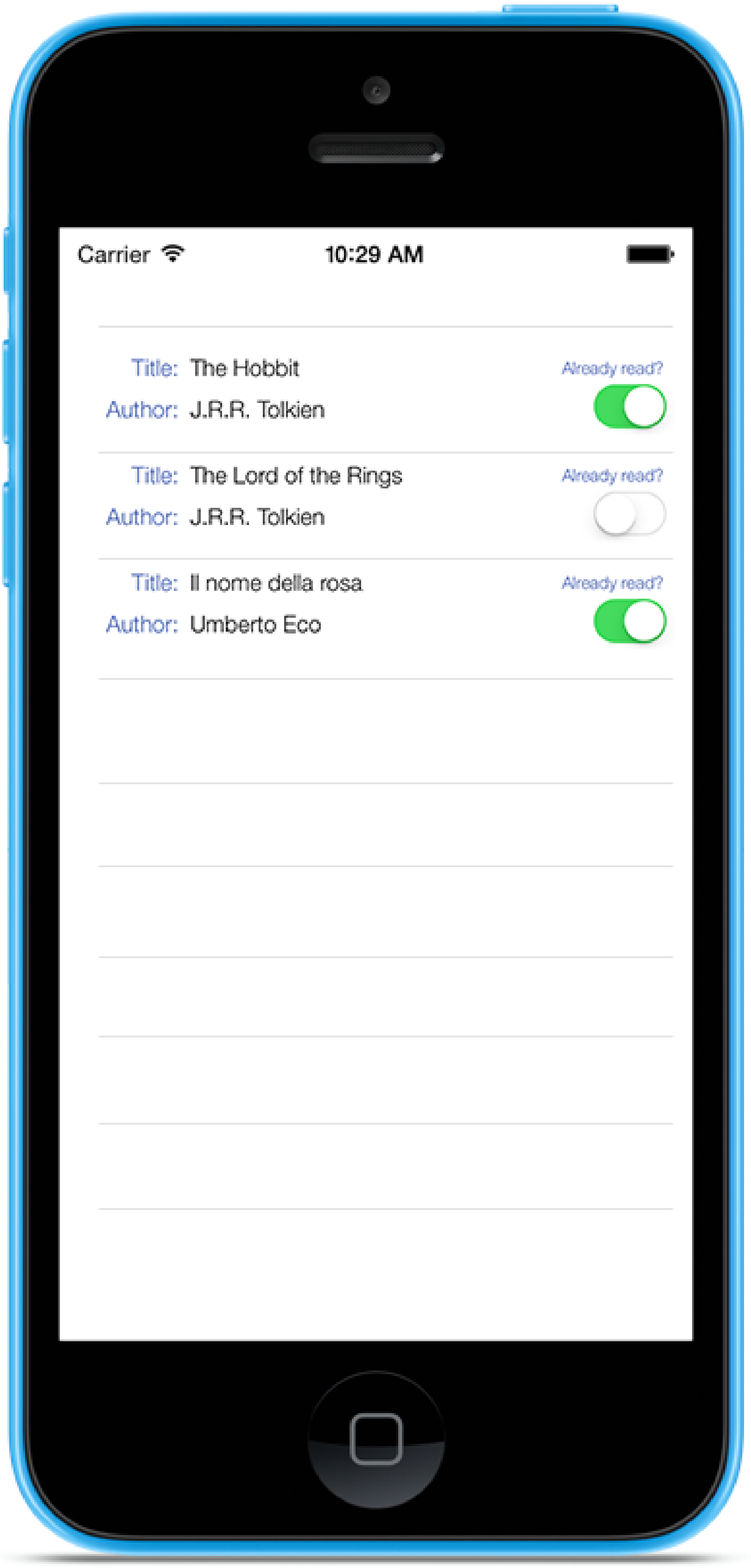


Figure 129.14: Web Content List Screenlet using the Default (`default`) Theme.

remote information under any circumstance. | `remote-first` | The Screenlet loads the list from the Liferay instance. If this succeeds, the Screenlet shows the list to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the list from the local cache. If the list doesn't exist there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show a possibly outdated version when there's no connection. | `cache-first` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but possibly outdated) data. |

Required Attributes

- `folderId`

Attributes

Attribute | Data type | Explanation | `groupId` | `number` | The ID of the site (group) where the web content exists. If set to `0`, the `groupId` specified in `LiferayServerContext` is used. The default value is `0`. | `folderId` | `number` | The ID of the web content folder. If set to `0`, the root folder is used. The default value is `0`. | `offlinePolicy` | `string` | The offline mode setting. The default value is `remote-first`. See the Offline section for details. | `autoLoad` | `boolean` | Whether the list loads automatically when the Screenlet appears in the app's UI. The default value is `true`. | `refreshControl` | `boolean` | Whether a standard iOS `UIRefreshControl` appears when the user does the pull to refresh gesture. The default value is `true`. | `firstPageSize` | `number` | The number of items to display on the first page. The default value is `50`. | `pageSize` | `number` | The number of items to display on the second and subsequent pages. The default value is `25`. | `obcClassName` | `string` | The name of the `OrderByComparator` class to use to sort the results. Omit this property if you don't want to sort the results. Click here to see some comparator classes. Note, however, that not all of these classes can be used with `obcClassName`. You can only use comparator classes that extend `OrderByComparator<JournalArticle>`. You can also create your own comparator classes that extend `OrderByComparator<JournalArticle>`. |

Methods

Method | Return | Explanation | `loadList()` | `boolean` | Starts the request to load the web content list. This list is shown when the response is received. Returns `true` if the request is sent successfully. |

Delegate

Web Content List Screenlet delegates some events to an object that conforms to the `WebContentListScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onWebContentListResponse:`: Called when a page of contents is received. Note that this method may be called more than once: one call for each page received.

- `- screenlet:onWebContentListError:`: Called when an error occurs in the process. The `NSError` object describes the error.

- - `screenlet:onWebContentSelected:`: Called when an item in the list is selected.

129.10 Image Gallery Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Image Gallery Screenlet shows a list of images from a Documents and Media folder in a Liferay instance. You can also use Image Gallery Screenlet to upload images to and delete images from the same folder. The Screenlet implements fluent pagination with configurable page size, and supports i18n in asset values.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`DLAppService`	`getFileEntries`	Load
`DLAppService`	`getFileEntriesCount`	
`DLAppService`	`addFileEntry`	Upload
`DLAppService`	`deleteFileEntry`	Delete

Module

- None

Themes

The default Theme uses a standard iOS `UICollectionView` to show the scrollable list as a grid. Other Themes may use a different component, such as `UITableView` or others, to show the contents.

This screenlet has three different Themes:

1. Grid (default)
2. Slideshow
3. List

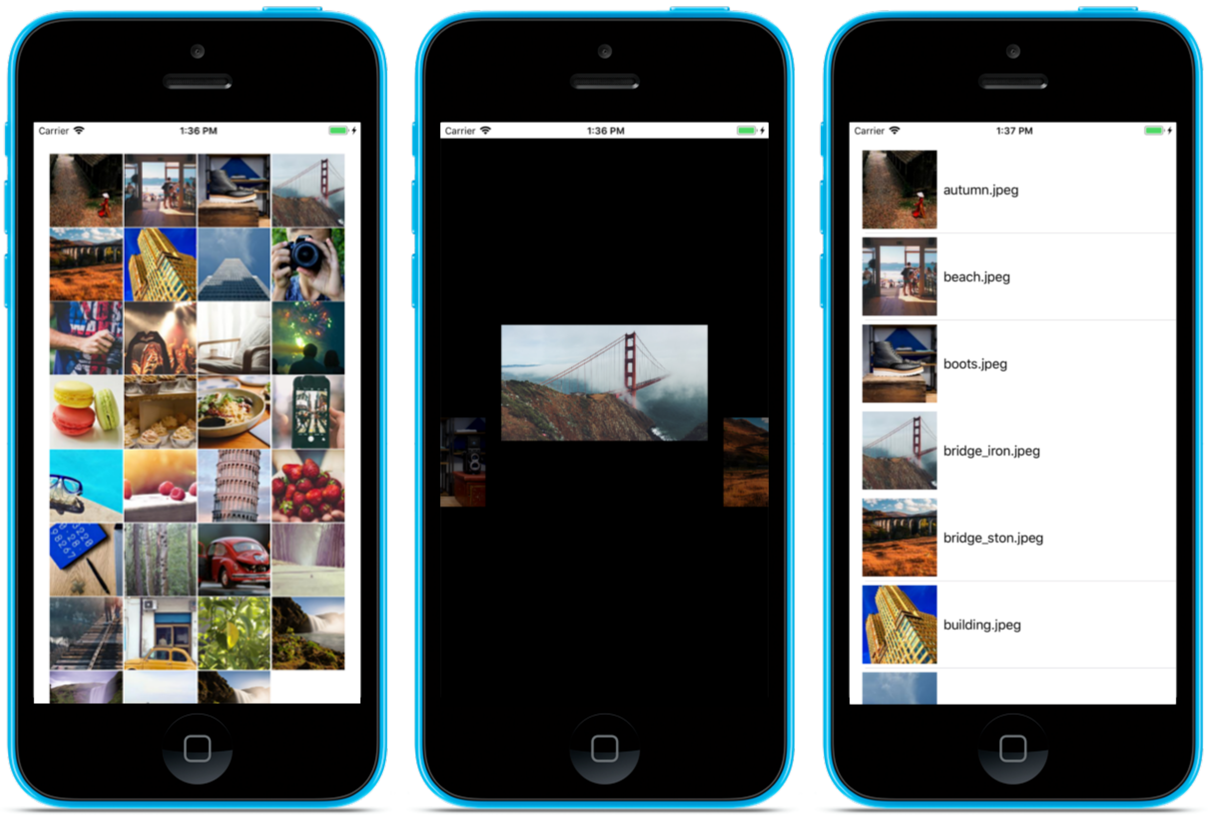


Figure 129.15: Image Gallery Screenlet using the Grid, Slideshow, and List Themes.

Offline

This Screenlet supports offline mode so it can function without a network connection when loading or uploading images (deleting images while offline is unsupported). For more information on how offline mode works, see the tutorial on its architecture. This Screenlet supports the `remote-only`, `cache-only`, `remote-first`, and `cache-first` offline mode policies.

These policies take the following actions when loading images from a Liferay instance:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show a possibly outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but possibly outdated) data. |

These policies take the following actions when uploading an image to a Liferay instance:

Policy | What happens | When to use | `remote-only` | The Screenlet sends the image to the Liferay instance. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error, but it also discards the image. | Use this policy to make sure the Liferay instance always has the most recent version of the image. | `cache-only` | The Screenlet stores the image in the local cache. | Use this policy when you need to save the image locally, but don't want to update it in the Liferay instance. | `remote-first` | The Screenlet sends the image to the Liferay instance. If this succeeds, it also stores the image in the local cache for later use. If a connection issue occurs, the Screenlet stores the image in the local cache and sends it to the Liferay instance when the connection is re-established. | Use this policy when you need to make sure the Screenlet sends the image to the Liferay instance as soon as the connection is restored. | `cache-first` | The Screenlet stores the image in the local cache and then attempts to send it to the Liferay instance. If a connection issue occurs, the Screenlet sends the image to the Liferay instance when the connection is re-established. | Use this policy when you need to make sure the Screenlet sends the image to the Liferay instance as soon as the connection is restored. Compared to `remote-first`, this policy always stores the image in the cache. The `remote-first` policy only stores the image in the event of a network error. |

Required Attributes

- `repositoryId`
- `folderId`

Attributes

Attribute | Data type | Explanation | `repositoryId` | `number` | The ID of the Liferay instance's Documents and Media repository that contains the image gallery. If you're using a site's default Documents and Media repository, then the `repositoryId` matches the site ID (`groupId`). | `folderId` | `number` | The ID of the Documents and Media repository folder that contains the image gallery. When accessing the folder in your browser, the `folderId` is at the end of the URL. | `mimeTypes` | `string` | The comma-separated list of MIME types for the Screenlet to support. | `filePrefix` | `string` | The prefix to use on uploaded image file names. | `offlinePolicy` | `string` | The offline mode setting. The default value is `remote-first`. See the Offline section for details. | `autoLoad` | `boolean` | Whether the list automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `refreshControl` | `boolean` | Whether a standard iOS `UIRefreshControl` appears when the user does the pull to refresh gesture. The default value is true. | firstPageSize | number | The number of items to display on the first page. The default value is 50. | pageSize | number | The number of items to display on the second and subsequent pages. The default value is 25. | obcClassName | string | The name of the OrderByComparator class to use to sort the results. Omit this property if you don't want to sort the results. Note that you can only use comparator classes that extend OrderByComparator<DLFileEntry>. Liferay contains no such comparator classes. You must therefore create your own by extending OrderByComparator<DLFileEntry>. To see examples of some comparator classes that extend other Document Library classes, click here. |

Methods

Method | Return | Explanation | loadList() | boolean | Starts the request to load the list of images. This list is shown when the response is received. Returns true if the request is sent successfully. |

Delegate

Image Gallery Screenlet delegates some events to an object that conforms to the ImageGalleryScreenletDelegate protocol. This protocol lets you implement the following methods:

- - screenlet:onImageEntriesResponse:: Called when a page of contents is received. Note that this method may be called more than once: one call for each page received.
- - screenlet:onImageEntriesError:: Called when an error occurs in the process. The NSError object describes the error.
- - screenlet:onImageEntrySelected:: Called when an item in the list is selected.
- - screenlet:onImageEntryDeleted:: Called when an image in the list is deleted.
- - screenlet:onImageEntryDeleteError:: Called when an error occurs during image file deletion. The NSError object describes the error.
- - screenlet:onImageUploadStart:: Called when an image is prepared for upload.
- - screenlet:onImageUploadProgress:: Called when the image upload progress changes.
- - screenlet:onImageUploadError:: Called when an error occurs in the image upload process. The NSError object describes the error.
- - screenlet:onImageUploaded:: Called when the image upload finishes.
- - screenlet:onImageUploadDetailViewCreated:: Called when the image upload View is instantiated. By default, the Screenlet uses a modal view controller to present this View. You only need to implement this method if you want to override this behavior. This method should present the View, passed as parameter, and then return true. For example, the following example implementation presents ImageUploadDetailViewBase as a parameter, and then uses it to customize the View's appearance:

```
func screenlet(screenlet: ImageGalleryScreenlet,
    onImageUploadDetailViewCreated uploadView: ImageUploadDetailViewBase) -> Bool {
        self.cardDeck?.cards[safe: 0]?.addPage(uploadView)
        self.cardDeck?.cards[safe: 0]?.moveRight()
        return true
}
```

129.11 Rating Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Rating Screenlet shows an asset's rating. It also lets users update or delete the rating. This Screenlet comes with different Themes that display ratings as thumbs, stars, and emojis.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensratingsentryService` (Screens compatibility plugin)	`getRatingsEntries`	With `entryId`
`ScreensratingsentryService` (Screens compatibility plugin)	`getRatingsEntries`	With `classPK` and `className`
`ScreensratingsentryService` (Screens compatibility plugin)	`updateRatingsEntry`	
`ScreensratingsentryService` (Screens compatibility plugin)	`deleteRatingsEntry`	

Module

- None

Themes

The default Theme uses the `CosmosView` library to show an asset's rating. Other custom Themes may use a different component, such as `UIButton` or others, to show the items.

This screenlet has four different Themes:

1. Like
2. Thumbs (default)
3. Stars
4. Emojis

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show a possibly outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but possibly outdated) data. |

Required Attributes

- `entryId`

If you don't use `entryId`, you must use these attributes:

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `layoutId` | `@layout` | The ID of the layout to use to show the Theme. | `autoLoad` | `boolean` | Whether the rating loads automatically when the Screenlet appears in the app's UI. The default value is `true`. | `editable` | `boolean` | Whether the user can change the rating. | `entryId` | `number` |

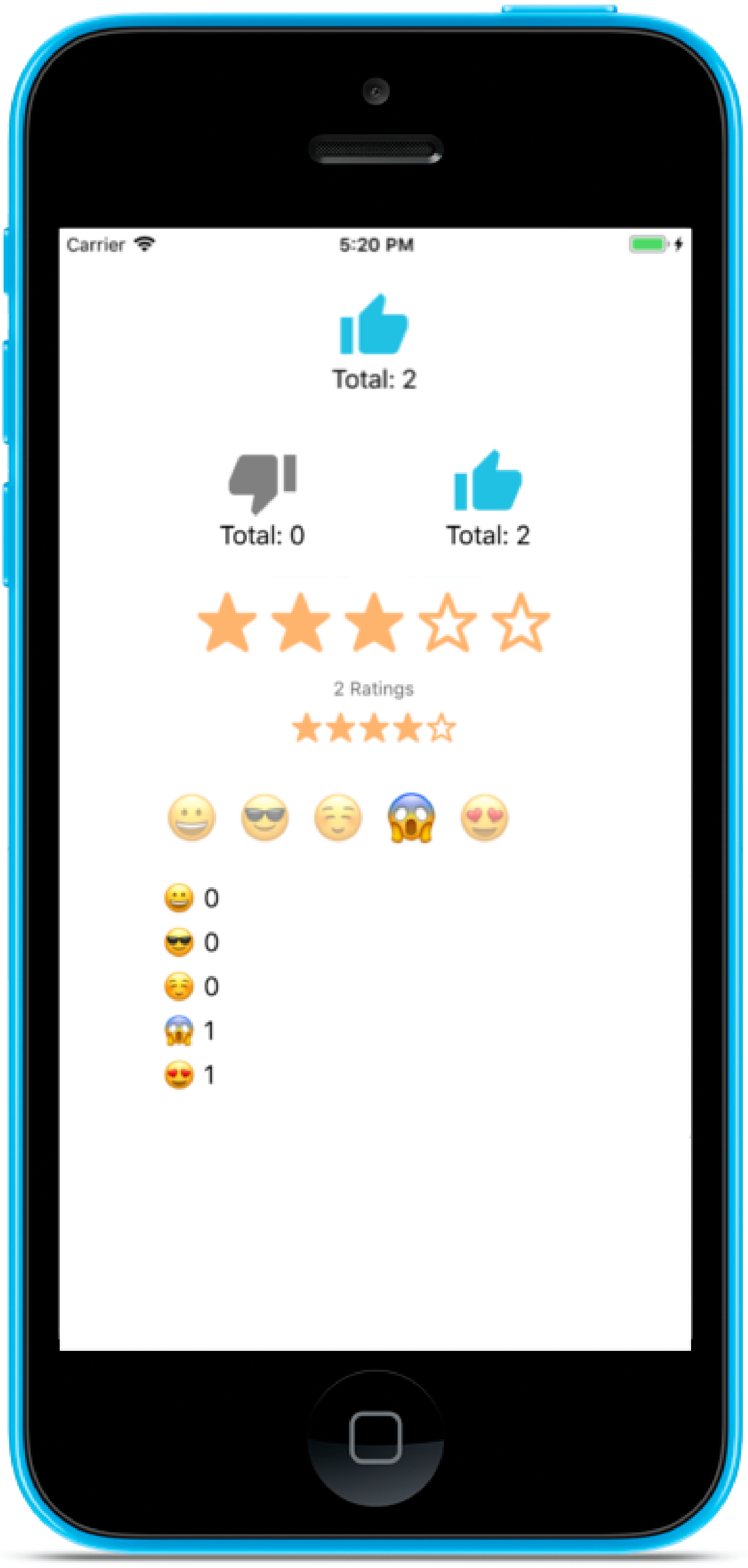


Figure 129.16: Rating Screenlet's different Themes.

The primary key of the asset with the rating to display. | `className` | `string` | The asset's fully qualified class name. For example, a blog entry's `className` is `com.liferay.blogs.kernel.model.BlogsEntry`. The `className` attribute is required when using it with `classPK` to instantiate the Screenlet.. | `classPK` | `number` | The asset's unique identifier. Only use this attribute when also using `className` to instantiate the Screenlet. | `groupId` | `number` | The ID of the site (group) containing the asset. | `offlinePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Methods

Method | Return | Explanation | `loadRatings()` | `boolean` | Starts the request to load the asset's ratings. |

Delegate

Rating Screenlet delegates some events to an object that conforms to the `RatingScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onRatingRetrieve:`: Called when the ratings are received.
- `- screenlet:onRatingDeleted:`: Called when a rating is deleted.
- `- screenlet:onRatingUpdated:`: Called when a rating is updated.
- `- screenlet:onRatingError:`: Called when an error occurs in the process. The `NSError` object describes the error.

129.12 Comment List Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Comment List Screenlet can list all the comments of an asset in a Liferay instance. It also lets the user update or delete comments.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreenscommentService` (Screens compatibility plugin)	`getCommentsWithClassName`	
`ScreenscommentService` (Screens compatibility plugin)	`getCommentsCount`	

Module

- None

Themes

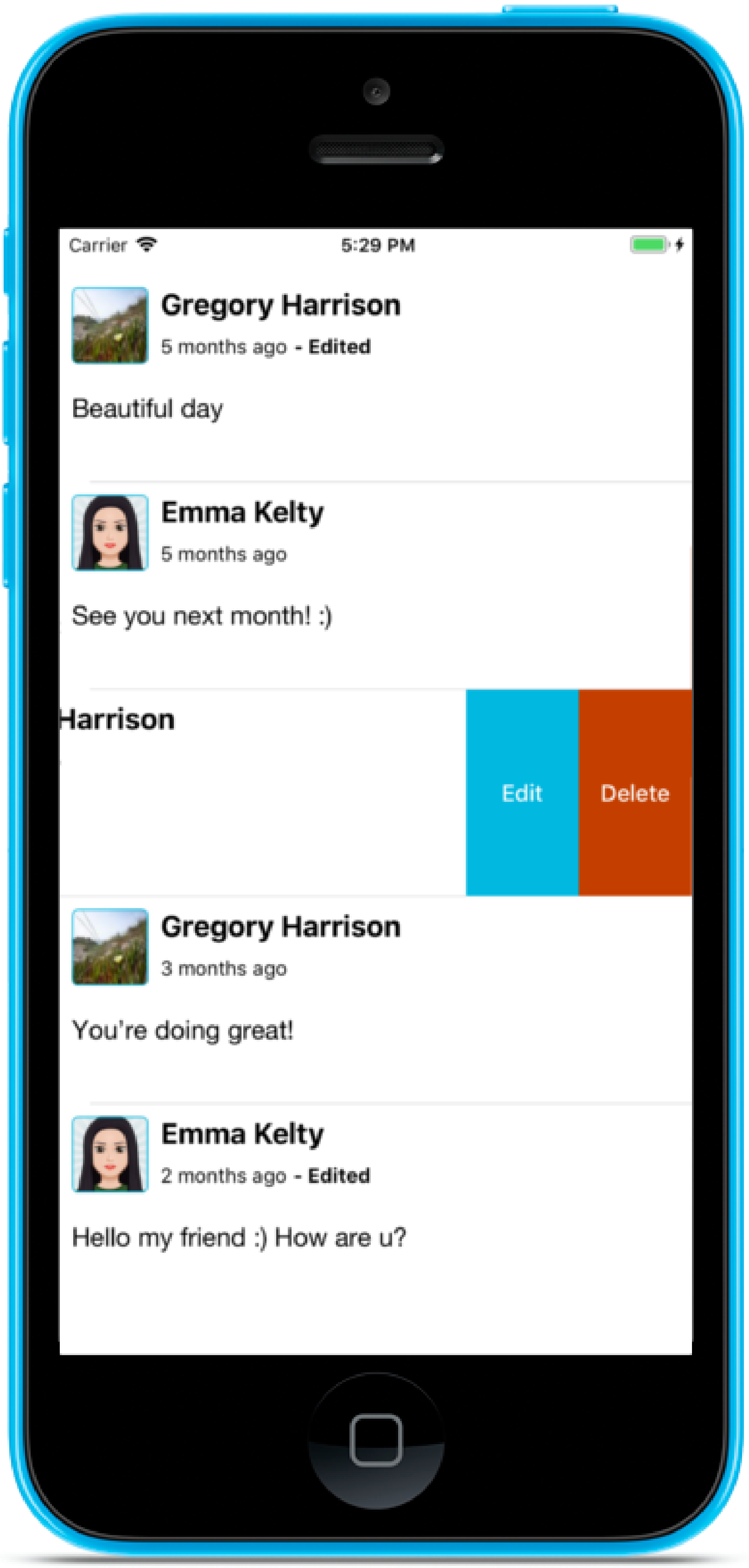
- Default

The Default Theme uses an iOS `UITableView` to show an asset's comments. Other Themes may use a different component, such as iOS's `UICollectionView` or others, to show the items.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the list from the Liferay instance. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error. If the Screenlet successfully loads the list, it stores the data in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the list from the Liferay instance. If this succeeds, the Screenlet shows the list to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the list from the local cache. If the list doesn't exist there, the Screenlet uses the delegate to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show a possibly outdated version when there's no connection. | `cache-first` | The Screenlet loads the list from the local cache. If the list isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but possibly outdated) data. |



Gregory Harrison
5 months ago - Edited
Beautiful day

Emma Kelty
5 months ago
See you next month! :)

Harrison

Gregory Harrison
3 months ago
You're doing great!

Emma Kelty
2 months ago - Edited
Hello my friend :) How are u?

Figure 129.17: Comment List Screenlet using the Default Theme.

Required Attributes

- `className`
- `classPK`

Attributes

Attribute	Data type	Explanation
`className`	`string`	The asset's fully qualified class name. For example, a blog entry's `className` is `com.liferay.blogs.kernel.model.BlogsEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet.
`classPK`	`number`	The asset's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet.
`offlinePolicy`	`string`	The offline mode setting. The default is `remote-first`. See the Offline section for details.
`editable`	`boolean`	Whether the user can edit the comment.
`autoLoad`	`boolean`	Whether the list should automatically load when the Screenlet appears in the app's UI. The default value is `true`.
`refreshControl`	`boolean`	Defines whether a standard iOS `UIRefreshControl` is shown when the user does the pull to refresh gesture. The default value is `true`.
`firstPageSize`	`number`	The number of items retrieved from the server for display on the first page. The default value is `50`.
`pageSize`	`number`	The number of items retrieved from the server for display on the second and subsequent pages. The default value is `25`.
`obcClassName`	`string`	The name of the `OrderByComparator` class to use to sort the results. You can only use classes that extend `OrderByComparator<MBMessage>`. If you don't want to sort the results, you can omit this property.

Methods

Method	Return	Explanation
`loadList()`	`boolean`	Starts the request to load the list. This list is shown when the response is received. Returns `true` if the request is sent.

Delegate

Comment List Screenlet delegates some events to an object that conforms to the `ComentListScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onListResponseComments:`: Called when the Screenlet receives the comments.
- `- screenlet:onCommentListError:`: Called when an error occurs in the process. The `NSError` object describes the error.
- `- screenlet:onSelectedComment:`: Called when a comment is selected.
- `- screenlet:onDeletedComment:`: Called when a comment is deleted.
- `- screenlet:onCommentDelete:`: Called when the Screenlet prepares a comment for deletion.
- `- screenlet:onUpdatedComment:`: Called when a comment is updated.
- `- screenlet:onCommentUpdate:`: Called when the Screenlet prepares a comment for update.

129.13 Comment Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Comment Display Screenlet can show one comment of an asset in a Liferay instance. It also lets the user update or delete the comment.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreenscommentService` (Screens compatibility plugin)	`getCommentWithCommentId`	
`ScreenscommentService` (Screens compatibility plugin)	`updateComment`	
`CommentmanagerjsonwsService`	`deleteComment`	

Module

- None

Themes

- Default

The Default Theme uses User Portrait Screenlet and iOS `UILabel` elements to show an asset's comment. Other Themes may use different components to show the comment.

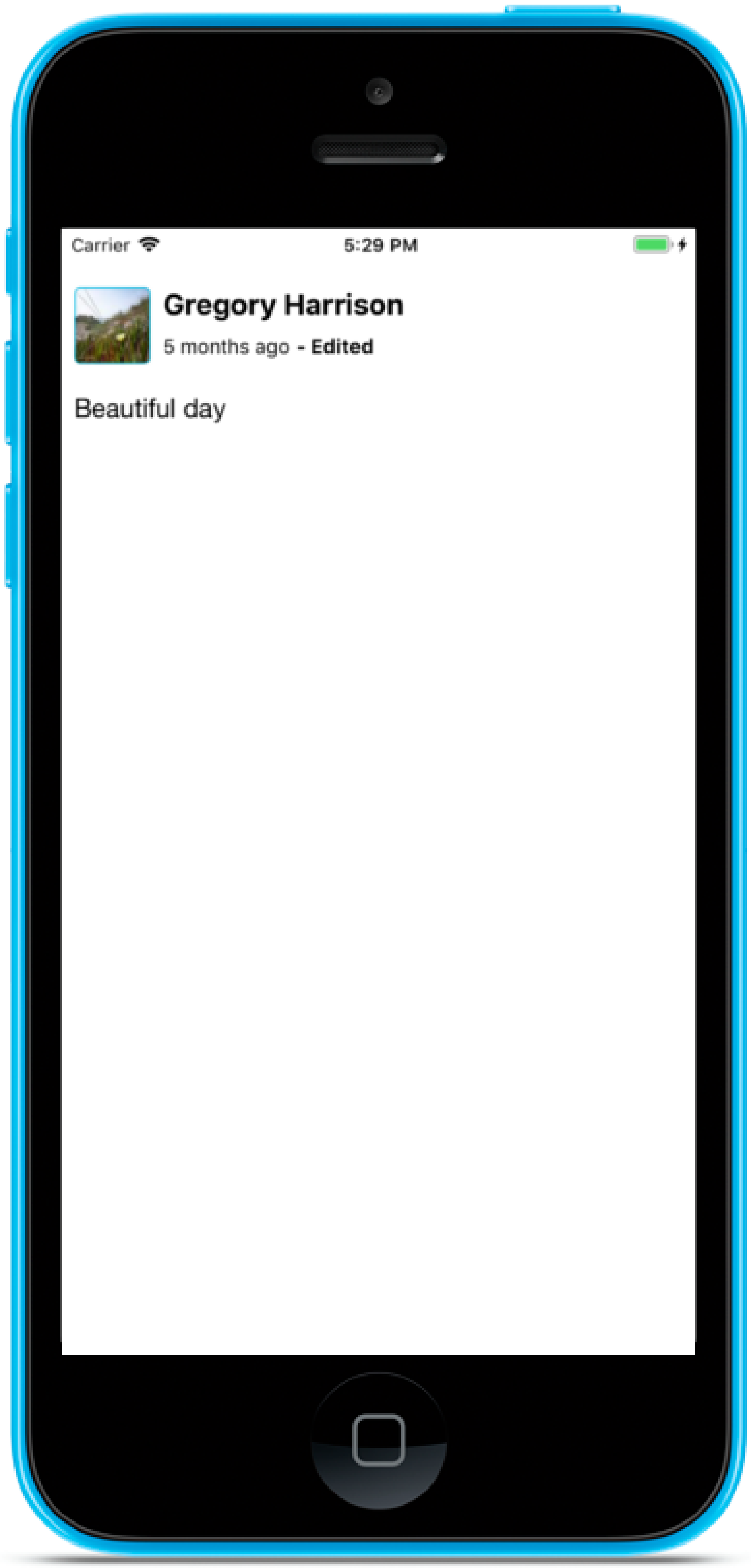


Figure 129.18: Comment Display Screenlet using the Default Theme.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. This Screenlet supports the `remote-only`, `cache-only`, `remote-first`, and `cache-first` offline mode policies.

These policies take the following actions when loading a comment from a Liferay instance:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

These policies take the following actions when updating or deleting a comment in a Liferay instance:

Policy | What happens | When to use | `remote-only` | The Screenlet sends the data to the Liferay instance. If a connection issue occurs, the Screenlet uses the delegate to notify the developer about the error, but it also discards the data. | Use this policy to make sure the Liferay instance always has the most recent version of the data. | `cache-only` | The Screenlet stores the data in the local cache. | Use this policy when you need to save the data locally, but don't want to update it in the Liferay instance. | `remote-first` | The Screenlet sends the data to the Liferay instance. If this succeeds, it also stores the data in the local cache for later use. If a connection issue occurs, the Screenlet stores the data in the local cache and sends it to the Liferay instance when the connection is re-established. | Use this policy when you need to make sure the Screenlet sends the data to the Liferay instance as soon as the connection is restored. | `cache-first` | The Screenlet stores the data in the local cache and then attempts to send it to the Liferay instance. If a connection issue occurs, the Screenlet sends the data to the Liferay instance when the connection is re-established. | Use this policy when you need to make sure the Screenlet sends the data to the Liferay instance as soon as the connection is restored. Compared to `remote-first`, this policy always stores the data in the cache. The `remote-first` policy only stores the data in the event of a network error. |

Required Attributes

- `commentId`

Attributes

Attribute | Data type | Explanation | `commentId` | `number` | The primary key of the comment to display. | `autoLoad` | `boolean` | Whether the list should automatically load when the Screenlet appears in the app's UI. The default value is `true`. | `editable` | `boolean` | Whether the user can edit the comment. | `offlinePolicy` | `string` | The offline mode setting. The default is `remote-first`. See the Offline section for details. |

Methods

Method | Return | Explanation | `load()` | none | Starts the request to load the comment. |

Delegate

Comment Display Screenlet delegates some events to an object that conforms to the `CommentDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onCommentLoaded:`: Called when the Screenlet loads the comment.
- `- screenlet:onLoadCommentError:`: Called when an error occurs in the process. The `NSError` object describes the error.
- `- screenlet:onSelectedComment:`: Called when a comment is selected.
- `- screenlet:onDeletedComment:`: Called when a comment is deleted.
- `- screenlet:onCommentDelete:`: Called when the Screenlet prepares the comment for deletion.
- `- screenlet:onUpdatedComment:`: Called when a comment is updated.
- `- screenlet:onCommentUpdate:`: Called when the Screenlet prepares the comment for update.

129.14 Comment Add Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Comment Add Screenlet can add a comment to an asset in a Liferay instance.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreenscommentService` (Screens compatibility plugin)	`addComment`	

Module

- None

Themes

- Default

The Default Theme uses the iOS elements `UITextField` and `UIButton` to add a comment to an asset. Other Themes may use other components to show the comment.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet sends the data to the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully sends the data, it also stores it in the local cache. | Use this policy when you always need to send updated data, and send nothing when there's no connection. | `cache-only` | The Screenlet sends the data to the local cache. If an error occurs, the Screenlet uses the listener to notify the developer. | Use this policy when you always need to store local data without sending remote information under any circumstance. | `remote-first` | The Screenlet sends the data to the Liferay instance. If this succeeds, the Screenlet also stores the data in the local cache. If a connection issue occurs, the Screenlet stores the data to the local cache and sends it to the Liferay instance when the connection is restored. If an error occurs, the Screenlet uses the listener to notify the developer. | Use this policy to send the most recent version of the data when connected, and store the data for later synchronization when there's no connection. | `cache-first` | The Screenlet sends the data to the local cache, then sends it to the Liferay instance. If sending the data to the Liferay instance fails, the Screenlet still stores the data locally and then notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and store local (but possibly outdated) data. |

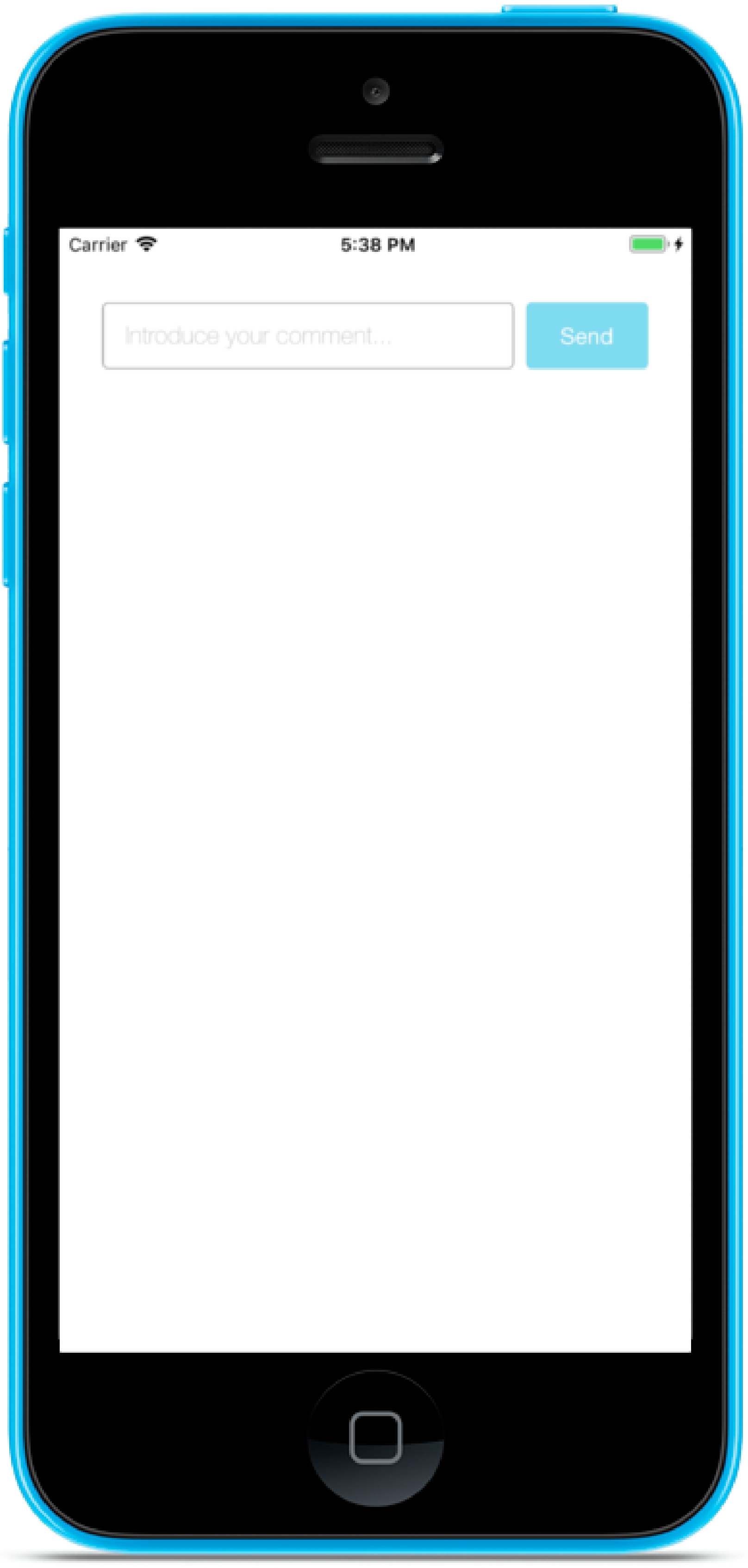


Figure 129.19: Comment Add Screenlet using the Default Theme.

Required Attributes

- `className`
- `classPK`

Attributes

Attribute	Data type	Explanation
`className`	`string`	The asset's fully qualified class name. For example, a blog entry's `className` is `com.liferay.blogs.kernel.model.BlogsEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet.
`classPK`	`number`	The asset's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet.
`offlinePolicy`	`string`	The offline mode setting. The default value is `remote-first`. See the Offline section for details.

Delegate

Comment Add Screenlet delegates some events to an object that conforms to the `CommentAddScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onCommentAdded:`: Called when the Screenlet adds a comment.

- `- screenlet:onAddCommentError:`: Called when an error occurs while adding a comment. The `NSError` object describes the error.

- `- screenlet:onCommentUpdated:`: Called when the Screenlet prepares a comment for update.

- `- screenlet:onUpdateCommentError:`: Called when an error occurs while updating a comment. The `NSError` object describes the error.

129.15 Asset Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Asset Display Screenlet can display an asset from a Liferay instance. The Screenlet can currently display Documents and Media files (`DLFileEntry` images, videos, audio files, and PDFs), blogs entries (`BlogsEntry`) and web content articles (`WebContent`).

Asset Display Screenlet can also display your custom asset types. See the delegate section of this document for details.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Themes

- Default

The Default Theme uses different UI elements to show each asset type. For example, it displays images with `UIImageView`, and blogs with `UILabel`.

This Screenlet can also render other Screenlets:

- Images: Image Display Screenlet
- Videos: Video Display Screenlet
- Audio: Audio Display Screenlet
- PDFs: PDF Display Screenlet
- Blog entries: Blogs Entry Display Screenlet
- Web content: Web Content Display Screenlet

These Screenlets can also be used alone without Asset Display Screenlet.

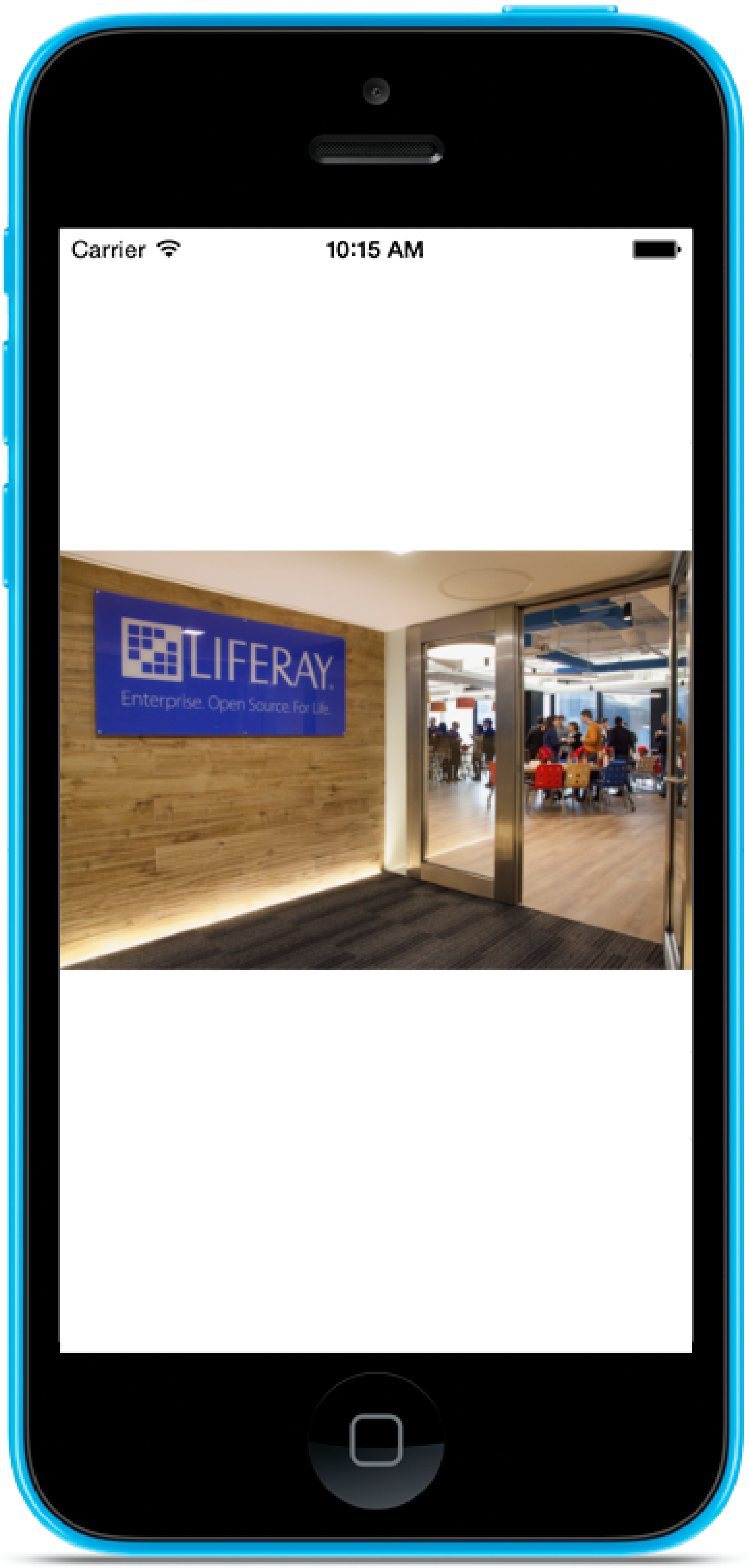


Figure 129.20: Asset Display Screenlet using the Default Theme.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `assetEntryId`

Instead of `assetEntryId`, you can use both of these attributes:

- `className`
- `classPK`

If you don't use the above attributes, you must use this attribute:

- `portletItemName`

Attributes

Attribute | Data type | Explanation | `assetEntryId` | `number` | The primary key of the asset. | `className` | `string` | The asset's fully qualified class name. For example, a blog entry's `className` is `com.liferay.blogs.kernel.model.BlogsEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The asset's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `portletItemName` | `string` | The name of the configuration template you used in the Asset Publisher. To use this feature, add an Asset Publisher to one of your site's pages (it may be a hidden page), configure the Asset Publisher's filter (in *Configuration → Setup → Asset Selection*), and then use the Asset Publisher's *Configuration Templates* option to save this configuration with a name. Use this name as this attribute's value. If there is more than one asset in the configuration, the Screenlet displays only the first one. | `assetEntry` | `Asset` | The `Asset` object to display, selected from a list of assets. Note that if you use this attribute, the Screenlet doesn't need to call the server. | `autoLoad` | `boolean` | Whether the asset automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `offlinePolicy` | `string` | The offline mode setting. The default value is `remote-first`. See the Offline section for details. |

Delegate

Asset Display Screenlet delegates some events to an object that conforms to the `AssetDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onAssetResponse:`: Called when the Screenlet receives the asset.
- `- screenlet:onAssetError:`: Called when an error occurs in the process. The `NSError` object describes the error.
- `- screenlet:onConfigureScreenlet:`: Called when the child Screenlet (the Screenlet rendered inside Asset Display Screenlet) has been successfully initialized. Use this method to configure or customize it. The example implementation here sets the child Blogs Entry Display Screenlet's background color to gray:

```
func screenlet(screenlet: AssetDisplayScreenlet, onConfigureScreenlet,
    childScreenlet: BaseScreenlet?, onAsset asset: Asset) {
        if childScreenlet is BlogsEntryDisplayScreenlet {
            childScreenlet?.screenletView?.backgroundColor = UIColor.grayColor()
        }
}
```

- `- screenlet:onAsset:`: Called to render a custom asset. The following example implementation renders a portal user (`User`). If the asset is a user, this method instantiates a custom `UserProfileView` to render that user:

```
public func screenlet(screenlet: AssetDisplayScreenlet, onAsset asset: Asset) -> UIView? {
    if let type = asset.attributes["object"]?.allKeys.first as? String {
        if type == "user" {
            let view = NSBundle.mainBundle().loadNibNamed("UserProfileView", owner: self,
                options: nil)![0] as? UserProfileView

            view?.user = User(attributes: asset.attributes)

            return view
        }
    }
    return nil
}
```

129.16 Blogs Entry Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Blogs Entry Display Screenlet displays a single blog entry. Image Display Screenlet renders any header image the blogs entry may have.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

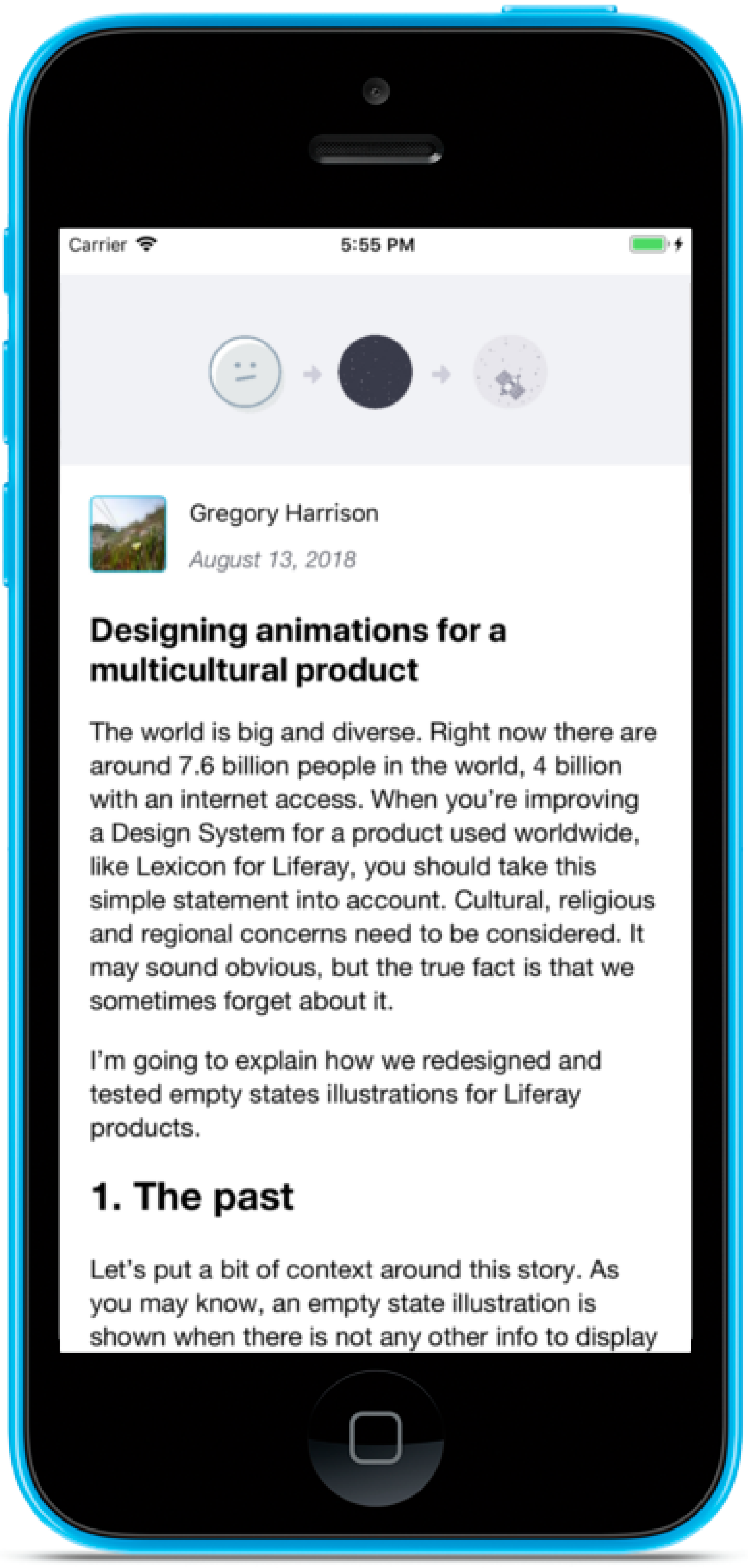
Themes

- Default

The Default Theme can use different components to show a blogs entry (`BlogsEntry`). For example, it uses `UILabel` to show a blog's text, and User Portrait Screenlet to show the profile picture of the Liferay user who posted it. Note that other Themes may use different components.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:



Carrier 5:55 PM

Gregory Harrison

August 13, 2018

Designing animations for a multicultural product

The world is big and diverse. Right now there are around 7.6 billion people in the world, 4 billion with an internet access. When you're improving a Design System for a product used worldwide, like Lexicon for Liferay, you should take this simple statement into account. Cultural, religious and regional concerns need to be considered. It may sound obvious, but the true fact is that we sometimes forget about it.

I'm going to explain how we redesigned and tested empty states illustrations for Liferay products.

1. The past

Let's put a bit of context around this story. As you may know, an empty state illustration is shown when there is not any other info to display

Figure 129.21: Blogs Entry Display Screenlet using the Default Theme.

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote data under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `assetEntryId` or `classPK`

Attributes

Attribute | Data type | Explanation | `assetEntryId` | `number` | The primary key of the blog entry (`BlogsEntry`). | `classPK` | `number` | The `BlogsEntry` object's unique identifier. | `autoLoad` | `boolean` | Whether the blog entry automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `offlinePolicy` | `string` | The offline mode setting. The default value is `remote-first`. See the Offline section for details. |

Delegate

Blogs Entry Display Screenlet delegates some events to an object that conforms to the `BlogsEntryDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onBlogEntryResponse:`: Called when the Screenlet receives the `BlogsEntry` object.

- `- screenlet:onBlogEntryError:`: Called when an error occurs in the process. The `NSError` object describes the error.

129.17 Image Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Image Display Screenlet displays an image file from a Liferay instance's Documents and Media Library.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Themes

- Default

The Default Theme uses an iOS `UIImageView` for displaying the image.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when

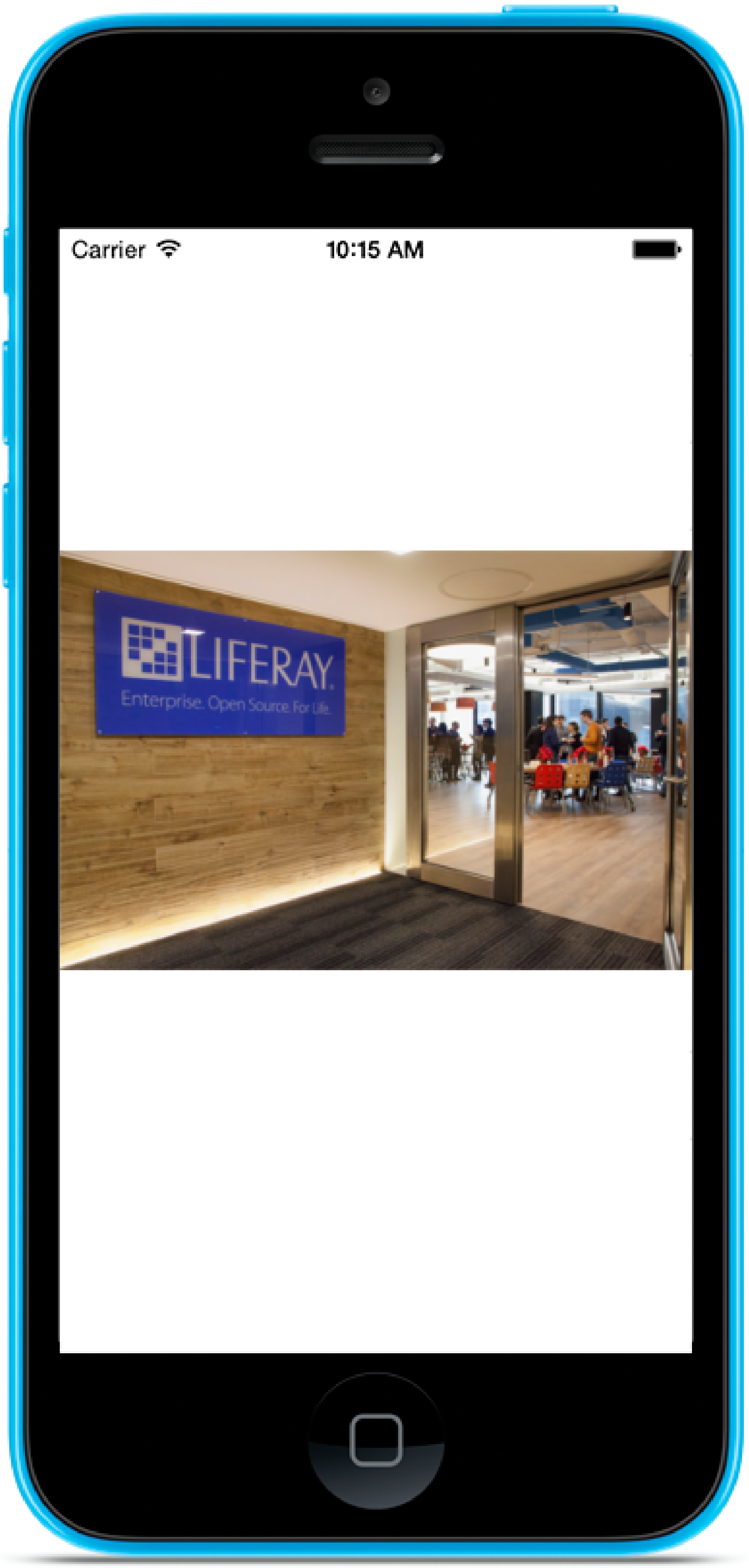


Figure 129.22: Image Display Screenlet using the Default Theme.

you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `assetEntryId`

If you don't use `assetEntryId`, you must use these attributes:

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `assetEntryId` | `number` | The primary key of the image. | `className` | `string` | The image's fully qualified class name. Since files in a Documents and Media Library are `DLFileEntry` objects, their `className` is `com.liferay.document.library.kernel.model.DLFileEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The image's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `autoLoad` | `boolean` | Whether the image automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `offlinePolicy` | `string` | The offline mode setting. The default value is `remote-first`. See the Offline section for details. |

Delegate

Because images are files, Image Display Screenlet delegates its events to an object that conforms to the `FileDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onFileAssetResponse:`: Called when the Screenlet receives the image file.

- `- screenlet:onFileAssetError:`: Called when an error occurs in the process. The `NSError` object describes the error.

129.18 Video Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Video Display Screenlet displays a video file from a Liferay instance's Documents and Media Library.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Themes

- Default

The Default Theme uses an iOS `AVPlayerViewController` to display the video.

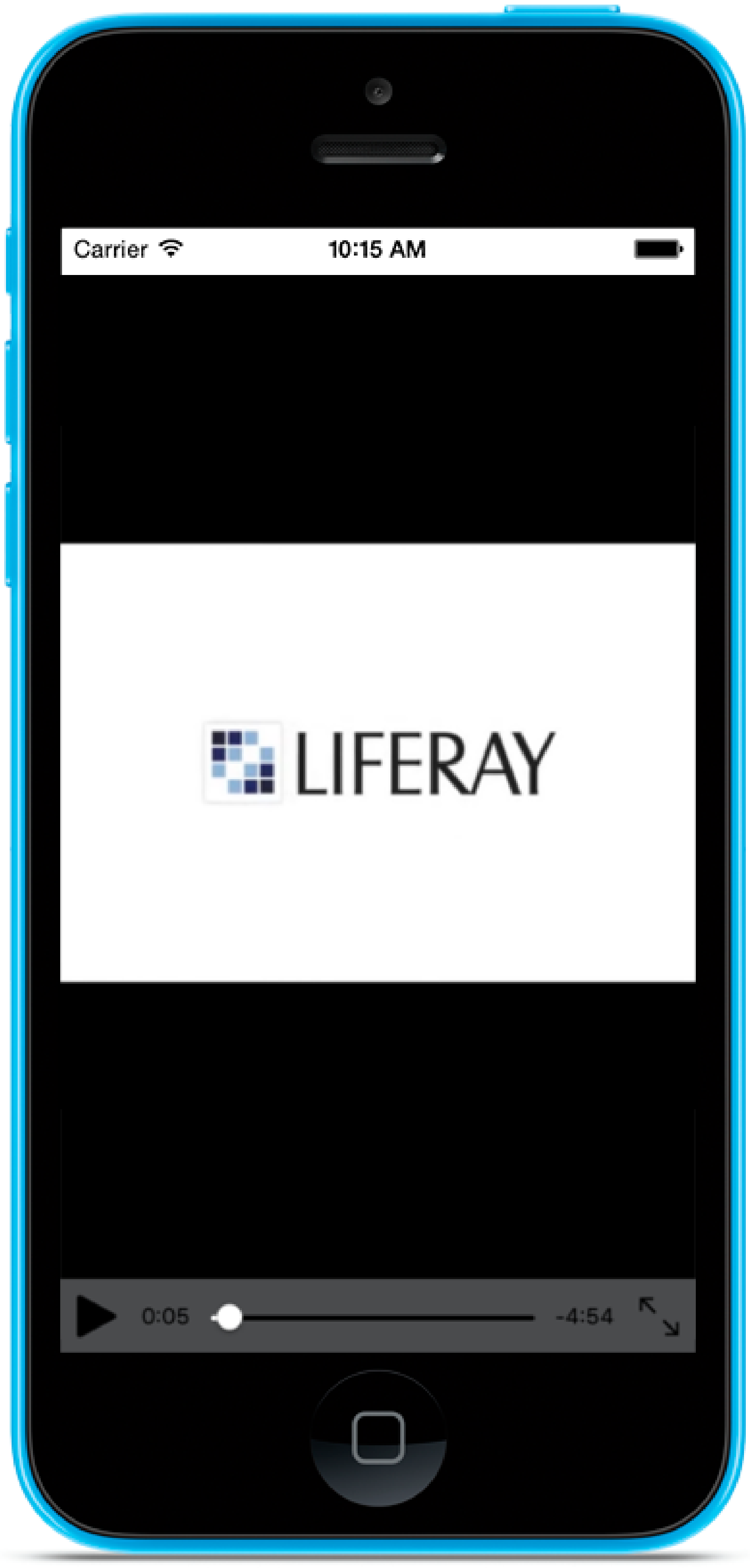


Figure 129.23: Video Display Screenlet using the Default Theme.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `assetEntryId`

If you don't use `assetEntryId`, you must use these attributes:

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `assetEntryId` | `number` | The primary key of the video file. | `className` | `string` | The video file's fully qualified class name. Since files in a Documents and Media Library are `DLFileEntry` objects, the `className` is `com.liferay.document.library.kernel.model.DLFileEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The video file's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `autoLoad` | `boolean` | Whether the video automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `offlinePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Delegate

Because images are files, Video Display Screenlet delegates its events to an object that conforms to the `FileDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onFileAssetResponse:`: Called when the Screenlet receives the image file.

- `- screenlet:onFileAssetError:`: Called when an error occurs in the process. The `NSError` object describes the error.

129.19 Audio Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Audio Display Screenlet displays an audio file from a Liferay instance's Documents and Media Library.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Themes

- Default

The Default Theme uses an iOS `AVAudioPlayer` to display the audio player. For the player components, this Theme uses `UIButton`, `UISlider`, and several `UILabel` instances.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `assetEntryId`

If you don't use `assetEntryId`, you must use these attributes:

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `assetEntryId` | `number` | The primary key of the audio file. | `className` | `string` | The audio file's fully qualified class name. Since files in a Documents and Media Library are `DLFileEntry` objects, their `className` is `com.liferay.document.library.kernel.model.DLFileEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The audio file's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `autoLoad` | `boolean` | Whether the audio file automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `offlinePolicy` | `string` | The offline mode setting. See the Offline section for details. |

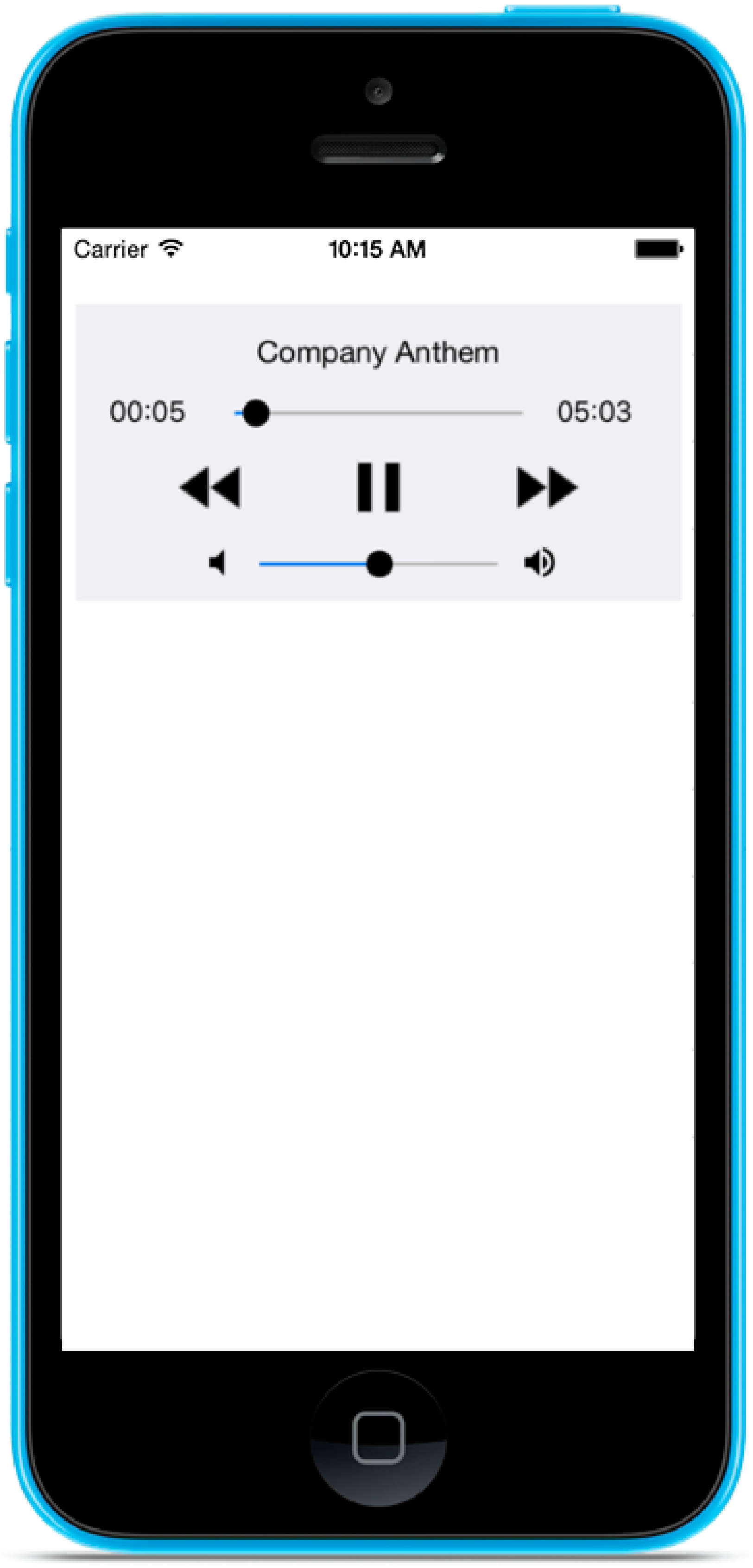


Figure 129.24: Audio Display Screenlet using the Default Theme.

Delegate

Audio Display Screenlet delegates its events to an object that conforms to the `FileDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onFileAssetResponse:`: Called when the Screenlet receives the audio file.
- `- screenlet:onFileAssetError:`: Called when an error occurs in the process. An `NSError` object describes the error.

129.20 PDF Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

PDF Display Screenlet displays a PDF file from a Liferay Instance's Documents and Media Library.

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Themes

- Default

The Default Theme uses an iOS `UIWebView` for displaying the PDF file.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Required Attributes

- `assetEntryId`

If you don't use `assetEntryId`, you must use these attributes:

- `className`
- `classPK`

Attributes

Attribute | Data type | Explanation | `assetEntryId` | `number` | The primary key of the PDF file. | `className` | `string` | The PDF file's fully qualified class name. Since files in a Documents and Media Library are `DLFileEntry` objects, their `className` is `com.liferay.document.library.kernel.model.DLFileEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The PDF file's unique identifier. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `autoLoad` |

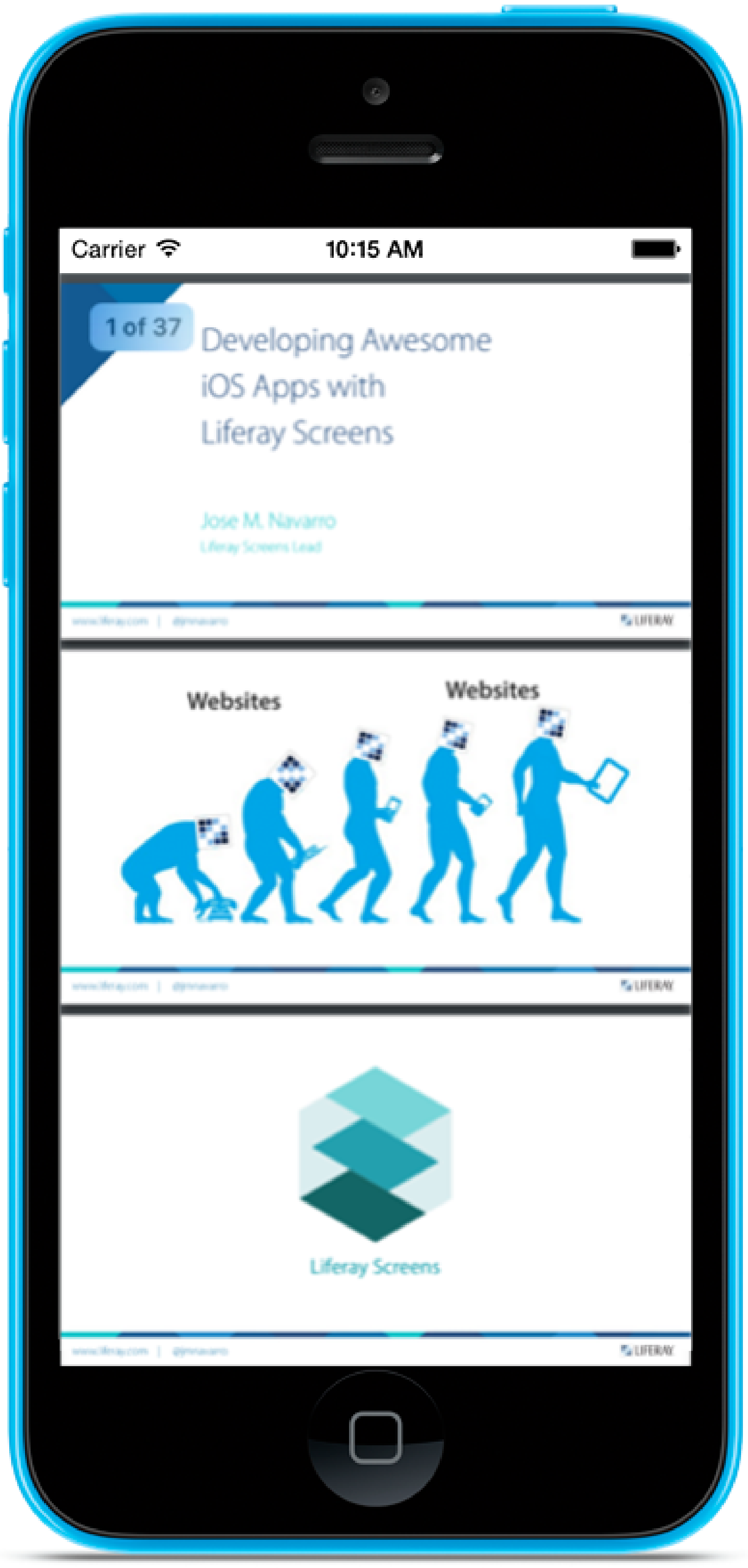


Figure 129.25: PDF Display Screenlet using the Default Theme.

`boolean` | Whether the PDF automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `offlinePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Delegate

Because PDFs are files, PDF Display Screenlet delegates some events to an object that conforms to the `FileDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onFileAssetResponse:`: Called when the Screenlet receives the PDF.
- `- screenlet:onFileAssetError:`: Called when an error occurs in the process. An `NSError` object describes the error.

129.21 File Display Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

File Display Screenlet shows a single file from a Liferay DXP instance's Documents and Media Library. Use this Screenlet to display file types not covered by the other display Screenlets (e.g., DOC, PPT, XLS).

JSON Services Used

Screenlets in Liferay Screens call JSON web services in the portal. This Screenlet calls the following services and methods.

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `entryId`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntry`	With `classPK` and `className`

Service	Method	Notes
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `entryQuery`
`ScreensassetentryService` (Screens compatibility plugin)	`getAssetEntries`	With `companyId`, `groupId`, and `portletItemName`

Module

- None

Themes

- Default

The Default View uses an iOS `UIWebView` for displaying the file.

Offline

This Screenlet supports offline mode so it can function without a network connection. For more information on how offline mode works, see the tutorial on its architecture. Here are the offline mode policies that you can use with this Screenlet:

Policy | What happens | When to use | `remote-only` | The Screenlet loads the data from the Liferay instance. If a connection issue occurs, the Screenlet uses the listener to notify the developer about the error. If the Screenlet successfully loads the data, it stores it in the local cache for later use. | Use this policy when you always need to show updated data, and show nothing when there's no connection. | `cache-only` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet uses the listener to notify the developer about the error. | Use this policy when you always need to show local data, without retrieving remote information under any circumstance. | `remote-first` | The Screenlet loads the data from the Liferay instance. If this succeeds, the Screenlet shows the data to the user and stores it in the local cache for later use. If a connection issue occurs, the Screenlet retrieves the data from the local cache. If the data doesn't exist there, the Screenlet uses the listener to notify the developer about the error. | Use this policy to show the most recent version of the data when connected, but show an outdated version when there's no connection. | `cache-first` | The Screenlet loads the data from the local cache. If the data isn't there, the Screenlet requests it from the Liferay instance and notifies the developer about any errors that occur (including connectivity errors). | Use this policy to save bandwidth and loading time in case you have local (but probably outdated) data. |

Attributes

Attribute | Data type | Explanation | `assetEntryId` | `number` | The primary key of the file. | `className` | `string` | The file's fully qualified class name. Since files in a Documents and Media Library are `DLFileEntry` objects, their `className` is `com.liferay.document.library.kernel.model.DLFileEntry`. The `className` and `classPK` attributes are required to instantiate the Screenlet. | `classPK` | `number` | The file's unique identifier.

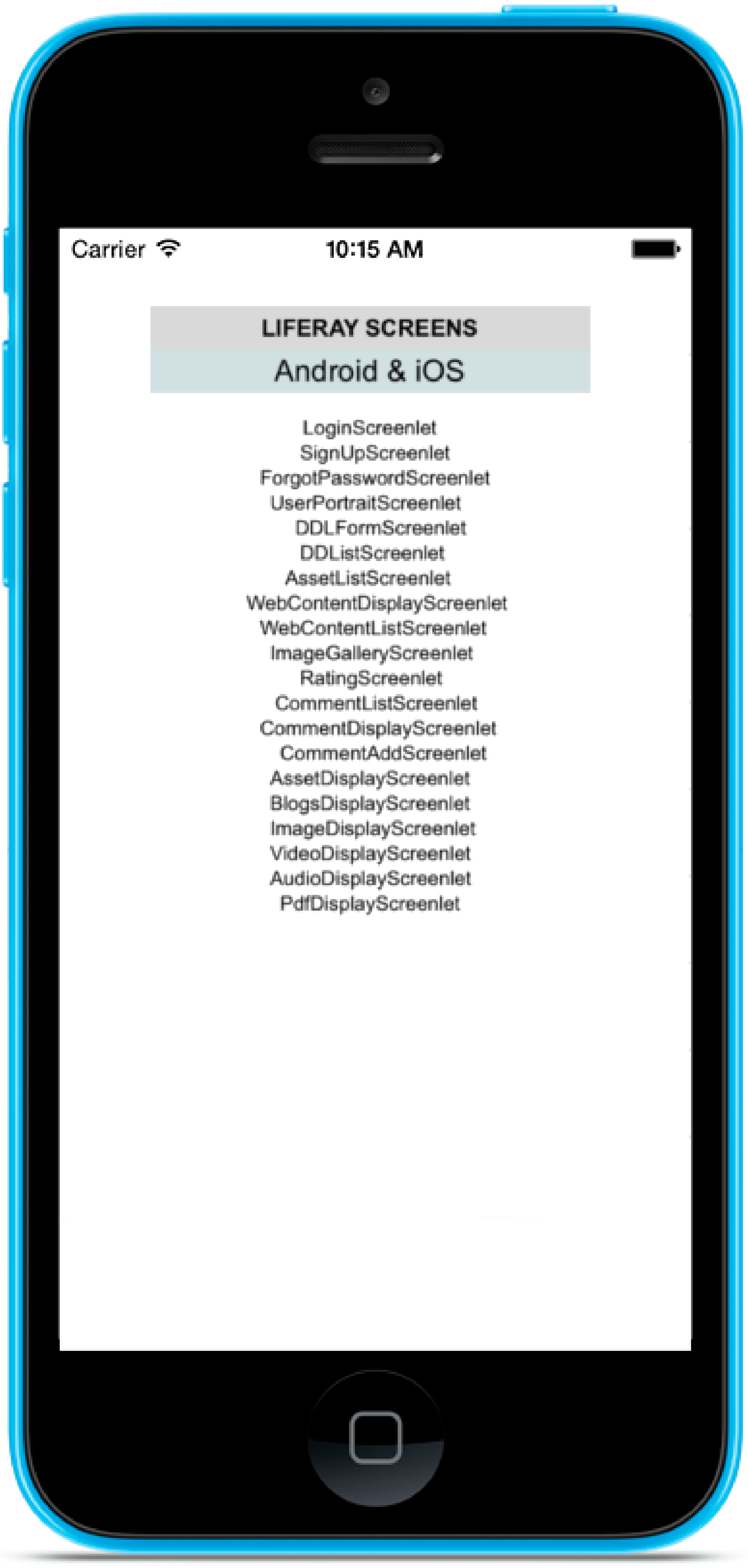


Figure 129.26: File Display Screenlet using the Default View.

The `className` and `classPK` attributes are required to instantiate the Screenlet. | `autoLoad` | `boolean` | Whether the file automatically loads when the Screenlet appears in the app's UI. The default value is `true`. | `offlinePolicy` | `string` | The offline mode setting. See the Offline section for details. |

Delegate

File Display Screenlet delegates some events to an object that conforms to the `FileDisplayScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `- screenlet:onFileAssetResponse:`: Called when the Screenlet receives the file.
- `- screenlet:onFileAssetError:`: Called when an error occurs in the process. An `NSError` object describes the error.

129.22 Web Screenlet for iOS

Requirements

- Xcode 9.3 or above
- iOS 11 SDK
- Liferay Portal 6.2 CE/EE, Liferay CE Portal 7.0/7.1, Liferay DXP 7.0+
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP 7.0+.

Compatibility

- iOS 9 and above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

Web Screenlet lets you display any web page. It also lets you customize the web page through injection of local and remote JavaScript and CSS files. If you're using Liferay DXP as backend, you can use Application Display Templates in your page to customize its content from the server side.

Module

- None

Themes

- Default

The Default Theme uses an iOS `WKWebView` for displaying the web page.

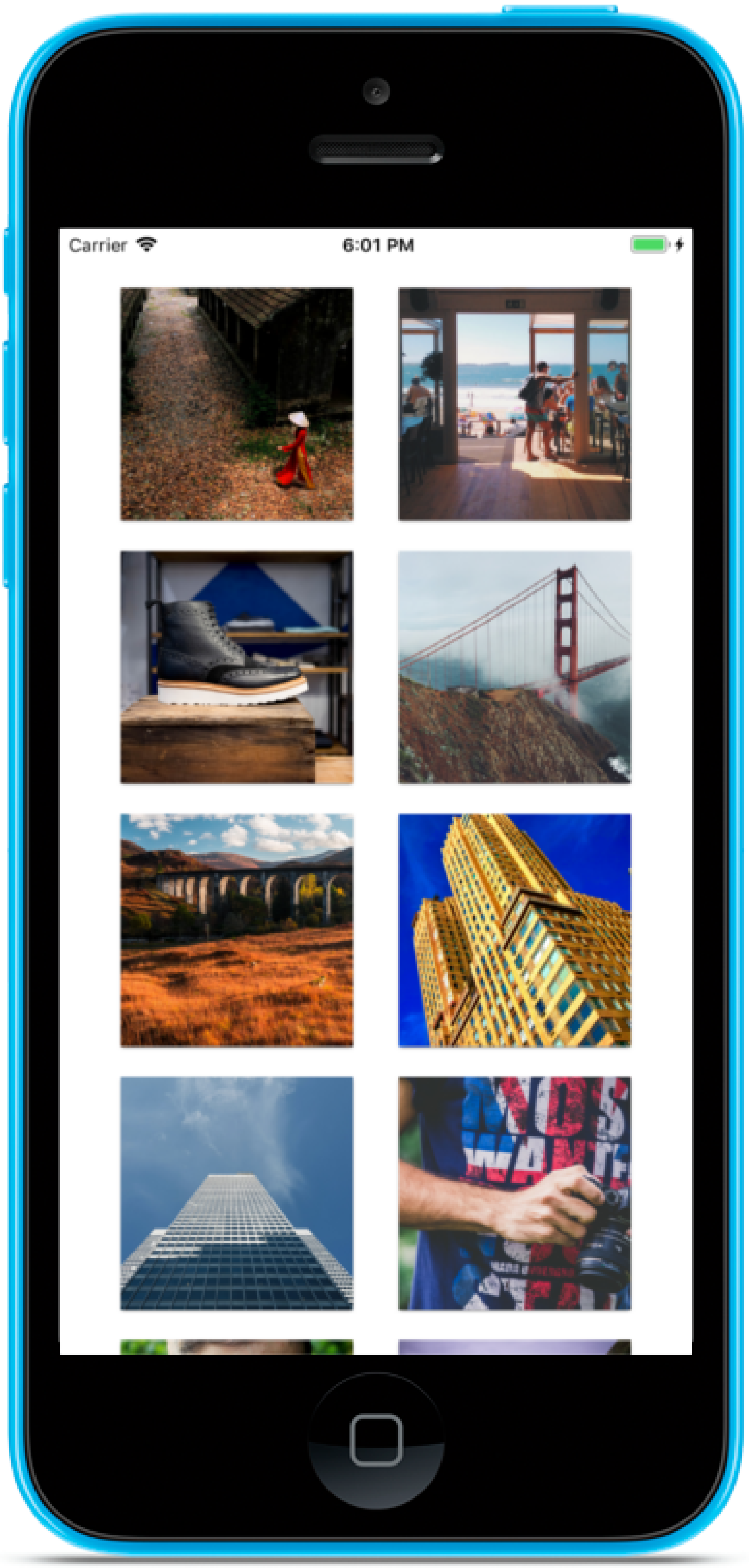


Figure 129.27: Web Screenlet using the Default Theme.

Configuration

To learn how to use Web Screenlet, follow the steps in the tutorial Rendering Web Pages in Your iOS App. That tutorial gives detailed instructions for using the configuration items described here.

Web Screenlet has `WebScreenletConfiguration` and `WebScreenletConfigurationBuilder` objects that you can use together to supply the parameters that the Screenlet needs to work. `WebScreenletConfigurationBuilder` has the following methods, which let you supply the described configuration parameters:

Method | Returns | Explanation | `addJs(localFile: String)` | `WebScreenletConfigurationBuilder` | Adds a local JavaScript file with the supplied filename. | `addCss(localFile: String)` | `WebScreenletConfigurationBuilder` | Adds a local CSS file with the supplied filename. | `addJs(url: String)` | `WebScreenletConfigurationBuilder` | Adds a JavaScript file from the supplied URL. | `addCss(url: String)` | `WebScreenletConfigurationBuilder` | Adds a CSS file from the supplied URL. | `set(webType: WebType)` | `WebScreenletConfigurationBuilder` | Sets the `WebType`. | `enableCordova()` | `WebScreenletConfigurationBuilder` | Enables Cordova inside the Web Screenlet. | `load()` | `WebScreenletConfiguration` | Returns the `WebScreenletConfiguration` object that you can set to the Screenlet instance. |

Note: If you want to add comments in the scripts, use the `/**/` notation.

WebType

- **WebType.liferayAuthenticated** (default): Displays a Liferay DXP page that requires authentication. The user must therefore be logged in with Screens via Login Screenlet or a `SessionContext` method. For this `WebType`, the URL you must pass to the `WebScreenletConfigurationBuilder` constructor is a relative URL. For example, if the full URL is `http://screens.liferay.org.es/web/guest/blog`, then the URL you must supply to the constructor is `/web/guest/blog`.

- **WebType.other**: Displays any other page. For this `WebType`, the URL you must pass to the `WebScreenletConfigurationBuilder` constructor is a full URL. For example, if the full URL is `http://screens.liferay.org.es/web/guest/blog`, then you must supply that URL to the constructor.

Attributes

Attribute | Data type | Explanation | `autoLoad` | `boolean` | Whether to load the page automatically when the Screenlet appears in the app's UI. The default value is `true`. | `loggingEnabled` | `boolean` | Whether logging is enabled. | `isScrollEnabled` | `boolean` | Whether to enable scrolling on the page inside the Screenlet. |

Delegate

Web Screenlet delegates some events to an object that conforms to the `WebScreenletDelegate` protocol. This protocol lets you implement the following methods:

- `onWebLoad(_:url:)`: Called when the Screenlet loads the page.

```
func onWebLoad(_ screenlet: WebScreenlet, url: String) {
    ...
}
```

- `screenlet(_:onScriptMessageNamespace:onScriptMessage:)`: Called when the `WKWebView` sends a message.

  ```
  func screenlet(_ screenlet: WebScreenlet,
            onScriptMessageNamespace namespace: String,
            onScriptMessage message: String) {
      ...
  }
  ```

- `screenlet(_:onError:)`: Called when an error occurs in the process. The `NSError` object describes the error.

  ```
  func screenlet(_ screenlet: WebScreenlet, onError error: NSError) {
      ...
  }
  ```

129.23 SyncManagerDelegate

The `SyncManagerDelegate` class is required to use Screenlets with offline mode. This class receives the events produced in the synchronization process. This document describes the class's methods.

Methods

The following method is invoked when the synchronization process is started. The number of items to be synced are passed.

```
syncManager(manager: SyncManager, itemsCount: UInt)
```

The following method is invoked when an item synchronization is about to start.

```
syncManager(manager: SyncManager, onItemSyncScreenlet screenlet: String,
    startKey: String, attributes: [String:AnyObject])
```

- `screenlet`: the screenlet name that stored this cache element
- `startKey`: the cache key where the item is stored
- `attributes`: some attributes stored together with the element. The specific attributes depend on the type of the entry. For more details, read the screenlet reference documentation.

The following method is invoked when an item synchronization is successfully completed.

```
syncManager(manager: SyncManager, onItemSyncScreenlet screenlet: String,
    completedKey: String, attributes: [String:AnyObject])
```

- `screenlet`: the screenlet name that stored this cache element
- `completedKey`: the cache key where the item is stored
- `attributes`: some attributes stored together with the element. The specific attributes depend on the type of the entry. For more details, read the screenlet reference documentation.

The following method is invoked when an item synchronization fails.

```
syncManager(manager: SyncManager, onItemSyncScreenlet screenlet: String,
    failedKey: String, attributes: [String:AnyObject], error: NSError)
```

- `screenlet`: the screenlet name that stored this cache element

- `failedKey`: the cache key where the item is stored
- `attributes`: some attributes stored together with the element. The specific attributes will depend on the type of the entry. For more details, read the screenlet reference documentation.
- `error`: the error occurred in the synchronization

The following method is invoked when an item synchronization detects a conflict. The method must invoke asynchronously a continuation argument with the conflict action result.

```
syncManager(manager: SyncManager, onItemSyncScreenlet screenlet: String,
    conflictedKey: String, remoteValue: AnyObject, localValue: AnyObject,
    resolve: SyncConflictResolution -> ())
```

- `screenlet`: the screenlet name that stored this cache element
- `conflictedKey`: the cache key where the item is stored
- `remoteValue`: the value stored in the server for the item being synchronized
- `localValue`: the value stored in the cache for the item being synchronized
- `resolve`: this is the continuation function to be called with the action result.

Supported values for `resolve` are:

- `UseRemote`: the remote version is overwritten with the local one. Both the local cache and the portal have the same version.
- `UseLocal`: the local version is overwritten with the remote one. Both the local cache and the portal have the same version
- `Discard`: the local version is removed and the remote one isn't overwritten.
- `Ignore`: data is not changed, so the next synchronization will detect the conflict again.

Chapter 130

Liferay Faces

Liferay Faces is an umbrella project that provides support for the JavaServer™ Faces (JSF) standard within Liferay Portal. It encompasses the following projects:

- **Liferay Faces Bridge** enables you to deploy JSF web apps as portlets without writing portlet-specific Java code. It also contains innovative features that make it possible to leverage the power of JSF 2.x inside a portlet application. Liferay Faces Bridge implements the JSR 329 Portlet Bridge Standard.
- **Liferay Faces Alloy** enables you to use AlloyUI components in a way that is consistent with JSF development.
- **Liferay Faces Portal** enables you to leverage Liferay-specific utilities and UI components in JSF portlets.

In this section of reference documentation, you'll learn more about each of these projects. You'll also learn about the Liferay Faces version scheme.

130.1 Liferay Faces Version Scheme

In this article, you'll learn which Liferay Faces artifacts should be used with your portlet and explore the Liferay Faces versioning scheme by discovering what each component of a version means. Once you have the versioning scheme mastered, you can view several example configurations.

Using The Liferay Faces Archetype Portlet

The Liferay Faces Archetype portlet can be used to determine the Liferay Faces artifacts and versions that you must include in your portlet. Select your preferred Liferay Portal version, JSF version, component suite (optional), and build tool, and the portlet will provide you with both a command to generate your portlet from a Maven archetype and a list of dependencies that can be copied into your build files. In the next section, you'll be provided with compatibility information about each version of the Liferay Faces artifacts.

Liferay Faces Alloy

Provides a suite of JSF components that utilize AlloyUI.

Branch|Example Artifact|AlloyUI|JSF API|Additional Info| master (3.x)|com.liferay.faces.alloy-3.0.1.jar|3.0.x|2.2+|*AlloyUI 3.0.x is the version that comes bundled with Liferay Portal 7.0+.*| 2.x|com.lif-eray.faces.alloy-2.0.1.jar|2.0.x|2.1+|*AlloyUI 2.0.x is the version that comes bundled with Liferay Portal 6.2.*| 1.x|com.liferay.faces.alloy-1.0.1.jar|2.0.x|1.2|*AlloyUI 2.0.x is the version that comes bundled with Liferay Portal 6.2.*|

Liferay Faces Bridge

Provides the ability to deploy JSF web applications as portlets within Apache Pluto, the reference implementation for JSR 286 (Portlet 2.0) and JSR 362 (Portlet 3.0).

Branch|Example Artifacts|Portlet API|JSF API|JCP Specification|Additional Info| API: 5.xIMPL: 5.x|com.liferay.faces.bridge.api-5.0.0.jarcom.liferay.faces.bridge.impl-5.0.0.jar|3.0|2.2|JSR 378|*The Expert Group began work in September 2015 and the Specification is currently under development.*| API: 4.xIMPL: 4.x|com.liferay.faces.bridge.api-4.1.0.jarcom.liferay.faces.bridge.impl-4.0.0.jar|2.0|2.2|JSR 329|*Includes non-standard bridge extensions for JSF 2.2.*| API: 3.xIMPL: 3.x|com.liferay.faces.bridge.api-3.1.0.jarcom.lif-eray.faces.bridge.impl-3.0.0.jar|2.0|2.1|JSR 329|*Includes non-standard bridge extensions for JSF 2.1.*| API: 2.xIMPL: 2.x|com.liferay.faces.bridge.api-2.1.0.jarcom.liferay.faces.bridge.impl-2.0.0.jar|2.0|1.2|JSR 329 (MR1)|*Includes support for Maintenance Release 1 (MR1).*| 1.x|N/A|1.0|1.2|JSR 301|*N/A (Not Applicable) since Liferay Faces Bridge has never implemented JSR 301.*|

Liferay Faces Bridge Ext

Extension to Liferay Faces Bridge that provides compatibility with Liferay Portal and also takes advantage of Liferay-specific features such as friendly URLs.

Branch |Example Artifact | Liferay Portal API | Bridge API | Portlet API |JSF API| 8.x|com.lif-eray.faces.bridge.ext-8.0.0.jar|7.3.0+|5.x|3.0|2.3| 7.x|com.liferay.faces.bridge.ext-7.0.0.jar|7.3.0+|5.x|3.0|2.2| 6.x|com.liferay.faces.bridge.ext-6.0.0.jar|7.3.0+|4.x|2.0|2.2| 5.x|com.liferay.faces.bridge.ext-5.0.4.jar|7.0.x/7.1.x/7.2.x|4.x| 4.x|UNUSED|N/A|N/A|N/A|N/A| 3.x|com.liferay.faces.bridge.ext-3.0.1.jar|6.2.x|4.x|2.0|2.2| 2.x|com.lif-eray.faces.bridge.ext-2.0.1.jar|6.2.x|3.x|2.0|2.1| 1.x|com.liferay.faces.bridge.ext-1.0.1.jar|6.2.x|2.x|2.0|1.2|

Liferay Faces Portal

Provides a suite of JSF components that are based on the JSP tags provided by Liferay Portal.

Branch|Example Artifact|Liferay Portal API | JSF API| 3.x|com.liferay.faces.portal-3.0.1.jar|7.0.x+|2.2+| 2.x|com.liferay.faces.portal-2.0.1.jar|6.2.x|2.1+| 1.x|com.liferay.faces.portal-1.0.1.jar|6.2.x|1.2|

Liferay Faces Util

Library that contains general purpose JSF utilities to support many of the sub-projects that comprise Liferay Faces.

Branch|Example Artifact| JSF API| 4.x|com.liferay.faces.util-3.1.0.jar|2.3| 3.x|com.liferay.faces.util-3.1.0.jar|2.2| 2.x|com.liferay.faces.util-2.1.0.jar|2.1| 1.x|com.liferay.faces.util-1.1.0.jar|1.2|

Now that you know all about the Liferay Faces versioning scheme, you may be curious as to how these components interact with each other. Refer to the following figure to view the Liferay Faces dependency diagram.

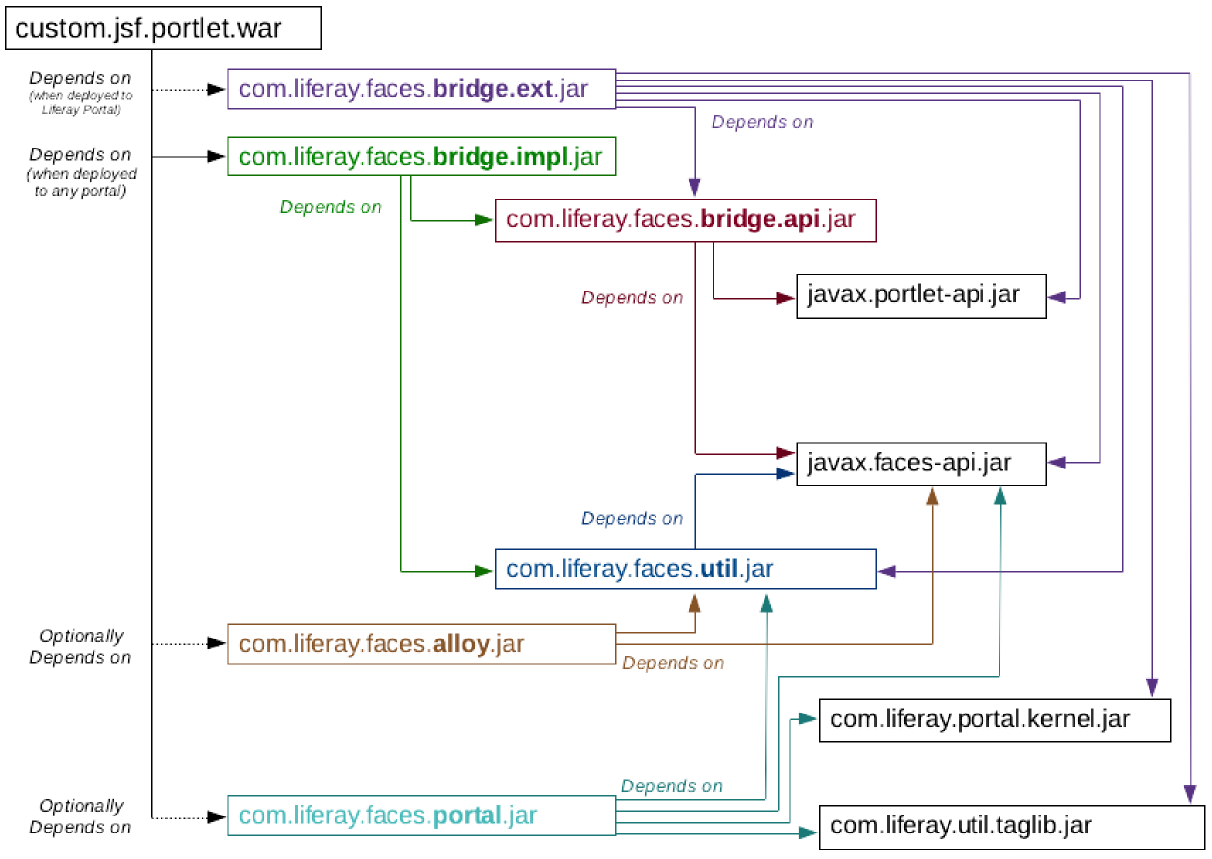

custom.jsf.portlet.war
Depends on (when deployed to Liferay Portal)
com.liferay.faces.bridge.ext.jar
Depends on
Depends on (when deployed to any portal)
com.liferay.faces.bridge.impl.jar
Depends on
com.liferay.faces.bridge.api.jar
Depends on
javax.portlet-api.jar
javax.faces-api.jar
Depends on
com.liferay.faces.util.jar
Optionally Depends on
com.liferay.faces.alloy.jar
Depends on
com.liferay.portal.kernel.jar
Depends on
Optionally Depends on
com.liferay.faces.portal.jar
com.liferay.util.taglib.jar

Figure 130.1: The Liferay Faces dependency diagram helps visualize how components interact and depend on each other.

Next, you can view some example configurations to see the new versioning scheme in action.

130.2 Understanding Liferay Faces Bridge

The Liferay Faces Bridge enables you to deploy JSF web apps as portlets without writing portlet-specific code. It also contains innovative features that make it possible to leverage the power of JSF 2.x inside a portlet application.

Liferay Faces Bridge is distributed in a .jar file. You can add Liferay Faces Bridge as a dependency to your portlet projects, in order to deploy your JSF web applications as portlets within JSR 286 (Portlet 2.0) compliant portlet containers, like Liferay Portal 5.2, 6.0, 6.1, 6.2, and 7.0.

The Liferay Faces Bridge project home page can be found here.

To fully understand Liferay Faces Bridge, you must first understand the portlet bridge standard. Because the Portlet 1.0 and JSF 1.0 specs were being created at essentially the same time, the Expert Group (EG) for the JSF specification constructed the JSF framework to be compliant with portlets. For example, the ExternalContext.getRequest() method returns an `Object` instead of an javax.servlet.http.HttpServletRequest. When this method is used in a portal, the `Object` can be cast to a javax.portlet.PortletRequest. Despite the EG's consciousness of portlet compatibility within the design of JSF, the gap between the portlet and JSF lifecycles had to be bridged.

Portlet bridge standards and implementations evolved over time.

Starting in 2004, several different JSF portlet bridge implementations were developed in order to provide JSF developers with the ability to deploy their JSF web apps as portlets. In 2006, the JCP formed the Portlet Bridge 1.0 (JSR 301) EG in order to define a standard bridge API, as well as detailed requirements for bridge implementations. JSR 301 was released in 2010, targeting Portlet 1.0 and JSF 1.2.

When the Portlet 2.0 (JSR 286) standard was released in 2008, it became necessary for the JCP to form the Portlet Bridge 2.0 (JSR 329) EG. JSR 329 was also released in 2010, targeting Portlet 2.0 and JSF 1.2.

After the JSR 314 EG released JSF 2.0 in 2009 and JSF 2.1 in 2010, it became evident that a Portlet Bridge 3.0 standard would be beneficial. In 2015 the JCP formed JSR 378) which is defining a bridge for Portlet 3.0 and JSF 2.2. There are also variants of *Liferay Faces Bridge* that support Portlet 2.0 and JSF 1.2/2.1/2.2.

Liferay Faces Bridge is the Reference Implementation (RI) of the Portlet Bridge Standard. It also contains innovative features that make it possible to leverage the power of JSF 2.x inside a portlet application.

Now that you're familiar with some of the history of the Portlet Bridge standards, you'll learn about the responsibilities required of the portlet bridge.

A JSF portlet bridge aligns the correct phases of the JSF lifecycle with each phase of the portlet lifecycle. For instance, if a browser sends an HTTP GET request to a portal page with a JSF portlet in it, the `RENDER_PHASE` is perfomed in the portlet's lifecycle. The JSF portlet bridge then initiates the `RESTORE_VIEW` and `RENDER_RESPONSE` phases in the JSF lifecycle. Likewise, when an HTTP POST is executed on a portlet and the portlet enters the `ACTION_PHASE`, then the full JSF lifecycle is initiated by the bridge.

Besides ensuring that the two lifecycles connect correctly, the JSF portlet bridge also acts as a mediator between the portal URL generator and JSF navigation rules. JSF portlet bridges ensure that URLs created by the portal comply with JSF navigation rules, so that a JSF portlet is able to switch to different views.

The JSR 329/378 standards defines several configuration options prefixed with the `javax.portlet.faces` namespace. Liferay Faces Bridge defines additional implementation-specific options prefixed with the `com.liferay.faces.bridge` namespace.

Liferay Faces Bridge is an essential part of the JSF development process for Liferay DXP.

Related Topics

Understanding Liferay Faces Alloy

Understanding Liferay Faces Portal

What is Service Builder?

130.3 Understanding Liferay Faces Alloy

Liferay Faces Alloy is distributed in a `.jar` file. You can add Liferay Faces Alloy as a dependency to your portlet projects, in order to use AlloyUI in a way that is consistent with JSF development.

During the creation of a JSF portlet in Liferay IDE/Developer Studio, you have the option of choosing the portlet's JSF Component Suite. The options include *JSF standard*, *ICEfaces*, *PrimeFaces*, *RichFaces*, and *Liferay Faces Alloy*.

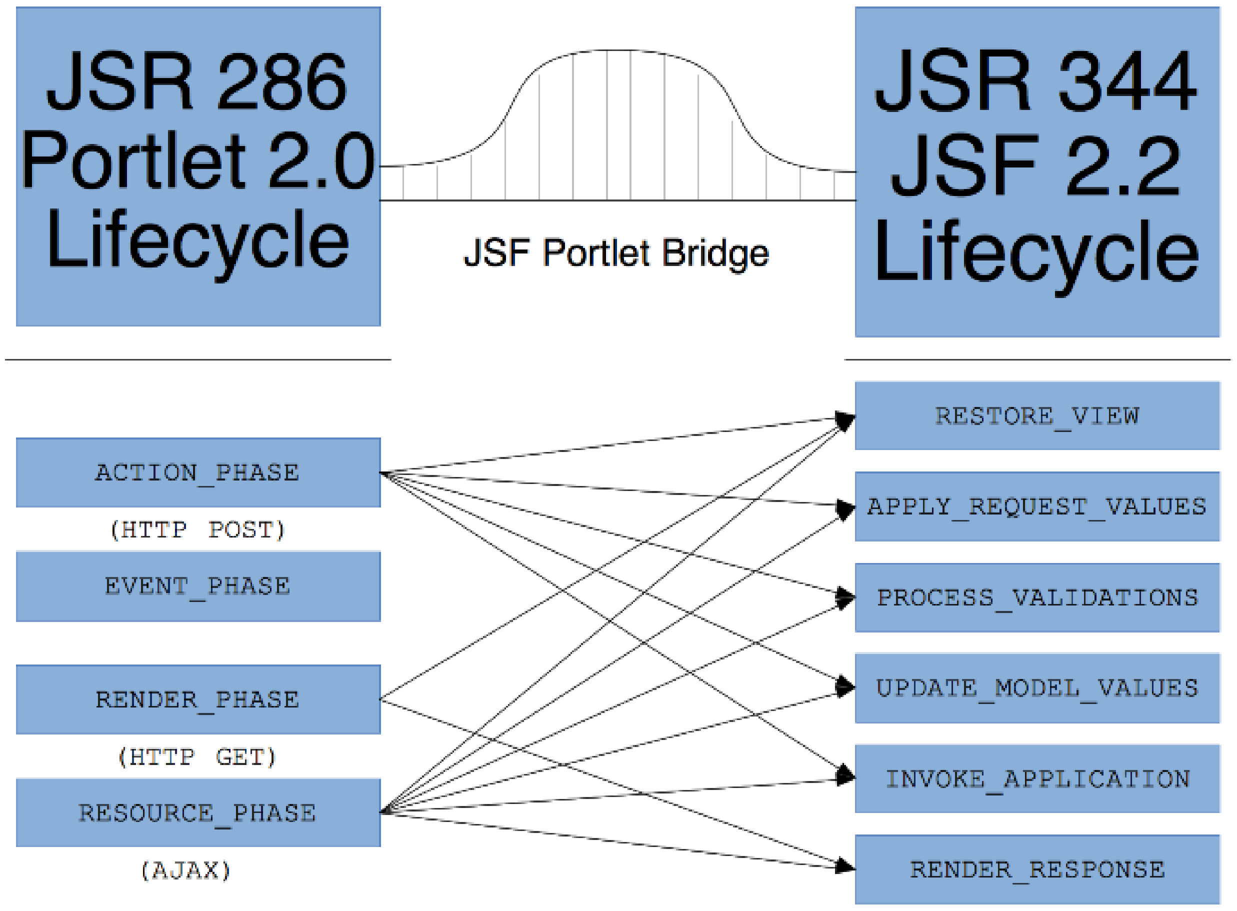


Figure 130.2: The different phases of the JSF Lifecycle are executed depending on which phase of the Portlet lifecycle is being executed.

If you selected the Liferay Faces Alloy JSF Component Suite during your portlet's setup, the `.jar` file is included in your portlet project.

The Liferay Faces Alloy project provides a set of UI components that utilize AlloyUI. For example, a brief list of some of the supported `aui:` tags are listed below:

- Input: `alloy:inputText`, `alloy:inputDate`, `alloy:inputFile`
- Panel: `alloy:accordion`, `alloy:column`, `alloy:fieldset`, `alloy:row`
- Select: `alloy:selectOneMenu`, `alloy:selectOneRadio`, `alloy:selectStarRating`

If you want to utilize Liferay's AlloyUI technology based on YUI3, you must include the Liferay Faces Alloy `.jar` file in your JSF portlet project. If you selected *Liferay Faces Alloy* during your JSF portlet's setup, you have Liferay Faces Alloy preconfigured in your project, so you're automatically able to use the `alloy:` tags.

As you can see, it's extremely easy to configure your JSF application to use Liferay's AlloyUI tags.

Related Topics

Creating a JSF Project Manually

Understanding Liferay Faces Bridge

Understanding Liferay Faces Portal

130.4 Understanding Liferay Faces Portal

Liferay Faces Portal is distributed in a `.jar` file. You can add Liferay Faces Portal as a dependency for your portlet projects to use its Liferay-specific utilities and UI components. When Liferay Faces Portal is included in a JSF portlet project, the `com.liferay.faces.portal.[version].jar` file resides in the portlet's library.

Some of the features included in Liferay Faces Portal are:

- Utilities: Provides the `LiferayPortletHelperUtil` which contains a variety Portlet-API and Liferay-specific convenience methods.
- JSF Components: Provides a set of JSF equivalents for popular Liferay DXP JSP tags (not exhaustive):
  - `liferay-ui:captcha` → `portal:captcha`
  - `liferay-ui:input-editor` → `portal:inputRichText`
  - `liferay-ui:search` → `portal:inputSearch`
  - `liferay-ui:header` → `portal:header`
  - `aui:nav` → `portal:nav`
  - `aui:nav-item` → `portal:navItem`
  - `aui:nav-bar` → `portal:navBar`
  - `liferay-security:permissionsURL` → `portal:permissionsURL`
  - `liferay-portlet:runtime` → `portal:runtime`

  For more information, visit https://liferayfaces.org/web/guest/portal-showcase.
- Expression Language: Adds a set of EL keywords such as `liferay` for getting Liferay-specific info, and `i18n` for integration with out-of-the-box Liferay internationalized messages.

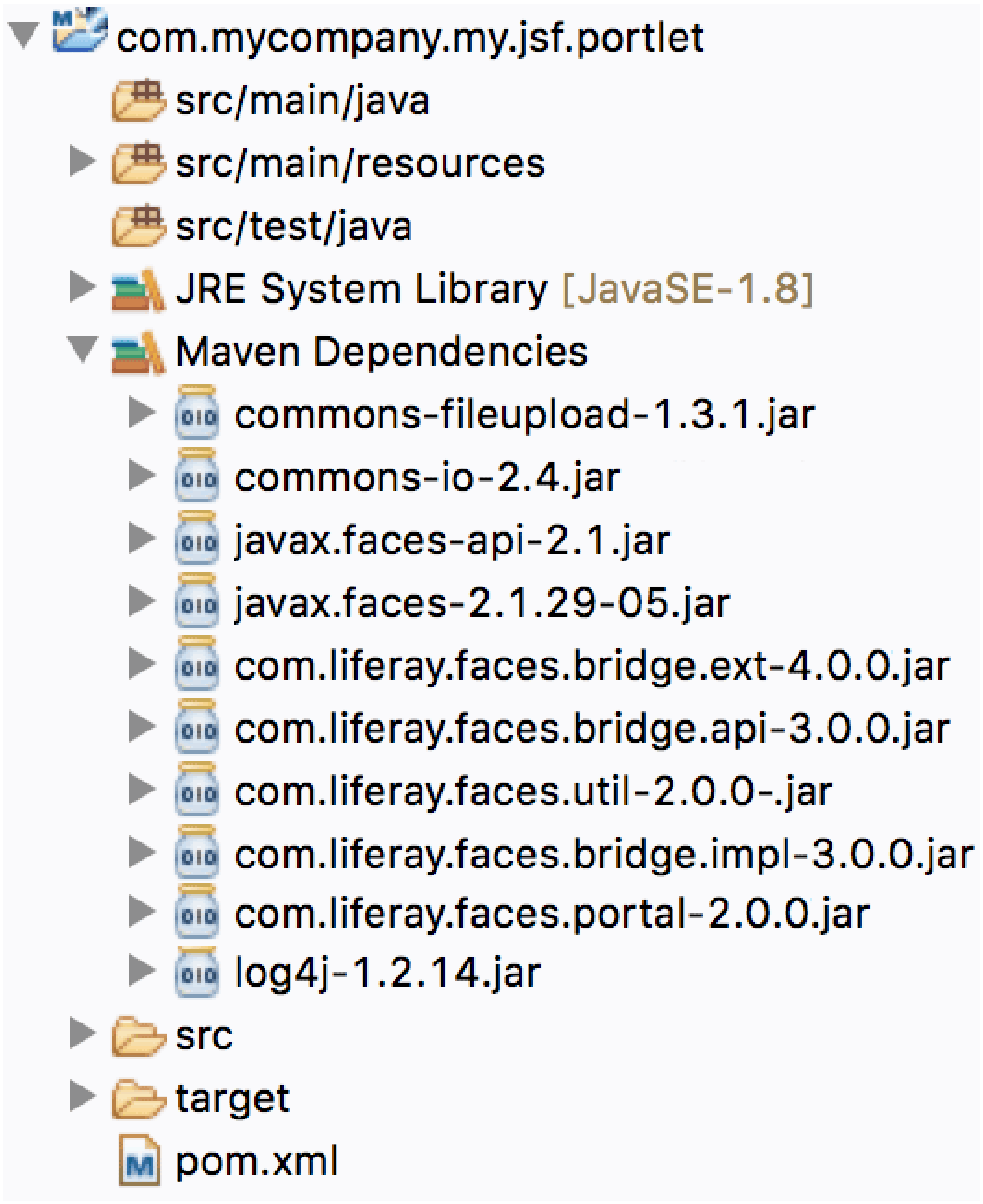
Great! You now have an understanding of what Liferay Faces Portal is, and what it accomplishes in your JSF application.

Related Topics

Creating a JSF Project Manually

Customizing Liferay Search

Understanding Liferay Faces Alloy



```
▼ com.mycompany.my.jsf.portlet
    src/main/java
  ▶ src/main/resources
    src/test/java
  ▶ JRE System Library [JavaSE-1.8]
  ▼ Maven Dependencies
    ▶ commons-fileupload-1.3.1.jar
    ▶ commons-io-2.4.jar
    ▶ javax.faces-api-2.1.jar
    ▶ javax.faces-2.1.29-05.jar
    ▶ com.liferay.faces.bridge.ext-4.0.0.jar
    ▶ com.liferay.faces.bridge.api-3.0.0.jar
    ▶ com.liferay.faces.util-2.0.0-.jar
    ▶ com.liferay.faces.bridge.impl-3.0.0.jar
    ▶ com.liferay.faces.portal-2.0.0.jar
    ▶ log4j-1.2.14.jar
  ▶ src
  ▶ target
    pom.xml
```

Figure 130.3: The required .jar files are downloaded for your JSF portlet based on the JSF UI Component Suite you configured.

Chapter 131

Gradle

Liferay provides plugins that you can apply to your Gradle project. This reference documentation describes how to apply and use Liferay's Gradle plugins.

Important: If you're using Liferay Workspace to create Liferay apps, many Liferay Gradle plugins are already applied by default. The `com.liferay.workspace` plugin provides the following plugins to all your apps in a Liferay Workspace:

- `com.liferay.css.builder`
- `com.liferay.js.module.config.generator`
- `com.liferay.js.transpiler`
- `com.liferay.javadoc.formatter`
- `com.liferay.jspc`
- `com.liferay.lang.builder`
- `com.liferay.source.formatter`
- `com.liferay.soy`
- `com.liferay.soy.translation`
- `com.liferay.tlddoc.builder`
- `com.liferay.tld.formatter`
- `com.liferay.test.integration`
- `com.liferay.xml.formatter`

Do not apply a Liferay Gradle plugin to an app that already has access to it.

Each article in this section describes how to apply the plugin, what Gradle tasks the plugin provides, the plugin's configuration properties, and the plugin's dependencies.

131.1 App Javadoc Builder Gradle Plugin

The App Javadoc Builder Gradle plugin lets you generate API documentation as a single, combined HTML document for an application that spans different subprojects, each one representing a different component of the same application.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in the build script of the root project:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.app.javadoc.builder", version: "1.2.2"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.app.javadoc.builder"
```

The App Javadoc Builder plugin automatically applies the `base` and `reporting-base` plugins.

Project Extension

The App Javadoc Builder plugin exposes the following properties through the extension named `appJavadocBuilder`:

Property Name | Type | Default Value | Description `copyTags` | `boolean` | `true` | Whether to copy the custom block tags configuration from the subprojects. It sets the Javadoc `-tag` argument for the `appJavadoc` task. `doclintDisabled` | `boolean` | `true` on JDK8+, `false` otherwise. | Whether to ignore Javadoc errors. It sets the Javadoc `-Xdoclint` and `-quiet` arguments for the `appJavadoc` task. `groupNameClosure` | `Closure<String>` | The subproject's description, or the subproject's name if the description is empty. | The closure invoked in order to get the group heading for a subproject. The given closure is passed a `Project` as its parameter. If `groupPackages` is `false`, this property is not used. `groupPackages` | `boolean` | `true` | Whether to separate packages on the overview page based on the subprojects they belong to. It sets the `-group` argument for the `appJavadoc` task. `subprojects` | `Set<Project>` | `project.subprojects` | The subprojects to include in the API documentation of the app.

The same extension exposes the following methods:

Method | Description `AppJavadocBuilderExtension onlyIf(Closure<Boolean> onlyIfClosure)` | Includes a subproject in the API documentation if the given closure returns `true`. The closure is evaluated at the end of the subproject configuration phase and is passed a single parameter: the subproject. If the closure returns `false`, the subproject is not included in the API documentation. `AppJavadocBuilderExtension onlyIf(Spec<Project> onlyIfSpec)` | Includes a subproject in the API documentation if the given spec is satisfied. The spec is evaluated at the end of the subproject configuration phase. If the spec is not satisfied, the subproject is not included in the API documentation. `AppJavadocBuilderExtension subprojects(Iterable<Project> subprojects)` | Include additional projects in the API documentation of the app. `AppJavadocBuilderExtension subprojects(Project... subprojects)` | Include additional projects in the API documentation of the app.

Tasks

The plugin adds two tasks to your project:

Name | Depends On | Type | Description `appJavadoc` | The `javadoc` tasks of the subprojects. | `Javadoc` | Generates Javadoc API documentation for the app. `jarAppJavadoc` | `appJavadoc` | `Jar` | Assembles a JAR archive containing the Javadoc files for this app.

The `appJavadoc` task is automatically configured with sensible defaults:

Property Name | Default Value classpath | The javadoc.classpath of all the subprojects. destinationDir | ${project.buildDir}/docs/javadoc options.encoding | "UTF-8" source | The javadoc.source of all the subprojects. title | project.reporting.apiDocTitle

131.2 Baseline Gradle Plugin

The Baseline Gradle plugin lets you verify that the OSGi semantic versioning rules are obeyed by your OSGi bundle.

When you run the baseline task, the plugin *baselines* the new bundle against the latest released non-snapshot bundle (i.e., the *baseline*). That is, it compares the public exported API of the new bundle with the baseline. If there are any changes, it uses the OSGi semantic versioning rules to calculate the minimum new version. If the new bundle has a lower version, errors are thrown.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.baseline", version: "2.1.0"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.baseline"
```

The Baseline plugin automatically applies the java and reporting-base plugins.

Since the plugin needs to download the baseline, you have to configure a repository that hosts it; for example, the central Maven 2 repository:

```
repositories {
    mavenCentral()
}
```

Project Extension

The Baseline plugin exposes the following properties through the baselineConfiguration extension:

Property Name | Type | Default Value | Description allowMavenLocal | boolean | false | Whether to let the baseline come from the local Maven cache (by default: ${user.home}/.m2). If the local Maven cache is not configured as a project repository, this property has no effect. lowestBaselineVersion | String | "1.0.0" | The greatest project version to ignore for the baseline check. If the project version is less than or equal to the value of this property, the baseline task is skipped. lowestMajorVersion | Integer | Content of the file ${project.projectDir}/.lfrbuild-lowest-major-version, where the default file name can be changed by setting the project property baseline.lowest.major.version.file. | The lowest major version of the released artifact to use in the baseline check. lowestMajorVersionRequired | boolean | false | Whether to fail the build if the lowestMajorVersion is not specified.

If the lowestMajorVersion is not specified, the plugin runs the check using the most recent released non-snapshot bundle as baseline, which matches the version range (,${project.version}). Otherwise, if the lowestMajorVersion is equal to a value L and the project has version M.x.y (with L less or equal than M), multiple checks are performed in order, using the following version ranges as baseline:

1. [L.0.0, (L + 1).0.0)
2. [(L + 1).0.0, (L + 2).0.0)
3. ...
4. [(M - 2).0.0, (M - 1).0.0)
5. [(M - 1).0.0, M.0.0)
6. [M.0.0, M.x.y)

The first failing check fails the whole build.

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description baseline | jar | BaselineTask | Compares the public API of this project with the public API of the previous released version, if found.

The baseline task is automatically configured with sensible defaults:

Property Name | Default Value baselineConfiguration | configurations.baseline bndFile | ${project.projectDir}/bnd.bnd newJarFile | project.tasks.jar.archivePath sourceDir | The first resources directory of the main source set (by default: src/main/resources).

BaselineTask

Task Properties Property Name | Type | Default Value | Description baselineConfiguration | Configuration | null | The configuration that contains exactly one dependency to the baseline bundle. bndFile | File | null | The BND file of the project. If provided, the task will automatically update the Bundle-Version header. forceCalculatedVersion | boolean | false | Whether to fail the baseline check if the Bundle-Version has been excessively increased. ignoreExcessiveVersionIncreases | boolean | false | Whether to ignore excessive package version increase warnings. ignoreFailures | boolean | false | Whether the build should not break when semantic versioning errors are found. logFile | File | null | The file to which the results of the baseline check are written. *(Read-only)* logFileName | String | "baseline/${task.name}.log" | The name of the file to which the results of the baseline check are written. If the reporting-base plugin is applied, the file name is relative to reporting.baseDir; otherwise, it's relative to the project directory. newJarFile | File | null | The file of the new OSGi bundle. reportDiff | boolean | true if the project property baseline.jar.report.level has either value "diff" or "persist"; false otherwise | Whether to show a granular, differential report of all changes that occurred in the exported packages of the OSGi bundle. reportOnlyDirtyPackages | boolean | Value of the project property baseline.jar.report.only.dirty.packages if specified; true otherwise. | Whether to show only packages with API changes in the report. sourceDir | File | null | The directory to which the packageinfo files are generated or updated.

The properties of type File support any type that can be resolved by project.file. Moreover, it is possible to use Closures and Callables as values for the String properties to defer evaluation until task execution.

Helper Tasks

If the lowestMajorVersion property is specified with a value L, the plugin creates a series of helper tasks of type BaselineTask at the end of the project evaluation, one for each major version between L and the major version M of the project:

1. Task `baseline${L + 1}`, which depends on `baseline${L + 2}` and uses the version range `[(L + 1).0.0, (L + 2).0.0)` as baseline.
2. Task `baseline${L + 2}`, which depends on `baseline${L + 3}` and uses the version range `[(L + 2).0.0, (L + 3).0.0)` as baseline.
3. …
4. Task `baseline${M - 2}`, which depends on `baseline${M - 1}` and uses the version range `[(M - 2).0.0, (M - 1).0.0)` as baseline.
5. Task `baseline${M - 1}`, which depends on `baseline${M}` and uses the version range `[(M - 1).0.0, M.0.0)` as baseline.
6. Task `baseline${M}`, which uses the version range `[M.0.0, M.x.y)` as baseline.

The `baseline` task is also configured to use the version range `[L.0.0, (L + 1).0.0)` as baseline, and to depend on the task `baseline${L + 1}`. This means that running the `baseline` task runs the baseline check against multiple versions, starting from the most recent `M` and going back to `L`.

Moreover, all tasks except `baseline${M}` have the property `ignoreExcessiveVersionIncreases` set to `true`.

Additional Configuration

There are additional configurations that can help you baseline your OSGi bundle.

Baseline Dependency

The plugin creates a configuration called `baseline` with a default dependency to a released non-snapshot version of the bundle:

- version range `[L.0.0, (L + 1).0.0)` if the `lowestMajorVersion` property is specified with a value `L`.
- version range `(,${project.version})` otherwise.

It is possible to override this setting and use a different version of the bundle as baseline.

System Properties

It is possible to set the default values of the `ignoreFailures` property for a `BaselineTask` task via system properties:

```
-D${task.name}.ignoreFailures=true
```

For example, run the following Bash command to execute the baseline check without breaking the build, in case of errors:

```
./gradlew baseline -Dbaseline.ignoreFailures=true
```

131.3 Change Log Builder Gradle Plugin

The Change Log Builder Gradle plugin lets you generate and maintain a change log file based on the Git commits in your project. A change log file generated by this plugin looks like this

```
#
# Bundle Version 1.0.1
#
9c77ff4c95cb1a325db3bdd089be105206e8b63c^..b421f00ac84b065685b131833fecc594fc01c760=LPS-123 LPS-1321

#
# Bundle Version 1.0.2
#
b421f00ac84b065685b131833fecc594fc01c760^..bc15d8d84e12b9544f78e4e3743c510dbaec2d89=LPS-456
```

Every time the `buildChangeLog` task is executed, a new line is added to the change log, which lists all Git commit prefixes (usually issue ticket IDs) that occurred in a certain range. The end of the range is always the tip of the current branch. The start range can vary, depending on the case:

- If `buildChangeLog` has never been executed for the project, the change log does not exist. Therefore, the most recent commit from two years ago is used for the range start.
- If a change log already exists for your project, the start range begins at the range end of the last line in the change log.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.change.log.builder", version: "1.1.3"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.change.log.builder"
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `buildChangeLog` | - | `BuildChangeLogTask` | Builds the change log file for this project.

The `buildChangeLog` task is automatically configured with sensible defaults, depending on whether the `java` plugin is applied:

Property Name | Default Value `changeLogHeader` | `"Bundle Version ${project.version}"` `changeLogFile` |

If the `java` plugin is applied: The `META-INF/liferay-releng.changelog` file in the first `resources` directory of the `main` source set (by default, `src/main/resources/META-INF/liferay-releng.changelog`).

Otherwise: `"${project.projectDir}/liferay-releng.changelog"`

`dirs` | `[project.projectDir]`

BuildChangeLogTask

Task Properties Property Name | Type | Default Value | Description `changeLogFile` | `File` | `null` | The change log file to build. `changeLogHeader` | `String` | `null` | The header for the new line in the change log. `dirs` | `FileCollection` | `[]` | The directories to consider when listing the commits in the range specified. `gitDir` | `File` | `project.rootDir` | The base directory to start searching for the `.git` directory. The search proceeds in all the ancestors of the directory specified. `rangeEnd` | `String` | `null` | The hash of the last commit to consider. If not set, it corresponds to the range end of the last line in the change log, or the most recent commit from at least two years ago if the change log file does not exist yet. `rangeStart` | `String` | `null` | The hash of the first commit to consider. If not set, it corresponds to the hash of the tip of the current branch. `ticketIdPrefixes` | `Set<String>` | `["CLDSVCS", "LPS", "SOS", "SYNC"]` | The valid prefix of the Git commit messages to add to the change log. For example, if a commit message is `"LPS-123 Bugfix"`, `"LPS-123"` will be added to the change log.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `String` properties to defer evaluation until task execution.

Task Methods Method | Description `BuildChangeLogTask dirs(Iterable<?> dirs)` | Adds directories to consider when listing the commits in the range specified. `BuildChangeLogTask dirs(Object... dirs)` | Adds directories to consider when listing the commits in the range specified. `BuildChangeLogTask ticketIdPrefixes(Iterable<String> ticketIdPrefixes)` | Adds valid prefixes of the Git commit messages to add to the change log. `BuildChangeLogTask ticketIdPrefixes(String... ticketIdPrefixes)` | Adds valid prefixes of the Git commit messages to add to the change log.

131.4 CSS Builder Gradle Plugin

The CSS Builder Gradle plugin lets you run the Liferay CSS Builder tool to compile Sass files in your project.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.css.builder", version: "3.0.0"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.css.builder"
```

Since the plugin automatically resolves the Liferay CSS Builder library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description buildCSS | - | BuildCSSTask | Compiles the Sass files in this project.

The plugin also adds the following dependencies to tasks defined by the java plugin:

Name | Depends On processResources | buildCSS

The buildCSS task is automatically configured with sensible defaults, depending on whether the java or the war plugins are applied:

Property Name | Default Value baseDir |

If the java plugin is applied: The first resources directory of the main source set (by default: src/main/resources).

If the war plugin is applied: project.webAppDir.

Otherwise: null

BuildCSSTask

Tasks of type BuildCSSTask extend JavaExec, so all its properties and methods, such as args and maxHeapSize, are available. They also have the following properties set by default:

Property Name | Default Value args | CSS Builder command line arguments classpath | project.configurations.cssBuilder defaultCharacterEncoding | "UTF-8" main | "com.liferay.css.builder.CSSBuilder" systemProperties | ["sass.compiler.jni.clean.temp.dir", true]

Task Properties Property Name | Type | Default Value | Description appendCssImportTimestamps | boolean | true | Whether to append the current timestamp to the URLs in the @import CSS at-rules. It sets the sass.append.css.import.timestamps argument. baseDir | File | null | The base directory that contains the SCSS files to compile. It sets the sass.docroot.dir argument. cssFiles | FileCollection | - | The SCSS files to compile. *(Read-only)* dirNames | List<String> | ["/"] | The name of the directories, relative to baseDir, which contain the SCSS files to compile. All sub-directories are searched for SCSS files as well. It sets the sass.dir argument. generateSourceMap | boolean | false | Whether to generate source maps for easier debugging. It sets the sass.generate.source.map argument. importDir | File | null | The META-INF/resources directory of the Liferay Frontend Common CSS artifact. This is required in order to make Bourbon and other CSS libraries available to the compilation. importFile | File | configurations.portalCommonCSS.singleFile | The Liferay Frontend Common CSS JAR file. If importDir is set, this property has no effect. importPath | File | - | The value of the importDir property if set; otherwise importFile. It sets the sass.portal.common.path argument. *(Read-only)* outputDirName | String | ".sass-cache/" | The name of the sub-directories where the SCSS files are compiled to. For each directory that contains SCSS files, a sub-directory with this name is created. It sets the sass.output.dir argument. outputDirs | FileCollection | - | The directories where the SCSS files are compiled to. Usually, these directories are ignored by the Version Control System. *(Read-only)* precision | int | 5 | The numeric precision of numbers in Sass. It sets the sass.precision argument. rtlExcludedPathRegexps | List<String> | [] | The SCSS file patterns to exclude when converting for right-to-left (RTL) support. It sets the sass.rtl.excluded.path.regexps argument. sassCompilerClassName | String | null | The type of Sass compiler to use. Supported values are "jni" and "ruby". If not set, defaults to "jni". It sets the sass.compiler.class.name argument.

Note: Liferay's CSS Builder is supported for Oracle's JDK and uses a native compiler for increased speed. If you're using an IBM JDK, you may experience issues when building your Sass files (e.g., when building a theme). It's recommended to switch to using the Oracle JDK, but if you prefer using the IBM JDK, you must use the fallback Ruby compiler. You can do this two ways:

- If you're working in a Liferay Workspace or using the Liferay Gradle Plugins plugin, set `sass.compiler.class.name=ruby` in your `gradle.properties` file.
- Otherwise, set `buildCSS.sassCompilerClassName='ruby'` in the project's `build.gradle` file.

The `sass.compiler.class.name=ruby` Gradle property only works for modules, so if you're using the Ruby compiler in a WAR project (e.g., theme), you must use the second option.

Be aware that the Ruby-based compiler doesn't perform as well as the native compiler, so expect longer compile times.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `int` and `String` properties, to defer evaluation until task execution.

Task Methods Method | Description `BuildCSSTask dirNames(Iterable<Object> dirNames)` | Adds sub-directory names, relative to `baseDir`, which contain the SCSS files to compile. `BuildCSSTask dirNames(Object... dirNames)` | Adds sub-directory names, relative to `baseDir`, which contain the SCSS files to compile. `BuildCSSTask rtlExcludedPathRegexps(Iterable<Object> rtlExcludedPathRegexps)` | Adds SCSS file patterns to exclude when converting for right-to-left (RTL) support. `BuildCSSTask rtlExcludedPathRegexps(Object... rtlExcludedPathRegexps)` | Adds SCSS file patterns to exclude when converting for right-to-left (RTL) support.

Additional Configuration

There are additional configurations that can help you use the CSS Builder.

Liferay CSS Builder Dependency

By default, the plugin creates a configuration called `cssBuilder` and adds a dependency to the latest released version of the Liferay CSS Builder. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `cssBuilder` configuration:

```
dependencies {
    cssBuilder group: "com.liferay", name: "com.liferay.css.builder", version: "3.0.0"
}
```

Liferay Frontend Common CSS Dependency

By default, the plugin creates a configuration called `portalCommonCSS` and adds a dependency to the latest released version of the Liferay Frontend Common CSS artifact. It is possible to override this setting and use a specific version of the artifact by manually adding a dependency to the `portalCommonCSS` configuration:

```
dependencies {
    portalCommonCSS group: "com.liferay", name: "com.liferay.frontend.css.common", version: "2.0.1"
}
```

131.5 DB Support Gradle Plugin

The DB Support Gradle plugin lets you run the Liferay DB Support tool to execute certain actions on a local Liferay database. So far, the following actions are available:

- Cleans the Liferay database from the Service Builder tables and rows of a module.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.db.support", version: "1.0.5"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.portal.tools.db.support"
```

Since the plugin automatically resolves the Liferay DB Support library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `cleanServiceBuilder` | - | `CleanServiceBuilderTask` | Cleans the Liferay database from the Service Builder tables and rows of a module.

The `cleanServiceBuilder` task is automatically configured with sensible defaults, depending on whether the `base` plugin is applied:

Property Name | Default Value `servletContextName` |

If the `base` plugin is applied: The bundle symbolic name of the project inferred via the `OsgiHelper` class.

Otherwise: `null`

`serviceXmlFile` | `"${project.projectDir}/service.xml"`

CleanServiceBuilderTask

Tasks of type `BuildDeploymentHelperTask` extend `JavaExec`, so all its properties and methods, such as `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `args` | The DB Support command line arguments. `classpath` | `project.configurations.dbSupport + project.configurations.dbSupportTool` `main` | `"com.liferay.portal.tools.db.support.D`

Task Properties Property Name | Type | Default Value | Description password | String | null | The user password for connecting to the Liferay database. It sets the --password argument. If propertiesFile is set, this property has no effect. propertiesFile | File | null | The portal-ext.properties file that contains the JDBC settings for connecting to the Liferay database. It sets the --properties-file argument. servletContextName | String | null | The servlet context name (usually the value of the Bundle-Symbolic-Name manifest header) of the module. It sets the --servlet-context-name argument. serviceXmlFile | File | null | The service.xml file of the module. It sets the --service-xml-file argument. url | String | null | The JDBC URL for connecting to the Liferay database. It sets the --url argument. If propertiesFile is set, this property has no effect. userName | String | null | The user name for connecting to the Liferay database. It sets the --user-name argument. If propertiesFile is set, this property has no effect.

The properties of type File support any type that can be resolved by project.file. Moreover, it is possible to use Closures and Callables as values for the int and String properties to defer evaluation until task execution.

Additional Configuration

There are additional configurations that can help you use the Deployment Helper.

JDBC Drivers Dependency

The plugin creates a configuration called dbSupport, which can be used to provide the suitable JDBC driver for your Liferay database:

```
dependencies {
    dbSupport group: "mysql", name: "mysql-connector-java", version: "5.1.23"
    dbSupport group: "org.mariadb.jdbc", name: "mariadb-java-client", version: "1.1.9"
    dbSupport group: "org.postgresql", name: "postgresql", version: "9.4-1201-jdbc41"
}
```

Liferay DB Support Dependency

By default, the plugin creates a configuration called dbSupportTool and adds a dependency to the latest released version of the Liferay DB Support. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the dbSupportTool configuration:

```
dependencies {
    dbSupportTool group: "com.liferay", name: "com.liferay.portal.tools.db.support", version: "1.0.8"
}
```

131.6 Dependency Checker Gradle Plugin

The Dependency Checker Gradle plugin lets you warn users if a specific configuration dependency is not the latest one available from the Maven central repository. The plugin eventually fails the build if the dependency age (the difference between the timestamp of the current version and the latest version) is above a predetermined threshold.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.dependency.checker", version: "1.0.3"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.dependency.checker"
```

Project Extension

The Dependency Checker Gradle plugin exposes the following properties through the extension named `dependencyChecker`:

Property Name | Type | Default Value | Description `ignoreFailures` | `boolean` | `true` | Whether to print an error message instead of failing the build when the dependency check fails, either for a network error or because the dependency is out-of-date.

The same extension exposes the following methods:

Method | Description `void maxAge(Map<?, ?> args)` | Declares the max age allowed for a dependency. The `args` map must contain the following entries:

`configuration`: the configuration name

`group`: the dependency group

`name`: the dependency name

`maxAge`: an instance of `groovy.time.Duration` that represents the maximum age allowed for the dependency

`throwError`: a `boolean` value representing whether to throw an error if the dependency is out-of-date

Additional Configuration

There are additional configurations that can help you use the Deployment Helper.

Project Properties

It is possible to set the default values of the `ignoreFailures` property via the project property `dependencyCheckerIgnoreFailures`:

```
-PdependencyCheckerIgnoreFailures=false
```

131.7 Deployment Helper Gradle Plugin

The Deployment Helper Gradle plugin lets you run the Liferay Deployment Helper tool to create a cluster deployable WAR from your OSGi artifacts.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.deployment.helper", version: "1.0.5"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.deployment.helper"
```

Since the plugin automatically resolves the Liferay Deployment Helper library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `buildDeploymentHelper` | - | `BuildDeploymentHelperTask` | Builds a WAR which contains one or more files that are copied once the WAR is deployed.

BuildDeploymentHelperTask

Tasks of type `BuildDeploymentHelperTask` extend `JavaExec`, so all its properties and methods, such as `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `args` | The Deployment Helper command line arguments. `classpath` | `project.configurations.deploymentHelper` `deploymentFiles` | The output files of the `jar` tasks of this project and all its sub-projects. `main` | `"com.liferay.deployment.helper.DeploymentHelper"` `outputFile` | `"${project.buildDir}/${project.name}.war"`

Task Properties Property Name | Type | Default Value | Description `deploymentFiles` | `FileCollection` | `[]` | The files or directories to include in the WAR and copy once the WAR is deployed. If a directory is added to this collection, all the JAR files contained in the directory are included in the WAR. `deploymentPath` | `File` | `null` | The directory to which the included files are copied. `outputFile` | `File` | `null` | The WAR file to build.

The properties of type `File` support any type that can be resolved by `project.file`.

Task Methods Method | Description `BuildDeploymentHelperTask deploymentFiles(Iterable<?> deploymentFiles)` | Adds files or directories to include in the WAR and copy once the WAR is deployed. The values are evaluated as per `project.files`. `BuildDeploymentHelperTask deploymentFiles(Object... deploymentFiles)` | Adds files or directories to include in the WAR and copy once the WAR is deployed. The values are evaluated as per `project.files`.

Additional Configuration

There are additional configurations that can help you use the Deployment Helper.

Liferay Deployment Helper Dependency

By default, the plugin creates a configuration called `deploymentHelper` and adds a dependency to the latest released version of the Liferay Deployment Helper. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `deploymentHelper` configuration:

```
dependencies {
    deploymentHelper group: "com.liferay", name: "com.liferay.deployment.helper", version: "1.0.4"
}
```

131.8 Go Gradle Plugin

The Go Gradle plugin lets you run Go as part of your build.

The plugin has been successfully tested with Gradle 3.5.1 up to 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.go", version: "1.0.0"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.go"
```

Project Extension

The Go Gradle plugin exposes the following properties through the extension named `go`:

Property Name | Type | Default Value | Description `goDir` | `File` | `"${project.buildDir}/go"` | The directory where the Go distribution is unpacked. `goUrl` | `String` | `"https://dl.google.com/go/go${go.goVersion}.${platform}-${bitMode}.${extension}` | The URL of the Go distribution to download. `goVersion` | `String` | `"1.11.4"` | The Go distribution's version to use. `workingDir` | `File` | `"${project.projectDir}"` | The directory that contains the project's Go source code.

Tasks

The plugin adds a series of tasks to your project:

Name | Depends On | Type | Description `downloadGo` | `-` | `DownloadGoTask` | Downloads and unpacks the local Go distribution for the project. `goBuild${programName}` | `downloadGo` | `ExecuteGoTask` | Compiles packages and dependencies for the Go program. `goClean${programName}` | `downloadGo` | `ExecuteGoTask` | Removes object files for the Go program. `goRun${programName}` | `downloadGo` | `ExecuteGoTask` | Compiles and runs the Go program. `goTest${programName}` | `downloadGo` | `ExecuteGoTask` | Tests packages for the Go program.

DownloadGoTask

The purpose of this task is to download and unpack a Go distribution.

Task Properties Property Name | Type | Default Value | Description goDir | File | null | The directory where the Go distribution is unpacked. goUrl | String | null | The URL of the Go distribution to download.

The File type properties support any type that can be resolved by project.file. Moreover, it is possible to use Closures and Callables as values for the String properties, to defer evaluation until task execution.

ExecuteGoTask

This is the base task to run Go in a Gradle build. All tasks of type ExecuteGoTask automatically depend on downloadGo.

Task Properties Property Name | Type | Default Value | Description args | List<Object> | [] | The arguments for the Go invocation. command | String | "go" | The file name of the executable to invoke. environment | Map<Object, Object> | [] | The environment variables for the Go invocation. inheritProxy | boolean | true | Whether to set the http_proxy, https_proxy, and no_proxy environment variables in the Go invocation based on the values of the system properties https.proxyHost, https.proxyPort, https.proxyUser, https.proxyPassword, https.nonProxyHosts, https.proxyHost, https.proxyPort, https.proxyUser, https.proxyPassword, and https.nonProxyHosts. If these environment variables are already set, their values will not be overwritten. goDir | File | go.goDir](#godir) | The directory that contains the executable to invoke. useGradleExec | boolean |

If running in a Gradle Daemon: true

Otherwise: false

| Whether to invoke Go using project.exec, which can solve hanging problems with the Gradle Daemon. workingDir | File | go.workingDir](#workingdir) | The working directory to use in the Go invocation.

The File type properties support any type that can be resolved by project.file. Moreover, it is possible to use Closures and Callables as values for the String properties to defer evaluation until task execution.

Task Methods Method | Description ExecuteGoTask args(Iterable<?> args) | Adds arguments for the Go invocation. ExecuteGoTask args(Object... args) | Adds arguments for the Go invocation. ExecuteGoTask environment(Map<?, ?> environment) | Adds environment variables for the Go invocation. ExecuteGoTask environment(Object key, Object value) | Adds an environment variable for the Go invocation.

gocommand{programName} Task

For each Go program in workingDir, four tasks of type ExecuteGoTask are added. Each of these tasks are automatically configured with sensible defaults:

Property Name | Default Value args | ["${command}", "${programFile.absolutePath}"]

131.9 Gulp Gradle Plugin

The Gulp Gradle plugin lets you run Gulp tasks as part of your build.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.gulp", version: "2.0.59"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.gulp"
```

The Gulp plugin automatically applies the `com.liferay.node` plugin.

Tasks

The plugin adds one task rule to your project:

Name | Depends On | Type | Description `gulp<Task>` | `downloadNode`, `npmInstall` | `ExecuteGulpTask` | Executes a named Gulp task.

ExecuteGulpTask

Tasks of type `ExecuteGulpTask` extend `ExecuteNodeScriptTask`, so all its properties and methods, such as `args` and `inheritProxy`, are available. They also have the following properties set by default:

Property Name | Default Value `scriptFile` | `"node_modules/gulp/bin/gulp.js"`

Gulp must be already installed in the `node_modules` directory of the project; otherwise, it will not be downloaded by the task. In order to ensure Gulp is installed, you can add the Gulp dependency to the project's `package.json` file.

Task Properties Property Name | Type | Default Value | Description `gulpCommand` | `String` | `null` | The Gulp task to execute.

It is possible to use Closures and Callables as values for the `String` properties to defer evaluation until task execution.

131.10 Jasper JSPC Gradle Plugin

The Jasper JSPC Gradle plugin lets you run the Liferay Jasper JSPC tool to compile the JavaServer Pages (JSP) files in your project. This can be useful to

- check for errors in the JSP files.
- pre-compile the JSP files for better performance.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.jasper.jspc", version: "2.0.5"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.jasper.jspc"
```

The Jasper JSPC plugin automatically applies the `java` plugin.

Since the plugin automatically resolves the Liferay Jasper JSPC library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds two tasks to your project:

Name | Depends On | Type | Description `compileJSP` | `generateJSPJava` | `JavaCompile` | Compiles JSP files to check for errors. `generateJSPJava` | `jar` | `CompileJSPTask` | Compiles JSP files to Java source files to check for errors.

The `generateJSPJava` task is automatically configured with sensible defaults, depending on whether the `war` plugin is applied:

Property Name | Default Value `classpath` | `project.configurations.jspCTool` `destinationDir` | `"${project.buildDir}/jspc"` `jspCClasspath` | `project.configurations.jspC` `webAppDir` |

If the `war` plugin is applied: `project.webAppDir`.

Otherwise: The first `resources` directory of the `main` source set (by default, `src/main/resources`).

The `compileJSP` task is also configured with the following defaults:

Property Name | Default Value `classpath` | `project.configurations.jspCTool + project.configurations.jspC` `destinationDir` | `compileJSP.temporaryDir` `source` | `generateJSPJava.outputs`

CompileJSPTask

Tasks of type `CompileJSPTask` extend `JavaExec`, so all its properties and methods, such as `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `main` | `"com.liferay.jasper.jspc.JspC"`

Task Properties Property Name | Type | Default Value | Description `destinationDir` | `File` | `null` | The directory where the the JSP files are compiled to. Package directories are automatically generated based on the directories containing the uncompiled JSP files. It sets the `-d` argument. `jspCClasspath` | `FileCollection` | `null` | The classpath to use for the JSP files compilation. `webAppDir` | `File` | `null` | The directory containing the web application. All JSP files in the directory and its subdirectories are compiled. It sets the `-webapp` argument.

The properties of type `File` support any type that can be resolved by `project.file`.

Additional Configuration

There are additional configurations that can help you use Jasper JSPC.

JSP Compilation Classpath

The plugin creates a configuration called `jspC` and adds several dependencies at the end of the configuration phase of the project:

- the JAR file of the project generated by the `jar` task.
- the output files of the `main` source set.
- the `compileClasspath` file collection of the `main` source set.

If necessary, it is possible to add more dependencies to the `jspC` configuration.

Liferay Jasper JSPC Dependency

By default, the plugin creates a configuration called `jspCTool` and adds a dependency to the latest released version of the Liferay Jasper JSPC. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `jspCTool` configuration:

```
dependencies {
    jspCTool group: "com.liferay", name: "com.liferay.jasper.jspc", version: "1.0.11"
    jspCTool group: "org.apache.ant", name: "ant", version: "1.9.4"
}
```

131.11 Javadoc Formatter Gradle Plugin

The Javadoc Formatter Gradle plugin lets you format project Javadoc comments using the Liferay Javadoc Formatter tool. The tool lets you generate:

- Default `@author` tags to all classes.
- Comment stubs to classes, fields, and methods.
- Missing `@Override` annotations.
- An XML representation of the Javadoc comments, which can be used by tools in order to index the Javadocs of the project.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.javadoc.formatter", version: "1.0.27"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.javadoc.formatter"
```

Since the plugin automatically resolves the Liferay Javadoc Formatter library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `formatJavadoc` | - | `FormatJavadocTask` | Runs the Liferay Javadoc Formatter to format files.

FormatJavadocTask

Tasks of type `FormatJavadocTask` extend `JavaExec`, so all its properties and methods, like `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `args` | Javadoc Formatter command line arguments `classpath` | `project.configurations.javadocFormatter` `main` | `"com.liferay.javadoc.formatter.JavadocFormatter"`

Task Properties Property Name | Type | Default Value | Description `author` | `String` | `"Brian Wing Shun Chan"` | The value of the `@author` tag to add at class level if missing. It sets the `javadoc.author` argument. `generateXML` | `boolean` | `false` | Whether to generate a XML representation of the Javadoc comments. The XML files are generated in the `src/main/resources` directory only if the Java files are contained in `src/main/java`. It sets the `javadoc.generate.xml` argument. `initializeMissingJavadocs` | `boolean` | `false` | Whether to add comment stubs at the class, field, and method levels. If `false`, only the class-level `@author` is added. It sets the `javadoc.init` argument. `limits` | `List<String>` | `[]` | The Java file name patterns, relative to `workingDir`, to include when formatting Javadoc comments. The patterns must be specified without the `.java` file type suffix. If empty, all Java files are formatted. It sets the `javadoc.limit` argument. `lowestSupportedJavaVersion` | `double` | `1.7` | If a method is annotated with the `@SinceJava` annotation and its `value` argument is greater than the value specified for the `lowestSupportedJavaVersion` property, then the `@Override` annotation is not automatically added, even if it is missing. It sets the `javadoc.lowest.supported.java.version` argument. See LPS-37353. `outputFilePrefix` | `String` | `"javadocs"` | The file name prefix of the XML representation of the Javadoc comments. If `generateXML` is `false`, this property is not used. It sets the `javadoc.output.file.prefix` argument. `updateJavadocs` | `boolean` | `false` | Whether to fix existing comment blocks by adding missing tags. It sets the `javadoc.update` argument.

It is possible to use Closures and Callables as values for the `String` properties, to defer evaluation until task execution.

Task Methods Method | Description `FormatJavadocTask dirNames(Iterable<Object> limits)` | Adds Java file name patterns, relative to `workingDir`, to include when formatting Javadoc comments. `FormatJavadocTask dirNames(Object... limits)` | Adds Java file name patterns, relative to `workingDir`, to include when formatting Javadoc comments.

Additional Configuration

There are additional configurations that can help you use the Javadoc Formatter.

Liferay Javadoc Formatter Dependency

By default, the plugin creates a configuration called `javadocFormatter` and adds a dependency to the latest released version of the Liferay Javadoc Formatter. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `javadocFormatter` configuration:

```
dependencies {
    javadocFormatter group: "com.liferay", name: "com.liferay.javadoc.formatter", version: "1.0.32"
}
```

If the `java` plugin is applied, the `javadocFormatter` configuration automatically extends from the `compile` configuration.

System Properties

It is possible to set the default values of the `generateXML`, `initializeMissingJavadocs`, `limits`, and `updateJavadocs` properties for a `FormatJavadocTask` task via system properties:

- `-D${task.name}.generate.xml=true`
- `-D${task.name}.init=SomeClassName1,SomeClassName2,com.liferay.portal.**`
- `-D${task.name}.limit=**/com/example/`
- `-D${task.name}.update=true`

131.12 JS Module Config Generator Gradle Plugin

The JS Module Config Generator Gradle plugin lets you run the Liferay AMD Module Config Generator to generate the configuration file needed to load AMD files via combo loader in Liferay.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.js.module.config.generator", version: "2.1.57"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.js.module.config.generator"
```

The JS Module Config Generator plugin automatically applies the `com.liferay.node` plugin.

Project Extension

The JS Module Config Generator plugin exposes the following properties through the extension named `jsModuleConfigGenerator`:

Property Name | Type | Default Value | Description `version` | `String` | `"1.2.1"` | The version of the Liferay AMD Module Config Generator to use.

Tasks

The plugin adds two tasks to your project:

Name | Depends On | Type | Description configJSModules | downloadLiferayModuleConfigGenerator, processResources | ConfigJSModulesTask | Generates the configuration file needed to load AMD files via combo loader in Liferay. downloadLiferayModuleConfigGenerator | downloadNode | DownloadNodeModuleTask | Downloads the Liferay AMD Module Config Generator in the project's node_modules directory.

By default, the downloadLiferayModuleConfigGenerator task downloads the version of liferay-module-config-generator declared in the jsModuleConfigGenerator.version property. If the project's package.json file, however, already lists the liferay-module-config-generator package in its dependencies or devDependencies, the downloadLiferayModuleConfigGenerator task is disabled.

The configJSModules task is automatically configured with sensible defaults, depending on whether the java plugin is applied:

Property Name | Default Value moduleConfigFile | "${project.projectDir}/package.json" outputFile | "${sourceSets.main.output.resourcesDir}/META-INF/config.json" sourceDir | "${sourceSets.main.output.resourcesDir}/ME INF/resources"

The plugin also adds the following dependencies to tasks defined by the java plugin:

Name | Depends On classes | configJSModules

If the com.liferay.js.transpiler plugin is applied, the configJSModules task is configured to always run after the transpileJS task.

ConfigJSModulesTask

Tasks of type ConfigJSModulesTask extend ExecuteNodeScriptTask, so all its properties and methods, such as args, inheritProxy, and workingDir, are available. The ConfigJSModulesTask instances also implement the PatternFilterable interface, which lets you specify include and exclude patterns for the files in sourceDir to process.

They also have the following properties set by default:

Property Name | Default Value includes | ["**/*.es.js*", "**/*.soy.js*"] scriptFile | "${downloadLiferayModuleConfig

The purpose of this task is to run the Liferay AMD Module Config Generator from the included files in sourceDir. The generator processes these files and creates a configuration file in the location specified by the outputFile property.

Task Properties Property Name | Type | Default Value | Description configVariable | String | null | The configuration variable to which the modules should be added. It sets the --config argument. customDefine | String | "Liferay.Loader" | The namespace of the define(...) call to use in the JS files. It sets the --namespace argument. ignorePath | boolean | false | Whether not to create module path and fullPath properties. It sets the --ignorePath argument. keepFileExtension | boolean | false | Whether to keep the file extension when generating the module name. It sets the --keepExtension argument. lowerCase | boolean | false | Whether to convert file name to lower case before using it as the module name. It sets the --lowerCase argument. moduleConfigFile | File | null | The JSON file which contains configuration data for the modules. It sets the --moduleConfig argument. moduleExtension | String | null | The extension for the module file (e.g., .js). If specified, use the provided string as an extension instead to get it automatically from the file name. It sets the --extension argument. moduleFormat | String | null | The regular expression and value to apply to the file name when generating the module name. It sets the --format argument. outputFile | File | null | The file where the generated configuration is stored. It sets the --output argument. sourceDir | File | null | The directory that contains the files to process.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `int` and `String` properties to defer evaluation until task execution.

131.13 JS Transpiler Gradle Plugin

The JS Transpiler Gradle plugin lets you run `metal-cli` to build Metal.js code, compile Soy files, and transpile ES6 to ES5.

Important: If you're using Liferay Workspace to create your app, the JS Transpiler Gradle plugin is applied by default. Do not apply the JS Transpiler Gradle plugin if you're using Liferay Workspace.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.js.transpiler", version: "2.4.36"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.js.transpiler"
```

There are two JS Transpiler Gradle plugins you can apply to your project:

- *JS Transpiler Plugin*: builds Metal.js code, compiles Soy files, and transpiles ES6 to ES5:

  ```
  apply plugin: "com.liferay.js.transpiler"
  ```

- *JS Transpiler Base Plugin*: provides a way to use Gradle dependencies (such as an external module or project dependencies) in Node.js scripts:

  ```
  apply plugin: "com.liferay.js.transpiler.base"
  ```

JS Transpiler Plugin

The JS Transpiler plugin automatically applies the *JS Transpiler Base Plugin*.

The plugin adds two tasks to your project:

Name | Depends On | Type | Description `downloadMetalCli` | `downloadNode` | `DownloadNodeModuleTask` | Downloads `metal-cli` in the project's `node_modules` directory. `transpileJS` | `downloadMetalCli`, `expandJSCompileDependencies`, `npmInstall`, `processResources` | `TranspileJSTask` | Builds Metal.js code.

By default, the `downloadMetalCli` task downloads the version 1.3.1 of `metal-cli`. If the project's `package.json` file, however, already lists the `metal-cli` package in its `dependencies` or `devDependencies`, the `downloadMetalCli` task is disabled.

The transpileJS task is automatically configured with sensible defaults, depending on whether the java plugin is applied:

Property Name | Default Value sourceDir | The directory META-INF/resources in the first resources directory of the main source set (by default, src/main/resources/META-INF/resources). workingDir | "${sourceSets.main.output.resourcesDir}/META-INF/resources"

The plugin also adds the following dependencies to tasks defined by the java plugin:

Name | Depends On classes | transpileJS

The plugin adds a new configuration to the project called soyCompile. If one or more dependencies are added to this configuration, they will be expanded into temporary directories and passed to the transpileJS task as additional soyDependencies values.

JS Transpiler Base Plugin

The JS Transpiler Base plugin automatically applies the com.liferay.node plugin.

The plugin adds a new configuration to the project called jsCompile. If one or more dependencies are added to this configuration, they will be expanded into sub-directories of the node_modules directory, with names equal to the names of the dependencies.

The plugin also adds one task to your project:

Name | Depends On | Type | Description expandJSCompileDependencies | - | DefaultTask | Expands the additional configured JavaScript dependencies. The task itself does not do any work, but depends on a series of Copy tasks called expandJSCompileDependency${file}, which expand each dependency declared in the jsCompile configuration into the node_modules directory.

All the tasks of type ExecuteNpmTask whose name starts with "npmRun" are configured to depend on expandJSCompileDependencies. This means that, before running any script declared in the package.json file of the project, all the jsCompile dependencies will be expanded into the node_modules directory.

Tasks

TranspileJSTask

Tasks of type TranspileJSTask extend ExecuteNodeScriptTask, so all its properties and methods, such as args, inheritProxy, and workingDir, are available. They also have the following properties set by default:

Property Name | Default Value scriptFile | "${downloadMetalCli.moduleDir}/index.js" soySrcIncludes | ["**/*.soy"] srcIncludes | ["**/*.es.js*", "**/*.soy.js*"]

The purpose of this task is to run the build command of metal-cli to build Metal.js code from sourceDir into the workingDir directory.

Task Properties Property Name | Type | Default Value | Description bundleFileName | String | null | The name of the final bundle file for formats (e.g., *globals*) that create one. It sets the --bundleFileName argument. globalName | String | null | The name of the global variable that holds exported modules. It sets the --globalName argument. This is only used by the *globals* format build. moduleName | String | null | The name of the project that is being compiled. All built modules are stored in a folder with this name. It sets the --moduleName argument. This is only used by the *amd* format build. modules | String | "amd" | The format(s) that the source files are built to. It sets the --format argument. skipWhenEmpty | boolean | true | Whether to disable the task and remove its dependencies if the sourceFiles property does not return any file at the end of the project evaluation. sourceDir | File | null | The directory that contains the files to build. sourceFiles | FileCollection | [] | The Soy and JS files to compile. *(Read-only)* sourceMaps | SourceMaps | enabled | Whether to generate source map files. Available values include disabled, enabled, and enabled_inline. soyDependencies | Set<String> | ["${npmInstall.workingDir}/node_modules/clay*/src/**/*.soy",

"${npmInstall.workingDir}/node_modules/metal*/src/**/*.soy"] | The path GLOBs of Soy files that the main source files depend on, but that should not be compiled. It sets the --soyDeps argument. soySkipMetalGeneration | boolean | false | Whether to just compile Soy files, without adding Metal.js generated code, like the component class. It sets the --soySkipMetalGeneration argument. soySrcIncludes | Set<String> | [] | The path GLOBs of the Soy files to compile. It sets the --soySrc argument. srcIncludes | Set<String> | [] | The path GLOBs of the JS files to compile. It sets the --src argument.

The properties of type File support any type that can be resolved by project.file. Moreover, it is possible to use Closures and Callables as values for the int and String properties to defer evaluation until task execution.

Task Methods Method | Description TranspileJSTask soyDependency(Iterable<?> soyDependencies) | Adds path GLOBs of Soy files that the main source files depend on, but that should not be compiled. TranspileJSTask soyDependency(Object... soyDependencies) | Adds path GLOBs of Soy files that the main source files depend on, but that should not be compiled. TranspileJSTask soySrcInclude(Iterable<?> soySrcIncludes) | Adds path GLOBs of Soy files to compile. TranspileJSTask soySrcInclude(Object... soySrcIncludes) | Adds path GLOBs of Soy files to compile. TranspileJSTask srcInclude(Iterable<?> srcIncludes) | Adds path GLOBs of JS files to compile. TranspileJSTask srcInclude(Object... srcIncludes) | Adds path GLOBs of JS files to compile.

131.14 JSDoc Gradle Plugin

The JSDoc Gradle plugin lets you run the JSDoc tool in order to generate documentation for your project's JavaScript files.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.jsdoc", version: "2.0.33"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}
```

There are two JSDoc Gradle plugins you can apply to your project:

- Apply the JSDoc Plugin to generate JavaScript documentation for your project:

  ```
  apply plugin: "com.liferay.jsdoc"
  ```

- Apply the App JSDoc Plugin in a parent project to generate the JavaScript documentation as a single, combined HTML document for an application that spans different subprojects, each one representing a different component of the same application:

  ```
  apply plugin: "com.liferay.app.jsdoc"
  ```

Both plugins automatically apply the com.liferay.node plugin.

JSDoc Plugin

The plugin adds two tasks to your project:

Name | Depends On | Type | Description `downloadJSDoc` | `downloadNode` | `DownloadNodeModuleTask` | Downloads JSDoc in the project's `node_modules` directory. `jsdoc` | `downloadJSDoc` | `JSDocTask` | Generates API documentation for the project's JavaScript code.

By default, the `downloadJSDoc` task downloads version `3.5.5` of the `jsdoc` package. If the project's `package.json` file, however, already lists the `jsdoc` package in its `dependencies` or `devDependencies`, the `downloadJSDoc` task is disabled.

The `jsdoc` task is automatically configured with sensible defaults, depending on whether the `java` plugin is applied:

Property Name | Default Value `destinationDir` |

If the `java` plugin is applied: `"${project.docsDir}/jsdoc"`

Otherwise: `"${project.buildDir}/jsdoc"`

`sourceDirs` | The directory `META-INF/resources` in the first `resources` directory of the `main` source set (by default, `src/main/resources/META-INF/resources`).

AppJSDoc Plugin

To use the App JSDoc plugin, it is required to apply the `com.liferay.app.jsdoc` plugin in a parent project (that is, a project that is a common ancestor of all the subprojects representing the various components of the app). It is also required to apply the `com.liferay.jsdoc` plugin to all the subprojects that contain JavaScript files.

The App JSDoc plugin adds three tasks to your project:

Name | Depends On | Type | Description `appJSDoc` | `downloadJSDoc` | `JSDocTask` | Generates API documentation for the app's JavaScript code. `downloadJSDoc` | `downloadNode` | `DownloadNodeModuleTask` | Downloads JSDoc in the app's `node_modules` directory. `jarAppJSDoc` | `appJSDoc` | `Jar` | Assembles a JAR archive containing the JavaScript documentation files for this app.

By default, the `downloadJSDoc` task downloads version `3.5.5` of the `jsdoc` package. If the project's `package.json` file, however, already lists the `jsdoc` package in its `dependencies` or `devDependencies`, the `downloadJSDoc` task is disabled.

The `appJSDoc` task is automatically configured with sensible defaults:

Property Name | Default Value `destinationDir` | `${project.buildDir}/docs/jsdoc` `sourceDirs` | The sum of all the `jsdoc.sourceDirs` values of the subprojects.

Project Extension

The App JSDoc plugin exposes the following properties through the extension named `appJSDocConfiguration`:

Property Name | Type | Default Value | Description `subprojects` | `Set<Project>` | `project.subprojects` | The subprojects to include in the JavaScript documentation of the app.

The same extension exposes the following methods:

Method | Description `AppJSDocConfigurationExtension subprojects(Iterable<Project> subprojects)` | Include additional projects in the JavaScript documentation of the app. `AppJSDocConfigurationExtension subprojects(Project... subprojects)` | Include additional projects in the JavaScript documentation of the app.

Tasks

JSDocTask

Tasks of type `JSDocTask` extend `ExecuteNodeScriptTask`, so all its properties and methods, such as `args`, `inheritProxy`, and `workingDir`, are available.

They also have the following properties set by default:

Property Name | Default Value `scriptFile` | `"${downloadJSDoc.moduleDir}/jsdoc.js"`

Task Properties Property Name | Type | Default Value | Description `configuration` | `TextResource` | `null` | The JSDoc configuration file. It sets the `--configure` argument. `destinationDir` | `File` | `null` | The directory where the JavaScript API documentation files are saved. It sets the `--destination` argument. `packageJsonFile` | `File` | `"${project.projectDir}/package.json"` | The path to the project's package file. It sets the `--package` argument. `sourceDirs` | `FileCollection` | `[]` | The directories that contains the files to process. `readmeFile` | `File` | `null` | The path to the project's README file. It sets the `--readme` argument. `tutorialsDir` | `File` | `null` | The directory in which JSDoc should search for tutorials. It sets the `--tutorials` argument.

The properties of type `File` support any type that can be resolved by `project.file`.

131.15 Lang Builder Gradle Plugin

The Lang Builder Gradle plugin lets you run the Liferay Lang Builder tool to sort and translate the language keys in your project.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.lang.builder", version: "3.0.12"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.lang.builder"
```

Since the plugin automatically resolves the Liferay Lang Builder library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

See this page on the *Liferay Developer Network* for more information about usage of the Lang Builder Gradle plugin.

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `buildLang` | - | `BuildLangTask` | Runs Liferay Lang Builder to translate language property files.

The `buildLang` task is automatically configured with sensible defaults, depending on whether the `java` plugin is applied:

Property Name | Default Value `langDir` |

If the `java` plugin is applied: The directory `content` in the first `resources` directory of the `main` source set (by default: `src/main/resources/content`).

Otherwise: `null`

BuildLangTask

Tasks of type `BuildLangTask` extend `JavaExec`, so all its properties and methods, such as `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `args` | Lang Builder command line arguments `classpath` | `project.configurations.langBuilder` `main` | `"com.liferay.lang.builder.LangBuilder"`

Task Properties Property Name | Type | Default Value | Description `excludedLanguageIds` | `Set<String>` | `["da", "de", "fi", "ja", "nl", "pt_PT", "sv"]` | The language IDs to exclude in the automatic translation. It sets the `lang.excluded.language.ids` argument. `langDir` | `File` | `null` | The directory where the language properties files are saved. It sets the `lang.dir` argument. `langFileName` | `String` | `"Language"` | The file name prefix of the language properties files (e.g., `Language_it.properties`). It sets the `lang.file` argument. `plugin` | `boolean` | `true` | Whether to check for duplicate language keys between the project and the portal. If `portalLanguagePropertiesFile` is not set, this property has no effect. It sets the `lang.plugin` argument. `portalLanguagePropertiesFile` | `File` | `null` | The `Language.properties` file of the portal. It sets the `lang.portal.language.properties.file` argument. `translate` | `boolean` | `true` | Whether to translate the language keys and generate a language properties file for each locale that's supported by Liferay. It sets the `lang.translate` argument. `translateSubscriptionKey` | `String` | `null` | The subscription key for Microsoft Translation integration. Subscription to the Translator Text Translation API on Microsoft Cognitive Services is required. Basic subscriptions, up to 2 million characters a month, are free. See here for more information. It sets the `lang.translate.subscription.key` argument.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `String` properties, to defer evaluation until task execution.

Task Methods Method | Description `BuildLangTask excludedLanguageIds(Iterable<Object> excludedLanguageIds)` | Adds language IDs to exclude in the automatic translation. `BuildLangTask excludedLanguageIds(Object... excludedLanguageIds)` | Adds language IDs to exclude in the automatic translation.

Additional Configuration

There are additional configurations that can help you use the Lang Builder.

Liferay Lang Builder Dependency

By default, the plugin creates a configuration called `langBuilder` and adds a dependency to the latest released version of the Liferay Lang Builder. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `langBuilder` configuration:

```
dependencies {
    langBuilder group: "com.liferay", name: "com.liferay.lang.builder", version: "1.0.31"
}
```

131.16 Maven Plugin Builder Gradle Plugin

The Maven Plugin Builder Gradle Plugin lets you generate the Maven plugin descriptor for any Mojos found in your project.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.maven.plugin.builder", version: "1.2.4"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.maven.plugin.builder"
```

Tasks

The plugin adds two tasks to your project:

Name | Depends On | Type | Description `buildPluginDescriptor` |`compileJava`, `WriteMavenSettings` | `BuildPluginDescriptorTask` | Generates the Maven plugin descriptor for the project. `WriteMavenSettings` | - | `WriteMavenSettingsTask` | Writes a temporary Maven settings file to be used during subsequent Maven invocations.

The Maven Plugin Builder Plugin automatically applies the `java` plugin.

The plugin also adds the following dependencies to tasks defined by the `maven` plugin:

Name | Depends On `install`, `uploadArchives`, and all the other tasks of type `Upload` | `buildPluginDescriptor`

The `buildPluginDescriptor` task is automatically configured with sensible defaults:

Property Name | Default Value `classesDir` | `sourceSets.main.output.classesDir` `mavenEmbedderClasspath` | `configurations.mavenEmbedder` `mavenSettingsFile` | `writeMavenSettings.outputFile` `outputDir` | The directory `META-INF/maven` in the first `resources` directory of the `main` source set (by default: `src/main/resources/META-INF/maven`). `pomArtifactId` | The bundle symbolic name of the project inferred via the `OsgiHelper` class. `pomGroupId` | `project.group` `pomVersion` | `project.version` (if it ends with `"-SNAPSHOT"`, the suffix will be removed) `sourceDir` | The first `java` directory of the `main` source set (by default: `src/main/java`).

The plugin ensures that the `processResources` task always runs before `buildPluginDescriptor` to let `processResources` copy the newly generated files in the `buildPluginDescriptor.outputDir` directory.

The `writeMavenSettings` task is also automatically configured with sensible defaults:

Property Name | Default Value `localRepositoryDir` | `maven.repo.local` system property `nonProxyHosts` | `http.nonProxyHosts` system property `outputFile` | `"${project.buildDir}/settings.xml"` `proxyHost` | `http.ProxyHost` or `https.proxyHost` system property (depending on the protocol of `repositoryUrl`) `proxyPassword` | `http.ProxyPassword` or `https.proxyPassword` system property (depending on the protocol of repositoryUrl) proxyPort | http.ProxyPort or https.proxyPort system property (depending on the protocol of repositoryUrl) proxyUser | http.ProxyUser or https.proxyUser system property (depending on the protocol of repositoryUrl) repositoryUrl | repository.url system property

If running on JDK8+, the plugin also disables the *doclint* feature in all tasks of type Javadoc.

BuildPluginDescriptorTask

Tasks of type BuildPluginDescriptorTask work by generating a temporary pom.xml file based on the project, and then invoking the Maven Embedder to build the Maven plugin descriptor.

It is possible to declare information for the plugin descriptor generation using either Java 5 Annotations or Javadoc Tags.

Task Properties Property Name | Type | Default Value | Description classesDir | File | null | The directory that contains the compiled classes. It sets the value of the build.outputDirectory element in the generated pom.xml file. configurationScopeMappings | Map<String, String> | ["compile": "compile", "provided", "provided"] | The mapping between the configuration names in the Gradle project and the dependency scopes in the pom.xml file. It is used to add dependencies.dependency elements in the generated pom.xml file. forcedExclusions | Set<String> | [] | The *group:name:version* notation of the dependencies to always exclude from the ones added in the pom.xml file. It adds dependencies.dependency.exclusions.exclusion elements to the generated pom.xml file. goalPrefix | String | null | The goal prefix for the Maven plugin specified in the descriptor. It sets the value of the build.plugins.plugin.configuration.goalPrefix element in the generated pom.xml file. mavenDebug | boolean | false | Whether to invoke the Maven Embedder in debug mode. mavenEmbedderClasspath | FileCollection | null | The classpath used to invoke the Maven Embedder. mavenEmbedderMainClassName | String | "org.apache.maven.cli.MavenCli" | The Maven Embedder's main class name. mavenPluginPluginVersion | String | "3.4" | The version of the Maven Plugin Plugin to use to generate the plugin descriptor for the project. mavenSettingsFile | File | null | The custom settings.xml file to use. It sets the --settings argument on the Maven Embedder invocation. outputDir | File | null | The directory where the Maven plugin descriptor files are saved. pomArtifactId | String | null | The identifier for the artifact that is unique within the group. It sets the value of the project.artifactId element in the generated pom.xml file. pomGroupId | String | null | The universally unique identifier for the project. It sets the value of the project.groupId element in the generated pom.xml file. pomRepositories | Map<String, Object> | ["liferay-public": "http://repository.liferay.com/nexus/content/groups/public"] | The name and URL of the remote repositories. It adds repositories.repository elements in the generated pom.xml file. pomVersion | String | null | The version of the artifact produced by this project. It sets the value of the project.version element in the generated pom.xml file. sourceDir | String | null | The directory that contains the source files. It sets the value of the build.sourceDirectory element in the generated pom.xml file. useSetterComments | boolean | true | Whether to allow Mojo Javadoc Tags in the setter methods of the Mojo.

The properties of type File support any type that can be resolved by project.file. Moreover, it is possible to use Closures and Callables as values for the String properties, to defer evaluation until task execution.

Task Methods

Method | Description void configurationScopeMapping(String configurationName, String scope) | Adds a mapping between a configuration name in the Gradle project and the dependency scope in the pom.xml file. BuildPluginDescriptorTask forcedExclusions(Iterable<String> forcedExclusions) | Adds *group:name:version* notations of dependencies to always exclude from the ones added in the pom.xml file. BuildPluginDescriptorTask forcedExclusions(String... forcedExclusions) | Adds *group:name:version* notations of dependencies to always exclude from the ones added in the pom.xml file. BuildPluginDescriptorTask pomRepositories(Map<String, ?> pomRepositories | Adds names and URLs of remote repositories in the pom.xml file. BuildPluginDescriptorTask pomRepository(String id, Object url) | Adds the name and URL of a remote repository in the pom.xml file.

WriteMavenSettingsTask

Task Properties Property Name | Type | Default Value | Description localRepositoryDir | String | null | The directory of the system's local repository. It sets the value of the localRepository element in the generated settings.xml file. nonProxyHosts | String | null | The patterns of the host that should be accessed without going through the proxy. It sets the value of the proxies.proxy.nonProxyHosts element in the generated settings.xml file. outputFile | File | null | The generated settings.xml file. proxyHost | String | null | The host name or address of the proxy server. It sets the value of the proxies.proxy.host element in the generated settings.xml file. proxyPassword | String | null | The password to use to access a protected proxy server. It sets the value of the proxies.proxy.password element in the generated settings.xml file. proxyPort | String | null | The port number of the proxy server. It sets the value of the proxies.proxy.port element in the generated settings.xml file. proxyUser | String | null | The user name to use to access a protected proxy server. It sets the value of the proxies.proxy.username element in the generated settings.xml file. repositoryUrl | String | null | The URL of the repository mirror. It sets the value of the mirrors.mirror.url element in the generated settings.xml file.

The properties of type File support any type that can be resolved by project.file. Moreover, it is possible to use Closures and Callables as values for the String properties, to defer evaluation until task execution.

Additional Configuration

There are additional configurations that can help you use the Maven Plugin Builder.

Maven Embedder Dependency

By default, the plugin creates a configuration called mavenEmbedder and adds a dependency to the 3.3.9 version of the Maven Embedder. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the mavenEmbedder configuration:

```
dependencies {
    mavenEmbedder group: "org.apache.maven", name: "maven-embedder", version: "3.3.9"
    mavenEmbedder group: "org.apache.maven.wagon", name: "wagon-http", version: "2.10"
    mavenEmbedder group: "org.eclipse.aether", name: "aether-connector-basic", version: "1.0.2.v20150114"
    mavenEmbedder group: "org.eclipse.aether", name: "aether-transport-wagon", version: "1.0.2.v20150114"
    mavenEmbedder group: "org.slf4j", name: "slf4j-simple", version: "1.7.5"
}
```

System Properties

It is possible to set the default value of the mavenDebug property for a BuildPluginDescriptorTask task via system property:

- -D${task.name}.maven.debug=true

For example, run the following Bash command to invoke the Maven Embedder in debug mode to attach a remote debugger:

```
./gradlew buildPluginDescriptor -DbuildPluginDescriptor.maven.debug=true
```

131.17 Node Gradle Plugin

The Node Gradle plugin lets you run Node.js and NPM as part of your build.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.node", version: "4.6.18"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.node"
```

Project Extension

The Node Gradle plugin exposes the following properties through the extension named `node`:

Property Name | Type | Default Value | Description `download` | `boolean` | `true` | Whether to download and use a local and isolated Node.js distribution instead of the one installed in the system. `global` | `boolean` | `false` | Whether to use a single Node.js installation for the whole multi-project build. This reduces the time required to unpack the Node.js distribution and the time required to download NPM packages thanks to a shared packages cache. If `download` is `false`, this property has no effect. `nodeDir` | `File` |

If `global` is `true`: `"${rootProject.buildDir}/node"`

Otherwise: `"${project.buildDir}/node"`

| The directory where the Node.js distribution is unpacked. If `download` is `false`, this property has no effect. `nodeUrl` | `String` | `"http://nodejs.org/dist/v${node.nodeVersion}/node-v${node.nodeVersion}-${platform}-x${bitMode}.${extension}"` | The URL of the Node.js distribution to download. If `download` is `false`, this property has no effect. `nodeVersion` | `String` | `"5.5.0"` | The version of the Node.js distribution to use. If `download` is `false`, this property has no effect. `npmArgs` | `List<String>` | `[]` | The arguments added automatically to every task of type `ExecuteNpmTask`. `npmUrl` | `String` | `"https://registry.npmjs.org/npm/-/npm-${node.npmVersion}.tgz"` | The URL of the NPM version to download. If `download` is `false`, this property has no effect. `npmVersion` | `String` | `null` | The version of NPM to use. If `null`, the version of NPM embedded inside the Node.js distribution is used. If `download` is `false`, this property has no effect.

It is possible to override the default value of the `download` property by setting the `nodeDownload` project property. For example, this can be done via command line argument:

```
./gradlew -PnodeDownload=false npmInstall
```

The same extension exposes the following methods:

Method | Description `NodeExtension npmArgs(Iterable<?> npmArgs)` | Adds arguments to automatically add to every task of type `ExecuteNpmTask`. `NodeExtension npmArgs(Object... npmArgs)` | Adds arguments to automatically add to every task of type `ExecuteNpmTask`.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for `String`, to defer evaluation until execution.

Please note that setting the `global` property of the `node` extension via the command line is not supported. It can only be set via Gradle script, which can be done by adding the following code to the `build.gradle` file in the root of a project (e.g., Liferay Workspace):

```
allprojects {
    plugins.withId("com.liferay.node") {
        node.global = true
    }
}
```

Tasks

The plugin adds a series of tasks to your project:

Name | Depends On | Type | Description `cleanNPM` | - | `Delete` | Deletes the `node_modules` directory, the `npm-shrinkwrap.json` file and the `package-lock.json` files from the project, if present. `downloadNode` | - | `DownloadNodeTask` | Downloads and unpacks the local Node.js distribution for the project. If `node.download` is `false`, this task is disabled. `npmInstall` | `downloadNode` | `NpmInstallTask` | Runs `npm install` to install the dependencies declared in the project's `package.json` file, if present. By default, the task is configured to run `npm install` two more times if it fails. `npmRun${script}` | `npmInstall` | `ExecuteNpmTask` | Runs the `${script}` NPM script. `npmPackageLock` | `cleanNPM`, `npmInstall` | `DefaultTask` | Deletes the NPM files and runs `npm install` to install the dependencies declared in the project's `package.json` file, if present. `npmShrinkwrap` | `cleanNPM`, `npmInstall` | `NpmShrinkwrapTask` | Locks down the versions of a package's dependencies in order to control which dependency versions are used.

DownloadNodeTask

The purpose of this task is to download and unpack a Node.js distribution.

Task Properties Property Name | Type | Default Value | Description `nodeDir` | `File` | `null` | The directory where the Node.js distribution is unpacked. `nodeExeUrl` | `String` | `null` | The URL of `node.exe` to download when on Windows. `nodeUrl` | `String` | `null` | The URL of the Node.js distribution to download. `npmUrl` | `String` | `null` | The URL of the NPM version to download.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `String` properties, to defer evaluation until task execution.

ExecuteNodeTask

This is the base task to run Node.js in a Gradle build. All tasks of type `ExecuteNodeTask` automatically depend on `downloadNode`.

Task Properties Property Name | Type | Default Value | Description `args` | `List<Object>` | `[]` | The arguments for the Node.js invocation. `command` | `String` | `"node"` | The file name of the executable to invoke. `environment` | `Map<Object, Object>` | `[]` | The environment variables for the Node.js invocation. `inheritProxy` | `boolean` | `true` | Whether to set the `http_proxy`, `https_proxy`, and `no_proxy` environment variables in the Node.js invocation based on the values of the system properties `https.proxyHost`, `https.proxyPort`, `https.proxyUser`, `https.proxyPassword`, `https.nonProxyHosts`, `https.proxyHost`, `https.proxyPort`, `https.proxyUser`, `https.proxyPassword`, and `https.nonProxyHosts`. If these environment variables are already set, their values will not be overwritten. `nodeDir` | `File` |

If `node.download` is `true`: `node.nodeDir`

Otherwise: `null`

| The directory that contains the executable to invoke. If `null`, the executable must be available in the system `PATH`. `npmInstallRetries` | `int` | `0` | The number of times the `node_modules` is deleted, the NPM cached data is verified (`npm cache verify`), and `npm install` is retried in case the Node.js invocation defined by this task fails. This can help solving corrupted `node_modules` directories by re-downloading the project's dependencies. `workingDir` | `File` | `project.projectDir` | The working directory to use in the Node.js invocation.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `String` properties to defer evaluation until task execution.

Task Methods Method | Description `ExecuteNodeTask args(Iterable<?> args)` | Adds arguments for the Node.js invocation. `ExecuteNodeTask args(Object... args)` | Adds arguments for the Node.js invocation. `ExecuteNodeTask environment(Map<?, ?> environment)` | Adds environment variables for the Node.js invocation. `ExecuteNodeTask environment(Object key, Object value)` | Adds an environment variable for the Node.js invocation.

ExecuteNodeScriptTask

The purpose of this task is to execute a Node.js script. Tasks of type `ExecuteNodeScriptTask` extend `ExecuteNodeTask`.

Task Properties Property Name | Type | Default Value | Description `scriptFile` | `File` | `null` | The Node.js script to execute.

The properties of type `File` support any type that can be resolved by `project.file`.

ExecuteNpmTask

The purpose of this task is to execute an NPM command. Tasks of type `ExecuteNpmTask` extend `ExecuteNodeScriptTask` with the following properties set by default:

Property Name | Default Value `command` |

If `nodeDir` is `null`: `"npm"`

Otherwise: `"node"`

`scriptFile` |

If `nodeDir` is `null`: `null`

Otherwise: `"${nodeDir}/lib/node_modules/npm/bin/npm-cli.js"` or `"${nodeDir}/node_modules/npm/bin/npm-cli.js"` on Windows.

Task Properties Property Name | Type | Default Value | Description `cacheConcurrent` | `boolean` |

If `node.npmVersion` is greater than or equal to `5.0.0`, or `node.nodeVersion` is greater than or equal to `8.0.0`: `true`

Otherwise: `false`

| Whether to run this task concurrently, in case the version of NPM in use supports multiple concurrent accesses to the same cache directory. `cacheDir` | `File` |

If `nodeDir` is `null`, or `node.npmVersion` is greater than or equal to `5.0.0`, or `node.nodeVersion` is greater than or equal to `8.0.0`: `null`

Otherwise: `"${nodeDir}/.cache"`

| The location of NPM's cache directory. It sets the `--cache` argument. Leave the property `null` to keep the default value. `logLevel` | `String` | Value to mirror the log level set in the task's `logger` object. | The NPM log level. It sets the –loglevel argument. `production` | `boolean` | `false` | Whether to run in production mode during the NPM invocation. It sets the `--production` argument. `progress` | `boolean` | `true` | Whether to show

a progress bar during the NPM invocation. It sets the `--progress` argument. `registry` | `String` | `null` | The base URL of the NPM package registry. It sets the `--registry` argument. Leave the property `null` or empty to keep the default value.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `String` properties, to defer evaluation until task execution.

DownloadNodeModuleTask

The purpose of this task is to download a Node.js package. The packages are downloaded in the `${workingDir}/node_modules` directory, which is equal, by default, to the `node_modules` directory of the project. Tasks of type `DownloadNodeModuleTask` extend `ExecuteNpmTask` in order to execute the command `npm install ${moduleName}@${moduleVersion}`.

`DownloadNodeModuleTask` instances are automatically disabled if the project's `package.json` file already lists a module with the same name in its `dependencies` or `devDependencies` object.

Task Properties Property Name | Type | Default Value | Description `moduleName` | `String` | `null` | The name of the Node.js package to download. `moduleVersion` | `String` | `null` | The version of the Node.js package to download.

It is possible to use Closures and Callables as values for the `String` properties, to defer evaluation until task execution.

NpmInstallTask

Purpose of these tasks is to install the dependencies declared in a `package.json` file. Tasks of type `NpmInstallTask` extend `ExecuteNpmTask` in order to run the command `npm install`.

`NpmInstallTask` instances are automatically disabled if the `package.json` file does not declare any dependency in the `dependency` or `devDependencies` object.

Task Properties Property Name | Type | Default Value | Description `nodeModulesCacheDir` | `File` | `null` |

The directory where `node_modules` directories are cached. By setting this property, it is possible to cache the `node_modules` directory of a project and avoid unnecessary invocations of `npm install`, useful especially in Continuous Integration environments.

The `node_modules` directory is cached based on the content of the project's `package-lock.json` (or `npm-shrinkwrap.json`, or `package.json` if absent). Therefore, if `NpmInstallTask` tasks in multiple projects are configured with the same `nodeModulesCacheDir`, and their `package-lock.json`, `npm-shrinkwrap.json` or `package.json` declare the same dependencies, their `node_modules` caches will be shared.

This feature is not available if the `com.liferay.cache` plugin is applied.

`nodeModulesCacheNativeSync` | `boolean` | `true` | Whether to use `rsync` (on Linux/macOS) or `robocopy` (on Windows) to cache and restore the `node_modules` directory. If `nodeModulesCacheDir` is not set, this property has no effect. `nodeModulesDigestFile` | `File` | `null` |

If this property is set, the content of the project's `package-lock.json` (or `npm-shrinkwrap.json`, or `package.json` if absent) is checked with the digest from the `node_modules` directory. If the digests match, do nothing. If the digests don't match, the `node_modules` directory is deleted before running `npm install`.

This feature is not available if the `com.liferay.cache` plugin is applied or if the property `nodeModulesCacheDir` is set.

`removeShrinkwrappedUrls` | `boolean` | `true` if the registry property has a value, `false` otherwise. | Whether to temporarily remove all the hard-coded URLs in the `from` and `resolved` fields of the `npm-shinkwrap.json` file before invoking `npm install`. This way, it is possible to force NPM to download all dependencies from a custom registry declared in the `registry` property. `useNpmCI` | `boolean` | `false` | Whether to run `npm ci` instead of `npm install`. If the `package-lock.json` file does not exist, this property has no effect.

The properties of type `File` support any type that can be resolved by `project.file`.

NpmShrinkwrapTask

The purpose of this task is to lock down the versions of a package's dependencies so that you can control exactly which dependency versions are used when your package is installed. Tasks of type `NpmShrinkwrapTask` extend `ExecuteNpmTask` to execute the command `npm shrinkwrap`.

The generated `npm-shrinkwrap.json` file is automatically sorted and formatted, so it's easier to see the changes with the previous version.

`NpmShrinkwrapTask` instances are automatically disabled if the `package.json` file does not exist.

Task Properties Property Name | Type | Default Value | Description `excludedDependencies` | `List<String>` | `[]` | The package names to exclude from the generated `npm-shrinkwrap.json` file. `includeDevDependencies` | `boolean` | `true` | Whether to include the package's `devDependencies`. It sets the `--dev` argument.

It is possible to use Closures and Callables as values for the `String` properties to defer evaluation until task execution.

Task Methods Method | Description `NpmShrinkwrapTask excludeDependencies(Iterable<?> excludedDependencies)` | Adds package names to exclude from the generated `npm-shrinkwrap.json` file. `NpmShrinkwrapTask excludeDependencies(Object... excludedDependencies)` | Adds package names to exclude from the generated `npm-shrinkwrap.json` file.

PublishNodeModuleTask

The purpose of this task is to publish a package to the NPM registry. Tasks of type `PublishNodeModuleTask` extend `ExecuteNpmTask` in order to execute the command `npm publish`.

These tasks generate a new temporary `package.json` file in the directory assigned to the `workingDir` property; then the `npm publish` command is executed. If the `package.json` file in that location does not exist, the one in the root of the project directory (if found) is copied; otherwise, a new file is created.

The `package.json` is then processed by adding the values provided by the task properties, if not already present in the file itself. It is still possible to override one or more fields of the `package.json` file with the values provided by the task properties by adding one or more keys (e.g., `"version"`) to the `overriddenPackageJsonKeys` property.

Task Properties Property Name | Type | Default Value | Description `moduleAuthor` | `String` | `null` | The value of the `author` field in the generated `package.json` file. `moduleBugsUrl` | `String` | `null` | The value of the `bugs.url` field in the generated `package.json` file. `moduleDescription` | `String` | `project.description` | The value of the `description` field in the generated `package.json` file. `moduleKeywords` | `List<String>` | `[]` | The value of the `keywords` field in the generated `package.json` file. `moduleLicense` | `String` | `null` | The value of the `license` field in the generated `package.json` file. `moduleMain` | `String` | `null` | The value of the `main` field in the generated `package.json` file. `moduleName` | `String` | Name based on `osgiHelper.bundleSymbolicName`: for example, if `osgiHelper.bundleSymbolicName` is `"com.liferay.gradle.plugins.node"`, the default value for the `moduleName` property is `"liferay-gradle-plugins-node"`. | The value of the `name` field in the generated `package.json` file. `moduleRepository` | `String` | `null` | The value of the `repository` field in the generated `package.json` file. `moduleVersion` | `String` | `project.version` | The value of the `version` field in the generated `package.json` file. `npmEmailAddress` | `String` | `null` | The email address of the npmjs.com user that publishes the package. npmPassword | String | null | The password of the npmjs.com user that publishes the package. npmUserName | String | null | The name of the npmjs.com user that publishes the package. overriddenPackageJsonKeys | Set<String> | [] | The field values to override in the generated package.json file.

Task Methods

Method	Description
PublishNodeModuleTask overriddenPackageJsonKeys(Iterable<String> overriddenPackageJsonKeys)	Adds field values to override in the generated package.json file.
PublishNodeModuleTask overriddenPackageJsonKeys(String... overriddenPackageJsonKeys)	Adds field values to override in the generated package.json file.

npmRun${script} Task

For each script declared in the package.json file of the project, one task npmRun${script} of type ExecuteNpmTask is added. Each of these tasks is automatically configured with sensible defaults:

Property Name | Default Value args | ["run-script", "${script}"]

If the java plugin is applied and the package.json file declares a script named "build", the script is executed before the classes task but after the processResources task.

If the lifecycle-base plugin is applied and the package.json file declares a script named test, the script is executed when running the check task.

131.18 Service Builder Gradle Plugin

The Service Builder Gradle plugin lets you generate a service layer defined in a Service Builder service.xml file.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.service.builder", version: "2.2.46"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.portal.tools.service.builder"
```

The Service Builder plugin automatically applies the java plugin.

Since the plugin automatically resolves the Liferay Service Builder library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `buildService` | - | `BuildServiceTask` | Runs the Liferay Service Builder.

The `buildService` task is automatically configured with sensible defaults, depending on whether the `war` plugin is applied, or whether the `osgiModule` property is `true`:

Property Name | Default Value `apiDir` |

If the `war` plugin is applied: `${project.webAppDir}/WEB-INF/service`

Otherwise: `null`

`hbmFile` |

If `osgiModule` is true: `${buildService.resourcesDir}/META-INF/module-hbm.xml`

Otherwise: `${buildService.resourcesDir}/META-INF/portlet-hbm.xml`

`implDir` | The first `java` directory of the `main` source set (by default: `src/main/java`). `inputFile` |

If the `war` plugin is applied: `${project.webAppDir}/WEB-INF/service.xml`

Otherwise: `${project.projectDir}/service.xml`

`modelHintsFile` | The file `META-INF/portlet-model-hints.xml` in the first `resources` directory of the `main` source set (by default: `src/main/resources/META-INF/portlet-model-hints.xml`). `pluginName` |

If `osgiModule` is true: `""`

Otherwise: `project.name`

`propsUtil` |

If `osgiModule` is true: `"${bundleSymbolicName}.util.ServiceProps"`The `bundleSymbolicName` of the project is inferred via the `OsgiHelper` class.

Otherwise: `"com.liferay.util.service.ServiceProps"`

`resourcesDir` | The first `resources` directory of the `main` source set (by default: `src/main/resources`). `springFile` |

If `osgiModule` is true: the file `META-INF/spring/module-spring.xml` in the first `resources` directory of the `main` source set (by default: `src/main/resources/META-INF/spring/module-spring.xml`)

Otherwise: the file `META-INF/portlet-spring.xml` in the first `resources` directory of the `main` source set (by default: `src/main/resources/META-INF/portlet-spring.xml`)

`sqlDir` |

If the `war` plugin is applied: `${project.webAppDir}/WEB-INF/sql`

Otherwise: The directory `META-INF/sql` in the first `resources` directory of the `main` source set (by default: `src/main/resources/META-INF/sql`).

In the typical scenario of a data-driven Liferay OSGi application split in `myapp-app`, `myapp-service` and `myapp-web` modules, the `service.xml` file is usually contained in the root directory of `myapp-service`. In the `build.gradle` of the same module, it is enough to apply the `com.liferay.service.builder` plugin as described, and then add the following snippet to enable the use of Liferay Service Builder:

```
buildService {
    apiDir = "../myapp-api/src/main/java"
    testDir = "../myapp-test/src/testIntegration/java"
}
```

While apiDir is required, the testDir property assignment can be left out, in which case Arquillian-based integration test classes are generated.

BuildServiceTask

Tasks of type BuildWSDDTask extend JavaExec, so all its properties and methods, such as args and maxHeapSize are available. They also have the following properties set by default:

Property Name | Default Value args | Service Builder command line arguments classpath | project.configurations.serviceBuilder main | "com.liferay.portal.tools.service.builder.ServiceBuilder" systemProperties | ["file.encoding": "UTF-8"]

Task Properties Property Name | Type | Default Value | Description apiDir | File | null | A directory where the service API Java source files are generated. It sets the service.api.dir argument. autoImportDefaultReferences | boolean | true | Whether to automatically add default references, like com.liferay.portal.ClassName, com.liferay.portal.Resource and com.liferay.portal.User, to the services. It sets the service.auto.import.default.references argument. autoNamespaceTables | boolean | true | Whether to prefix table names by the namespace specified in the service.xml file. It sets the service.auto.namespace.tables argument. beanLocatorUtil | String | "com.liferay.util.bean.PortletBeanLocatorUtil" | The fully qualified class name of a bean locator class to use in the generated service classes. It sets the service.bean.locator.util argument. buildNumber | long | 1 | A specific value to assign the build.number property in the service.properties file. It sets the service.build.number argument. buildNumberIncrement | boolean | true | Whether to automatically increment the build.number property in the service.properties file by one at every service generation. It sets the service.build.number.increment argument. databaseNameMaxLength | int | 30 | The upper bound for database table and column name lengths to ensure it works on all databases. It sets the service.database.name.max.length argument. hbmFile | File | null | A Hibernate Mapping file to generate. It sets the service.hbm.file argument. implDir | File | null | A directory where the service Java source files are generated. It sets the service.impl.dir argument. inputFile | File | null | The project's service.xml file. It sets the service.input.file argument. modelHintsConfigs | Set | ["classpath*:META-INF/portal-model-hints.xml", "META-INF/portal-model-hints.xml", "classpath*:META-INF/ext-model-hints.xml", "classpath*:META-INF/portlet-model-hints.xml"] | Paths to the model hints files for Liferay Service Builder to use in generating the service layer. It sets the service.model.hints.configs argument. modelHintsFile | File | null | A model hints file for the project. It sets the service.model.hints.file argument. osgiModule | boolean | false | Whether to generate the service layer for OSGi modules. It sets the service.osgi.module argument. pluginName | String | null | If specified, a plugin can enable additional generation features, such as Clp class generation, for non-OSGi modules. It sets the service.plugin.name argument. propsUtil | String | null | The fully qualified class name of the service properties util class to generate. It sets the service.props.util argument. readOnlyPrefixes | Set | ["fetch", "get", "has", "is", "load", "reindex", "search"] | Prefixes of methods to consider read-only. It sets the service.read.only.prefixes argument. resourceActionsConfigs | Set | ["META-INF/resource-actions/default.xml", "resource-actions/default.xml"] | Paths to the resource actions files for Liferay Service Builder to use in generating the service layer. It sets the service.resource.actions.configs argument. resourcesDir | File | null | A directory where the service non-Java files are generated. It sets the service.resources.dir argument. springFile | File | null | A service Spring file to generate. It sets the service.spring.file argument. springNamespaces | Set | ["beans"] | Namespaces of Spring XML Schemas to add to the service Spring file. It sets the service.spring.namespaces argument. sqlDir | File | null | A directory where the SQL files are generated. It sets the service.sql.dir argument. sqlFileName | String | "tables.sql" | A name (relative to sqlDir) for the file in which the SQL table creation instructions are generated. It sets the service.sql.file argument. sqlIndexesFileName | String | "indexes.sql" |

A name (relative to sqlDir) for the file in which the SQL index creation instructions are generated. It sets the `service.sql.indexes.file` argument. `sqlSequencesFileName` | `String` | `"sequences.sql"` | A name (relative to sqlDir) for the file in which the SQL sequence creation instructions are generated. It sets the `service.sql.sequences.file` argument. `targetEntityName` | `String` | `null` | If specified, it's the name of the entity for which Liferay Service Builder should generate the service. It sets the `service.target.entity.name` argument. `testDir` | `File` | `null` | If specified, it's a directory where integration test Java source files are generated. It sets the `service.test.dir` argument. `uadDir` | `File` | `null` | A directory where the UAD (user-associated data) Java source files are generated. It sets the `service.uad.dir` argument. `uadTestIntegrationDir` | `File` | `null` | A directory where integration test UAD (user-associated data) Java source files are generated. It sets the `service.uad.test.integration.dir` argument.

The properties of type `File` supports any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `String` properties, to defer evaluation until task execution.

Additional Configuration

There are additional configurations that can help you use Service Builder.

Liferay Service Builder Dependency

By default, the plugin creates a configuration called `serviceBuilder` and adds a dependency to the latest released version of Liferay Service Builder. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `serviceBuilder` configuration:

```
dependencies {
    serviceBuilder group: "com.liferay", name: "com.liferay.portal.tools.service.builder", version: "1.0.292"
}
```

If you're applying the `com.liferay.gradle.plugins` or `com.liferay.gradle.plugins.workspace` plugins to your project, the Service Builder dependency is already added to the `serviceBuilder` configuration. Therefore, if you try to apply a customized version of Service Builder, it's not recognized; you must override the configuration already applied.

To do this, you must customize the classpath of the `buildService` task. If you're supplying the customized Service Builder plugin through a module named `custom-sb-api`, you could modify the `buildService` task like this:

```
buildService {
    apiDir = "../custom-sb-api/src/main/java"
    classpath = configurations.serviceBuilder.filter { file -> !file.name.contains("com.liferay.portal.tools.service.builder") }
}
```

If you do this in conjunction with the `serviceBuilder` dependency configuration, the custom Service Builder version is used.

131.19 Source Formatter Gradle Plugin

The Source Formatter Gradle plugin lets you format project files using the Liferay Source Formatter tool.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.source.formatter", version: "2.3.413"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.source.formatter"
```

Since the plugin automatically resolves the Liferay Source Formatter library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds two tasks to your project:

Name | Depends On | Type | Description `checkSourceFormatting` | - | `FormatSourceTask` | Runs the Liferay Source Formatter to check for source formatting errors. `formatSource` | - | `FormatSourceTask` | Runs the Liferay Source Formatter to format the project files.

If desired, it is possible to check for source formatting errors while executing the `check` task by adding the following dependency:

```
check {
    dependsOn checkSourceFormatting
}
```

The same can be achieved by adding the following snippet to the `build.gradle` file in the root directory of a *Liferay Workspace*:

```
subprojects {
    afterEvaluate {
        if (plugins.hasPlugin("base") && plugins.hasPlugin("com.liferay.source.formatter")) {
            check.dependsOn checkSourceFormatting
        }
    }
}
```

The tasks `checkSourceFormatting` and `formatSource` are automatically skipped if another task with the same name is being executed in a parent project.

FormatSourceTask

Tasks of type `FormatSourceTask` extend `JavaExec`, so all its properties and methods, like `args` and `maxHeapSize` are available. They also have the following properties set by default:

Property Name | Default Value `args` | Source Formatter command line arguments `classpath` | `project.configurations.sourceFormatter` `main` | `"com.liferay.source.formatter.SourceFormatter"`

Task Properties Property Name | Type | Default Value | Description `autoFix` | `boolean` | `false` | Whether to automatically fix source formatting errors. It sets the `source.auto.fix` argument. `baseDir` | `File` | | The Source Formatter base directory. It sets the `source.base.dir` argument. *(Read-only)* `baseDirName` | `String` | `"./"` | The name of the Source Formatter base directory, relative to the project directory. `fileExtensions` | `List<String>` | `[]` | The file extensions to format. If empty, all file extensions will be formatted. It sets the `source.file.extensions` argument. `files` | `List<File>` | | The list of files to format. It sets the `source.files` argument. *(Read-only)* `fileNames` | `List<String>` | `null` | The file names to format, relative to the project directory. If `null`, all files contained in `baseDir` will be formatted. `formatCurrentBranch` | `boolean` | `false` | Whether to format only the files contained in `baseDir` that are added or modified in the current Git branch. It sets the `format.current.branch` argument. `formatLatestAuthor` | `boolean` | `false` | Whether to format only the files contained in `baseDir` that are added or modified in the latest Git commits of the same author. It sets the `format.latest.author` argument. `formatLocalChanges` | `boolean` | `false` | Whether to format only the unstaged files contained in `baseDir`. It sets the `format.local.changes` argument. `gitWorkingBranchName` | `String` | `"master"` | The Git working branch name. It sets the `git.working.branch.name` argument. `includeSubrepositories` | `boolean` | `false` | Whether to format files that are in read-only subrepositories. It sets the `include.subrepositories` argument. `maxLineLength` | `int` | `80` | The maximum number of characters allowed in Java files. It sets the `max.line.length` argument. `printErrors` | `boolean` | `true` | Whether to print formatting errors on the Standard Output stream. It sets the `source.print.errors` argument. `processorThreadCount` | `int` | `5` | The number of threads used by Source Formatter. It sets the `processor.thread.count` argument. `showDebugInformation` | `boolean` | `false` | Whether to show debug information, if present. It sets the `show.debug.information` argument. `showDocumentation` | `boolean` | `false` | Whether to show the documentation for the source formatting issues, if present. It sets the `show.documentation` argument. `showStatusUpdates` | `boolean` | `false` | Whether to show status updates during source formatting, if present. It sets the `show.status.updates` argument. `throwException` | `boolean` | `false` | Whether to fail the build if formatting errors are found. It sets the `source.throw.exception` argument.

Additional Configuration

There are additional configurations that can help you use the Source Formatter.

Liferay Source Formatter Dependency

By default, the plugin creates a configuration called `sourceFormatter` and adds a dependency to the latest released version of Liferay Source Formatter. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `sourceFormatter` configuration:

```
dependencies {
    sourceFormatter group: "com.liferay", name: "com.liferay.source.formatter", version: "1.0.885"
}
```

System Properties

It is possible to set the default values of the `fileExtensions`, `fileNames`, `formatCurrentBranch`, `formatLatestAuthor`, and `formatLocalChanges` properties for a `FormatSourceTask` task via system properties:

- `-D${task.name}.file.extensions=java,xml`
- `-D${task.name}.file.names=README.markdown,src/main/resources/hello.txt`
- `-D${task.name}.format.current.branch=true`
- `-D${task.name}.format.latest.author=true`
- `-D${task.name}.format.local.changes=true`

For example, run the following Bash command to format only the unstaged files in the project:

```
./gradlew formatSource -DformatSource.format.local.changes=true
```

131.20 Soy Gradle Plugin

The Soy Gradle plugin lets you compile Closure Templates into JavaScript functions. It also lets you use a custom localization mechanism in the generated `.soy.js` files by replacing `goog.getMsg` definitions with a different function call (e.g., `Liferay.Language.get`).

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.soy", version: "3.1.8"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}
```

There are two Soy Gradle plugins you can apply to your project:

- Apply the *Soy Plugin* to compile Closure Templates into JavaScript functions:

  ```
  apply plugin: "com.liferay.soy"
  ```

- Apply the *Soy Translation Plugin* to use a custom localization mechanism in the generated `.soy.js` files:

  ```
  apply plugin: "com.liferay.soy.translation"
  ```

Since the Soy Gradle plugin automatically resolves the Soy library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Soy Plugin

The Soy plugin adds two tasks to your project:

Name | Depends On | Type | Description `buildSoy` | - | `BuildSoyTask` | Compiles Closure Templates into JavaScript functions. `wrapSoyAlloyTemplate` | - `configJSModules` if `com.liferay.js.module.config.generator` is applied - `processResources` if `java` is applied - `transpileJS` if `com.liferay.js.transpiler` is applied | `WrapSoyAlloyTemplateTask` | Wraps the JavaScript functions compiled from Closure Templates into AlloyUI modules.

The plugin also adds the following dependencies to tasks defined by the `java` plugin:

Name | Depends On `classes` | `wrapSoyAlloyTemplate`

The `buildSoy` task is automatically configured with sensible defaults, depending on whether the `java` plugin is applied:

Property Name | Default Value `includes` | `["**/*.soy"]` `source` |

If the `java` plugin is applied: The first `resources` directory of the `main` source set (by default, `src/main/resources`).

Otherwise: `[]`

The `wrapSoyAlloyTemplate` task is **disabled by default**, and it is automatically configured with sensible defaults, depending on whether the `java` plugin is applied:

Property Name | Default Value `enabled` | `false` `includes` | `["**/*.soy.js"]` `source` |

If the `java` plugin is applied: `project.sourceSets.main.output.resourcesDir`

Otherwise: `[]`

Additional Configuration

There are additional configurations that can help you use the Soy library.

Soy Dependency By default, the plugin creates a configuration called soy and adds a dependency to the `2015-04-10` version of the Soy library. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the soy configuration:

```
dependencies {
    soy group: "com.google.template", name: "soy", version: "2015-04-10"
}
```

Soy Translation Plugin

The Soy Translation plugin adds one task to your project:

Name | Depends On | Type | Description `replaceSoyTranslation` | - `configJSModules` if `com.liferay.js.module.config.gene` is applied - `processResources` if `java` is applied - `transpileJS` if `com.liferay.js.transpiler` is applied | `ReplaceSoyTranslationTask` | Replaces `goog.getMsg` definitions with `Liferay.Language.get` calls.

The plugin also adds the following dependencies to tasks defined by the `java` plugin:

Name | Depends On `classes` | `replaceSoyTranslation`

The `replaceSoyTranslation` task is automatically configured with sensible defaults, depending on whether the `java` plugin is applied:

Property Name | Default Value `includes` | `["**/*.soy.js"]` `replacementClosure` | Replaces `goog.getMsg` definitions with `Liferay.Language.get` calls. `source` |

If the `java` plugin is applied: `project.sourceSets.main.output.resourcesDir`

Otherwise: `[]`

Tasks

BuildSoyTask

Tasks of type `BuildSoyTask` extend `SourceTask`, so all its properties and methods, such as `include` and `exclude`, are available.

Task Properties Property Name | Type | Default Value | Description `classpath` | `FileCollection` | `project.configurations.soy` | The classpath for executing the Liferay Portal Tools Soy Builder.

WrapSoyAlloyTemplateTask

Tasks of type `WrapSoyAlloyTemplateTask` extend `SourceTask`, so all its properties and methods, such as `include` and `exclude`, are available.

Task Properties Property Name | Type | Default Value | Description `moduleName` | `String` | `null` | The name of the AlloyUI module. `namespace` | `String` | `null` | The namespace of the Closure Templates of the project.

It is possible to use Closures and Callables as values for the `String` properties to defer evaluation until task execution.

ReplaceSoyTranslationTask

The `ReplaceSoyTranslationTask` task type finds all the `goog.getMsg` definitions in the project's files and replaces them with a custom function call.

```
var MSG_EXTERNAL_123 = goog.getMsg('welcome-to-{$releaseInfo}', { 'releaseInfo': opt_data.releaseInfo });
```

A `goog.getMsg` definition looks like the example above, and it has the following components:

- *variable name*: `MSG_EXTERNAL_123`
- *language key*: `welcome-to-{$releaseInfo}`
- *arguments object*: `{ 'releaseInfo': opt_data.releaseInfo }`

Tasks of type `ReplaceSoyTranslationTask` extend `SourceTask`, so all its properties and methods, such as `include` and `exclude`, are available.

Task Properties Property Name | Type | Default Value | Description `replacementClosure` | `Closure<String>` | `null` | The Closure invoked in order to get the replacement for `goog.getMsg` definitions. The given Closure is passed the *variable name*, *language key*, and *arguments object* as its parameters.

131.21 Target Platform Gradle Plugin

The Target Platform Gradle plugin helps with building multiple projects against a declared API target platform. Java dependencies can be managed with Maven BOMs and OSGi modules can be resolved against an OSGi distribution.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.target.platform", version: "1.1.13"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}
```

There are two Target Platform Gradle plugins you can apply to your project:

- The *Target Platform Plugin* helps to configure your projects to build against an established set of platform artifacts, including Java and OSGi dependencies.

  ```
  apply plugin: "com.liferay.target.platform"
  ```

- The *Target Platform IDE Plugin* is a superset of the Target Platform Plugin (it applies the above plugin) and also adds IDE integration for searching and debugging source code in the target platform artifacts.

  ```
  apply plugin: "com.liferay.target.platform.ide"
  ```

Since the plugin automatically resolves target platform configurations as dependencies, you must configure a repository that hosts these artifacts. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Target Platform Plugin

The plugin applies the Spring Dependency Management Plugin and then adds several specific configurations to configure the BOMs that are imported to manage Java dependencies and the various artifacts used in resolving OSGi dependencies. Also, a new `resolve` task is added to resolve all OSGi requirements against a declared distribution artifact.

The plugin adds a series of configurations to your project:

Name | Description `targetPlatformBOMs` | Configures all the BOMs to import as managed dependencies. `targetPlatformBundles` | Configures all the bundles in addition to the distro to resolve against. `targetPlatformDistro` | Configures the distro JAR file to use as base for resolving against. `targetPlatformRequirements` | Configures the list of JAR files to use as run requirements for resolving.

The plugin adds a task `resolve` of type `ResolveTask` to your project that performs an OSGi resolve operation using the `targetPlatformRequirements` configuration as the basis of the requirements. The `targetPlatformBundles` configuration is used as a repository for the resolver to resolve requirements. Lastly, the `targetPlatformDistro` configuration is used to provide the *distro* artifact for the resolve process. The *distro* is the artifact that provides all the OSGi capabilities of the target platform. All of these parameters are used to create a `bndrun` file that can be used as input into the Bndrun resolve operation.

Target Platform IDE Plugin

The plugin applies the Target Platform and the `eclipse` plugins to your project, and also adds a special `targetPlatformIDE` configuration, which is used to configure both the `eclipse` and `idea` plugin model in Gradle to add all target platform artifacts to the classpath so they are visible to both Eclipse and IntelliJ's Java Model Search (for looking up sources to classes).

Project Extension

The Target Platform plugin exposes the following properties through the extension named `targetPlatform`:

Property Name | Type | Default Value | Description `ignoreResolveFailures` | `boolean` | `true` | Whether to ignore resolve failures found when executing tasks of type `ResolveTask`. `subprojects` | `Set<Project>` | `project.subprojects` | The subprojects to configure with target platform support, including dependency management and the `resolve` task.

The same extension exposes the following methods:

Method | Description `TargetPlatformExtension applyToConfiguration(Iterable<?> configurationNames)` | Adds additional configurations to configure the BOMs that are imported to manage Java dependencies and the various artifacts used in resolving OSGi dependencies. `TargetPlatformExtension applyToConfiguration(Object... configurationNames)` | Adds additional configurations to configure the BOMs that are imported to manage Java dependencies and the various artifacts used in resolving OSGi dependencies. `TargetPlatformExtension onlyIf(Closure<Boolean> onlyIfClosure)` | Includes a subproject in the target platform configuration if the given closure returns `true`. The closure is evaluated at the end of the subproject configuration phase and is passed a single parameter: the subproject. If the closure returns `false`, the subproject is not included in the target platform configuration `TargetPlatformExtension onlyIf(Spec<Project> onlyIfSpec)` | Includes a subproject in the target platform configuration if the given spec is satisfied. The spec is evaluated at the end of the subproject configuration phase. If the spec is not satisfied, the subproject is not included in the target platform configuration. `TargetPlatformExtension resolveOnlyIf(Closure<Boolean> resolveOnlyIfClosure)` | Includes a subproject in the resolving process (including both the requirements and bundles configuration) if the given closure returns `true`. The closure is evaluated at the end of the subproject configuration phase and is passed a single parameter: the subproject. If the closure returns `false`, the subproject is the resolution process. `TargetPlatformExtension resolveOnlyIf(Spec<Project> resolveOnlyIfSpec)` | Includes a subproject in the resolving platform configuration if the given spec is satisfied. The spec is evaluated at the end of the subproject configuration phase. If the spec is not satisfied, the subproject is not included in the target platform configuration. `TargetPlatformExtension subprojects(Iterable<Project> subprojects)` | Includes additional projects to be configured with Target Platform support. `TargetPlatformExtension subprojects(Project... subprojects)` | Includes additional projects to be configured with Target Platform support.

Tasks

ResolveTask

The purpose of this task is to resolve an OSGi module (or all OSGi modules of subprojects) against the available `targetPlatformBundles` and `targetPlatformDistro` configurations. By default, the `targetPlatformBundles` are all the artifacts created by all the subprojects. The `targetPlatformDistro` must be set explicitly to a valid distribution artifact. When the task is performed, a `bndrun` file is generated using the specified `targetPlatformDistro` as the `-distro` instruction; the `-runrequirements` are a set of `osgi.identity` requirements for the `targetPlatformRequirements` configuration. If the resolve operation is able to find a valid set of `-runbundles` that match the `-runrequirements`, then the task passes successfully (the resolution is valid). If a set of run bundles can't be found, the resolution has failed and the failed requirements are listed as output of the task.

Task Properties Property Name | Type | Default Value | Description `bndrunFile` | `File` | `null` | If this property is specified, it is used as the `bndrun` file to input into the resolver. `bundlesFileCollection` | `FileCollection` | All JAR files of subprojects with `jar` task | The input to `bndrun` resolve operation. `distroFileCollection` | `FileCollection` | `null` | The *distro* parameter for the generated `bndrun` file. `ignoreFailures` | `boolean` | `false` | Whether the `resolve` task should ignore failing the build for resolution errors. `offline` | `boolean` | `null` | Whether to run the bndrun resolve operation in offline mode. `requirementsFileCollection` | `FileCollection` |

For the root project: All the output JAR files of the subprojects.

For subprojects: The output JAR file of the subproject.

| For each resolve operation, the requirements must be specified in the `bndrun` file; each of the JARs in this collection generate an `osgi.identify` requirement in the `bndrun` file.

Additional Configuration

There are additional configurations that you can use to configure the target platform.

Target Platform BOMs Dependency

The plugin creates a configuration called `targetPlatformBOMs` with no defaults. You can use this dependency to set which BOMs to import to configure your target platform.

```
dependencies {
    targetPlatformBOMs group: "com.liferay", name: "com.liferay.ce.portal.bom", version: "7.1.0"
    targetPlatformBOMs group: "com.liferay", name: "com.liferay.ce.portal.compile.only", version: "7.1.0"
}
```

Target Platform Bundles Dependency

The plugin creates a configuration called `targetPlatformBundles`. It is configured with default dependencies to all resolvable bundles in a multi-project build (e.g., all projects in multi-project build that have a `jar` task). This can be used to specify additional bundles that should be added to the set of bundles given to `resolve` task to resolve against when checking for OSGi requirements.

```
dependencies {
    targetPlatformBundles group: "com.google.guava", name: "guava", version: "23.0"
}
```

Target Platform Distro Dependency

The plugin creates a configuration called `targetPlatformDistro`. It is has no default so you must specify which artifact you want to use as the distribution to resolve against.

```
dependencies {
    targetPlatformDistro group: "com.liferay", name: "com.liferay.ce.portal.distro", version: "7.1.0"
}
```

If you have created your own custom distro JAR that is available locally, you can use the `files` method to add it to the configuration.

```
dependencies {
    targetPlatformDistro files("custom-distro.jar")
}
```

Target Platform Requirements Dependency

The plugin creates a configuration called `targetPlatformRequirements`. It is configured with default dependencies to all resolvable bundles in a multi-project build (e.g., all projects in multi-project build that have a `jar` task). This is can be used to specify additional bundles that should be added to the set of bundles given to the `resolve` task to set as `osgi.identity` requirements.

```
dependencies {
    targetPlatformRequirements group: "com.liferay", name: "com.liferay.other.bundle", version: "1.0"
}
```

131.22 Theme Builder Gradle Plugin

The Theme Builder Gradle plugin lets you run the Liferay Theme Builder tool to build the Liferay theme files in your project.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.theme.builder", version: "2.0.7"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.portal.tools.theme.builder"
```

The Theme Builder plugin automatically applies the `war` plugin. It also applies the `com.liferay.css.builder` plugin to compile the Sass files in the theme.

Since the plugin automatically resolves the Liferay Theme Builder library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `buildTheme` | - | `BuildThemeTask` | Builds the theme files.

The plugin also adds the following dependencies to tasks defined by the `com.liferay.css.builder` and `war` plugins:

Name | Depends On `buildCSS` | `buildTheme` `war` | `buildTheme`

The `buildCSS` dependency compiles the Sass files contained in the directory specified by the `buildTheme.outputDir` property. Moreover, the `war` task is configured as follows

- exclude the directory specified in the `buildTheme.diffsDir` property from the WAR file.
- include the files contained in the `buildTheme.outputDir` directory into the WAR file.
- include only the compiled CSS files, not SCSS files, into the WAR file.

The `buildTheme` task is automatically configured with sensible defaults:

Property Name | Default Value `diffsDir` | `project.webAppDir` `outputDir` | `"${project.buildDir}/buildTheme"` `parentFile` | The first JAR file in the `parentThemes` configuration that contains a `META-INF/resources/${buildTheme.parentName}` directory, or the first WAR file in the `parentThemes` configuration whose name starts with `${parentName}-theme-`. `parentName` | `"_styled"` `templateExtension` | `"ftl"` `themeName` | `project.name` `unstyledFile` | The first JAR file in the `parentThemes` configuration that contains a `META-INF/resources/_unstyled` directory.

BuildThemeTask

Tasks of type `BuildThemeTask` extend `JavaExec`, so all its properties and methods, such as `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `args` | Theme Builder command line arguments `classpath` | `project.configurations.themeBuilder` `main` | `"com.liferay.portal.tools.theme.builder.ThemeBuilder"`

Task Properties Property Name | Type | Default Value | Description `diffsDir` | `File` | `null` | The directory that contains the files to copy over the parent theme. It sets the `--diffs-dir` argument. `outputDir` | `File` | `null` | The directory where to build the theme. It sets the `--output-dir` argument. `parentDir` | `File` | `null` | The directory of the parent theme. It sets the `--parent-path` argument. `parentFile` | `File` | `null` | The JAR file of the parent theme. If `parentDir` is specified, this property has no effect. It sets the `--parent-path` argument. `parentName` | `String` | `null` | The name of the parent theme. It sets the `--parent-name` argument. `templateExtension` | `String` | `null` | The extension of the template files, usually `"ftl"` or `"vm"`. It sets the `--template-extension` argument. `themeName` | `String` | `null` | The name of the new theme. It sets the `--name` argument. `unstyledDir` | `File` | `null` | The directory of Liferay Frontend Theme Unstyled. It sets the `--unstyled-dir` argument. `unstyledFile` | `File` | `null` | The JAR file of Liferay Frontend Theme Unstyled. If `unstyledDir` is specified, this property has no effect. It sets the `--unstyled-dir` argument.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `String` properties to defer evaluation until task execution.

Additional Configuration

There are additional configurations that can help you use the CSS Builder.

Liferay Theme Builder Dependency

By default, the plugin creates a configuration called `themeBuilder` and adds a dependency to the latest released version of the Liferay Theme Builder. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `themeBuilder` configuration:

```
dependencies {
    themeBuilder group: "com.liferay", name: "com.liferay.portal.tools.theme.builder", version: "1.1.7"
}
```

Parent Theme Dependencies

By default, the plugin creates a configuration called `parentThemes` and adds dependencies to the latest released versions of the Liferay Frontend Theme Styled, Liferay Frontend Theme Unstyled, and Liferay Frontend

Theme Classic artifacts. It is possible to override this setting and use a specific version of the artifacts by manually adding dependencies to the `parentThemes` configuration. For example,

```
dependencies {
    parentThemes group: "com.liferay", name: "com.liferay.frontend.theme.styled", version: "VERSION"
    parentThemes group: "com.liferay", name: "com.liferay.frontend.theme.unstyled", version: "VERSION"
    parentThemes group: "com.liferay.plugins", name: "classic-theme", version: "VERSION"
}
```

Specifying dependency versions is not required when leveraging workspace's Target Platform functionality. All dependencies with the group ID `com.liferay` or `com.liferay.portal` are automatically set when targeting a platform. For external theme dependencies (e.g., `classic-theme` with the group ID `com.liferay.plugins`), you can find the version used by your specific Liferay DXP instance by leveraging the Gogo shell. In a Gogo shell prompt, execute the following command:

```
lb -s theme
```

This lists the deployed theme bundles and their versions. Extract the versions for the theme dependencies you want to leverage and add them to your configuration.

131.23 TLDDoc Builder Gradle Plugin

The TLDDoc Builder Gradle plugin lets you run the Tag Library Documentation Generator tool in order to generate documentation for the JSP Tag Library Descriptor (TLD) files in your project.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.tlddoc.builder", version: "1.3.3"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}
```

There are two TLDDoc Builder Gradle plugins you can apply to your project:

- Apply the *TLDDoc Builder Plugin* to generate tag library documentation for your project:

  ```
  apply plugin: "com.liferay.tlddoc.builder"
  ```

- Apply the *App TLDDoc Builder Plugin* in a parent project to generate the tag library documentation as a single, combined HTML document for an application that spans different subprojects, each one representing a different component of the same application:

  ```
  apply plugin: "com.liferay.app.tlddoc.builder"
  ```

Since the plugin automatically resolves the Tag Library Documentation Generator library as a dependency, you must configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

TLDDoc Builder Plugin

The plugin adds three tasks to your project:

Name | Depends On | Type | Description `copyTLDDocResources` | - | `Copy` | Copies the tag library documentation resources from `src/main/tlddoc` to the destination directory of the `tlddoc` task. `tlddoc` | `copyTLDDocResources`, `validateTLD` | `TLDDocTask` | Generates the tag library documentation. `validateTLD` | - | `ValidateSchemaTask` | Validates the TLD files in the project.

The `tlddoc` task is automatically configured with sensible defaults, depending on whether the `java` plugin is applied:

Property Name | Default Value with the `java` plugin `destinationDir` | `${project.docsDir}/tlddoc` `includes` | `["**/*.tld"]` `source` | `project.sourceSets.main.resources.srcDirs`

The `validateTLD` task is also automatically configured with sensible defaults, depending on whether the `java` plugin is applied:

Property Name | Default Value `includes` |

If the `java` plugin is applied: `["**/*.tld"]`

Otherwise: `[]`

`source` |

If the `java` plugin is applied: `project.sourceSets.main.resources.srcDirs`

Otherwise: `null`

By default, the `tlddoc` task generates the documentation for all the TLD files that are found in the resources directories of the `main` source set. The documentation files are saved in `build/docs/tlddoc` and include the files copied from `src/main/tlddoc`.

The `copyTLDDocResources` task lets you add references to images and other resources directly in the TLD files. For example, if the project includes an image called `breadcrumb.png` in the `src/main/tlddoc/images` directory, you can reference it in a TLD file contained in the `src/main/resources` directory:

```
<description>Hello World <![CDATA[<img src="../images/breadcrumb.png"]]></description>
```

App TLDDoc Builder Plugin

In order to use the App TLDDoc Builder plugin, it is required to apply the `com.liferay.app.tlddoc.builder` plugin in a parent project (that is, a project that is a common ancestor of all the subprojects representing the various components of the app). It is also required to apply the `com.liferay.tlddoc.builder` plugin to all the subprojects that contain TLD files.

The App TLDDoc Builder plugin automatically applies the `base` plugin. It also adds three tasks to your project:

Name | Depends On | Type | Description `appTLDDoc` | `copyAppTLDDocResources`, the `validateTLD` tasks of the subprojects | `TLDDocTask` | Generates tag library documentation for the app. `copyAppTLDDocResources` | - | `Copy` | Copies the tag library documentation resources defined as inputs for the `copyTDLDocResources` tasks of the subprojects, aggregating them into the destination directory of the `appTLDDoc` task. `jarAppTLDDoc` | `appTLDDoc` | `Jar` | Assembles a JAR archive containing the tag library documentation files for this app.

The `appTLDDoc` task is automatically configured with sensible defaults:

Property Name | Default Value `destinationDir` | `${project.buildDir}/docs/tlddoc` `source` | The sum of all the `tlddoc.source` values of the subprojects

Project Extension

The App TLDDoc Builder plugin exposes the following properties through the extension named `appTLDDocBuilder`:

Property Name | Type | Default Value | Description `subprojects` | `Set<Project>` | `project.subprojects` | The subprojects to include in the tag library documentation of the app.

The same extension exposes the following methods:

Method | Description `AppTLDDocBuilderExtension subprojects(Iterable<Project> subprojects)` | Include additional projects in the tag library documentation of the app. `AppTLDDocBuilderExtension subprojects(Project... subprojects)` | Include additional projects in the tag library documentation of the app.

Tasks

TLDDocTask

Tasks of type `TLDDocTask` extend `JavaExec`, so all its properties and methods, such as `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `args` | Tag Library Documentation Generator command line arguments `classpath` | `project.configurations.tlddoc` `main` | `"com.sun.tlddoc.TLDDoc"` `maxHeapSize` | `"256m"`

The `TLDDocTask` class is also very similar to `SourceTask`, which means it provides a `source` property and lets you specify include and exclude patterns.

Task Properties Property Name | Type | Default Value | Description `destinationDir` | `File` | `null` | The directory where the tag library documentation files are saved. `excludes` | `Set<String>` | `[]` | The TLD file patterns to exclude. `includes` | `Set<String>` | `[]` | The TLD file patterns to include. `source` | `FileTree` | `[]` | The TLD files to generate documentation for, after the include and exclude patterns have been applied. `xsltDir` | `File` | `null` | The directory that contains the custom XSLT stylesheets used by the Tag Library Documentation Generator to produce the final documentation files. It sets the `-xslt` argument.

The properties of type `File` support any type that can be resolved by `project.file`.

Task Methods The methods available for `TLDDocTask` are exactly the same as the one defined in the `SourceTask` class.

ValidateSchemaTask

Tasks of type `ValidateSchemaTask` extend `SourceTask`, so all its properties and methods, such as `include` and `exclude`, are available.

Tasks of this type invoke the `schemavalidate` Ant task in order to validate XML files described by an XML schema.

Task Properties Property Name | Type | Default Value | Description `dtdDisabled` | `boolean` | `false` | Whether to disable DTD support. `fullChecking` | `boolean` | `true` | Whether to enable full schema checking. `lenient` | `boolean` | `false` | Whether to only check if the XML document is well-formed. `xmlParserClassName` | `String` |

null | The class name of the XML parser to use. xmlParserClasspath | FileCollection | null | The classpath with the XML parser.

It is possible to use Closures and Callables as values for the String properties to defer evaluation until task execution.

Additional Configuration

There are additional configurations that can help you use the TLDDoc Builder.

Tag Library Documentation Generator Dependency

By default, the plugin creates a configuration called tlddoc and adds a dependency to the 1.3 version of the Tag Library Documentation Generator. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the tlddoc configuration:

```
dependencies {
    tlddoc group: "taglibrarydoc", name: "tlddoc", version: "1.3"
}
```

131.24 TLD Formatter Gradle Plugin

The TLD Formatter Gradle plugin lets you format a project's TLD files using the Liferay TLD Formatter tool.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.tld.formatter", version: "1.0.9"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.tld.formatter"
```

Since the plugin automatically resolves the Liferay TLD Formatter library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description formatTLD | - | FormatTLDTask | Runs the Liferay TLD Formatter to format files.

FormatTLDTask

Tasks of type `FormatTLDTask` extend `JavaExec`, so all its properties and methods, such as `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `args` | TLD Formatter command line arguments `classpath` | `project.configurations.tldFormatter` `main` | `"com.liferay.tld.formatter.TLDFormatter"`

Task Properties Property Name | Type | Default Value | Description `plugin` | `boolean` | `true` | Whether to format all the TLD files contained in the `workingDir` directory. If `false`, all `liferay-portlet-ext.tld` files are ignored. It sets the `tld.plugin` argument.

Additional Configuration

There are additional configurations that can help you use the TLD Formatter.

Liferay TLD Formatter Dependency

By default, the plugin creates a configuration called `tldFormatter` and adds a dependency to the latest released version of Liferay TLD Formatter. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `tldFormatter` configuration:

```
dependencies {
    tldFormatter group: "com.liferay", name: "com.liferay.tld.formatter", version: "1.0.5"
}
```

131.25 Whip Gradle Plugin

The Whip Gradle plugin lets you use the Liferay Whip library to ensure that unit tests fully cover your project's code. See here for a usage sample.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.whip", version: "1.0.7"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.whip"
```

Since the plugin automatically resolves the Liferay Whip library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

By default, Whip is automatically applied to all tasks of type `Test`. If a task has Whip applied and Whip is enabled, then Whip is configured as a Java Agent.

Project Extension

The Whip Gradle plugin exposes the following properties through the extension named `whip`:

Property Name | Type | Default Value | Description `version` | `String` | `latest.release` | The version of the Liferay Whip library to use.

The same extension exposes the following methods:

Method | Description `void applyTo(Task task)` | Applies Whip to a task. The task instance must implement the `JavaForkOptions` interface.

Task Extension

If Whip is applied, the following task properties are available through the extension named `whip`:

Property Name | Type | Default Value | Description `dataFile` | `File` | `test-coverage/whip.dat` | `enabled` | `boolean` | `true` | Whether to configure Whip as a Java Agent. `excludes` | `List<String>` | `[]` | The class name patterns to exclude when checking for unit test code coverage. For example, a value could be `['.*Test', '.*Test\\$.*', '.*\\$Proxy.*', 'com/liferay/whip/.*']`. `includes` | `List<String>` | `[]` | The class name patterns to include when checking for unit test code coverage. `instrumentDump` | `boolean` | `false` | `whipJarFile` | `File` | The first file in the `whip` configuration whose name starts with `com.liferay.whip-`. | The Whip JAR file.

The same extension exposes the following methods:

Method | Description `WhipTaskExtension excludes(Iterable<Object> excludes)` | Adds class name patterns to exclude when checking for unit test coverage. `WhipTaskExtension excludes(Object... excludes)` | Adds class name patterns to exclude when checking for unit test coverage. `WhipTaskExtension includes(Iterable<Object> includes)` | Adds class name patterns to include when checking for unit test coverage. `WhipTaskExtension includes(Object... includes)` | Adds class name patterns to include when checking for unit test coverage.

Additional Configuration

There are additional configurations that can help you use Whip.

Liferay Whip Dependency

By default, the Whip Gradle plugin creates a configuration called `whip` and adds a dependency to the version of Liferay Whip configured in the `whip.version` extension property. It is possible to override this setting and use a specific version of the library by manually adding a dependency to the `whip` configuration:

```
dependencies {
    whip group: "com.liferay", name: "com.liferay.whip", version: "1.0.1"
}
```

In order to leverage the sensible default of the `whip.whipJarFile` task property, the name of the dependency must be `com.liferay.whip`. Otherwise, it will be necessary to set the value of the `whip.whipJarFile` property manually.

131.26 WSDD Builder Gradle Plugin

The WSDD Builder Gradle plugin lets you run the Liferay WSDD Builder tool to generate the Apache Axis Web Service Deployment Descriptor (WSDD) files from a Service Builder `service.xml` file.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.wsdd.builder", version: "1.0.13"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.portal.tools.wsdd.builder"
```

The WSDD Builder plugin automatically applies the `java` plugin.

Since the plugin automatically resolves the Liferay WSDD Builder library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `buildWSDD` | `compileJava` | `BuildWSDDTask` | Runs the Liferay WSDD Builder.

By default, the `buildWSDD` task uses the `${project.projectDir}/service.xml` file as input. Then, it generates `${project.projectDir}/server-config.wsdd` and the `*_deploy.wsdd` and `*_undeploy.wsdd` files in the first `resources` directory of the `main` source set (by default: `src/main/resources`).

If the `war` plugin is applied, the task uses `${project.webAppDir}/WEB-INF/service.xml` as input to generate `${project.webAppDir}/WEB-INF/server-config.wsdd`. The `*_deploy.wsdd` and `*_undeploy.wsdd` files are still generated in the first resources directory of the `main` source set.

Liferay WSDD Build Service requires an additional classpath (configured with the `buildWSDD.builderClasspath` property), to correctly generate the WSDD files. The `buildWSDD` task uses the following default value, which creates an implicit dependency to the `compileJava` task:

```
tasks.compileJava.outputs.files + sourceSets.main.compileClasspath + sourceSets.main.runtimeClasspath
```

BuildWSDDTask

Tasks of type `BuildWSDDTask` extend `JavaExec`, so all its properties and methods, such as `args` and `maxHeapSize`, are available. They also have the following properties set by default:

Property Name | Default Value `args` | WSDD Builder command line arguments `classpath` | `project.configurations.wsddBuilder` `main` | `"com.liferay.portal.tools.wsdd.builder.WSDDBuilder"`

Task Properties Property Name | Type | Default Value | Description `builderClasspath` | `String` | `null` | A classpath that the Liferay WSDD Builder uses to generate WSDD files. It sets the `wsdd.class.path` argument. `inputFile` | `File` | `null` | A `service.xml` from which to generate the WSDD files. It sets the `wsdd.input.file` argument. `outputDir` | `File` | `null` | A directory where the `*_deploy.wsdd` and `*_undeploy.wsdd` files are generated. It sets the `wsdd.output.path` argument. `serverConfigFile` | `File` | `${project.projectDir}/server-config.wsdd` | A `server-config.wsdd` file to generate. It sets the `wsdd.server.config.file` argument. `serviceNamespace` | `String` | `"Plugin"` | A namespace for the WSDD Service. It sets the `wsdd.service.namespace` argument.

The properties of type `File` support any type that can be resolved by `project.file`. Moreover, it is possible to use Closures and Callables as values for the `String` properties, to defer evaluation until task execution.

Additional Configuration

There are additional configurations that can help you use the WSDD Builder.

Liferay WSDD Builder Dependency

By default, the plugin creates a configuration called `wsddBuilder` and adds a dependency to the latest released version of the Liferay WSDD Builder. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `wsddBuilder` configuration:

```
dependencies {
    wsddBuilder group: "com.liferay", name: "com.liferay.portal.tools.wsdd.builder", version: "1.0.10"
}
```

131.27 WSDL Builder Gradle Plugin

The WSDL Builder Gradle plugin lets you generate Apache Axis client stubs from Web Service Description (WSDL) files.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.wsdl.builder", version: "2.0.3"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.wsdl.builder"
```

The WSDL Builder plugin automatically applies the java plugin.

Since the plugin automatically resolves the Apache Axis library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one main task to your project:

Name | Depends On | Type | Description `buildWSDL` | - | `BuildWSDLTask` | Generates WSDL client stubs.

By default, the `buildWSDL` task looks for WSDL files in the `${project.projectDir}/wsdl` directory. If the `war` plugin is applied, it looks in the `${project.webAppDir}/WEB-INF/wsdl` directory.

For each WSDL file that can be found, the task generates client stubs via direct invocation of the *WSDL2Java* tool, saving them in the first `java` directory of the `main` source set (by default: `src/main/java`).

If configured to do so, `buildWSDL` can instead save the client stub Java files in a temporary directory, compile them, and package them in JAR files. The JAR files are named after the WSDL file and saved in `${project.projectDir}/lib`, by default, or in `${project.webAppDir}/WEB-INF/lib`, if the `war` plugin is applied.

BuildWSDLTask

Tasks of type `FormatWSDLTask` extend `SourceTask`, so all its properties and methods, such as `include` and `exclude`, are available.

Task Properties Property Name | Type | Default Value | Description `buildLibs` | `boolean` | `true` | Whether to package the client stub classes of each WSDL file in JAR files, saved to the directory the `destinationDir` property references. If `false`, the task generates the client stub Java files to the `destinationDir` directory. `destinationDir` | `File` | `null` | A directory where the client stub Java files (if `buildLibs` is `false`) or the client stub JAR files (if `buildLibs` is `true`) are saved. `generateOptions.mapping` | `Map` | `[:]` | Namespace-to-package mappings (sets the `--NStoPkg` argument in the *WSDL2Java* invocation). It is possible to use a `Closure` or a `Callable`, to defer evaluation until task execution.. `generateOptions.noWrapped` | `boolean` | `false` | Whether to turn off support for "wrapped" document/literal (sets the `--noWrapped` argument in the *WSDL2Java* invocation). `generateOptions.serverSide` | `boolean` | `false` | Whether to emit server-side bindings for the web service (sets the `--server-side` argument in the *WSDL2Java* invocation). `generateOptions.verbose` | `boolean` | `false` | Whether to print informational messages (sets the `--verbose` argument in the *WSDL2Java* invocation). `includeSource` | `boolean` | `true` | Whether to package the client stub Java files in the JAR file's `OSGI-OPT/src` directory. If `buildLibs` is `false`, this property has no effect. `includeWSDLs` | `boolean` | `true` | Whether to configure the `processResources` task to include the WSDL files in the project JAR's `wsdl` directory.

The properties of type `File` support any type that can be resolved by `project.file`.

Task Methods Method Signature | Description `generateOptions.mapping(Object namespace, Object packageName)` | Adds a namespace-to-package mapping. `generateOptions.mappings(Map mappings)` | Adds multiple namespace-to-package mappings.

Helper Tasks At the end of the project evaluation, a series of helper tasks are created for each WSDL file returned by the `source` property of the `BuildWSDLTask` tasks. The names of the helper tasks start with the WSDL file name, without any extension.

- `${WSDL file title}Generate` of type `JavaExec`: invokes *WSDL2Java* to generate the client stubs for the WSDL file.

If `buildWSDLTask.buildLibs` is `true`, the following helper tasks are also created:

- `${WSDL file title}Compile` of type `JavaCompile`: compiles the client stub Java files for the WSDL file.
- `${WSDL file title}Jar` of type `Jar`: packages in a JAR file called `${WSDL file title}-ws.jar`, the client stub for the WSDL file.

Additional Configuration

There are additional configurations that can help you use WSDL Builder.

Apache Axis Dependency

By default, the plugin creates a configuration called `wsdlBuilder` and adds the following dependencies:

- `axis:axis-wsdl4j:1.5.1`
- `com.liferay:org.apache.axis:1.4.LIFERAY-PATCHED-1`
- `commons-discovery:commons-discovery:0.2`
- `commons-logging:commons-logging:1.0.4`
- `javax.activation:activation:1.1`
- `javax.mail:mail:1.4`
- `org.apache.axis:axis-jaxrpc:1.4`
- `org.apache.axis:axis-saaj:1.4`

It is possible to override this setting and use a specific version of Apache Axis, by manually populating the `wsdlBuilder` configuration with the desired dependencies.

131.28 XML Formatter Gradle Plugin

The XML Formatter Gradle plugin lets you format a project's XML files using the Liferay XML Formatter tool.
The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.xml.formatter", version: "1.0.11"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.xml.formatter"
```

Since the plugin automatically resolves the Liferay XML Formatter library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds one task to your project:

Name | Depends On | Type | Description `formatXML` | - | `FormatXMLTask` | Runs the Liferay XML Formatter to format the project files.

If the `java` plugin is applied, the task formats XML files contained in the `resources` directories of the `main` source set (by default: `src/main/resources/**/*.xml`).

FormatXMLTask

Tasks of type `FormatXMLTask` extend `SourceTask`, so all its properties and methods, such as `include` and `exclude`, are available.

Task Properties Property Name | Type | Default Value | Description `classpath` | `FileCollection` | `project.configurations.xmlFormatter` | The classpath for executing the main class. `mainClassName` | `String` | `"com.liferay.xml.formatter.XMLFormatter"` | The fully qualified name of the XML Formatter Main class. `stripComments` | `boolean` | `false` | Whether to remove all the comments from the XML files. It sets the `xml.formatter.strip.comments` argument.

Additional Configuration

There are additional configurations that can help you use the XML Formatter.

Liferay XML Formatter Dependency

By default, the plugin creates a configuration called `xmlFormatter` and adds a dependency to the latest released version of the Liferay XML Formatter. It is possible to override this setting and use a specific version of the tool by manually adding a dependency to the `xmlFormatter` configuration:

```
dependencies {
    xmlFormatter group: "com.liferay", name: "com.liferay.xml.formatter", version: "1.0.5"
}
```

131.29 XSD Builder Gradle Plugin

The XSD Builder Gradle plugin lets you generate Apache XMLBeans bindings from XML Schema (XSD) files.

The plugin has been successfully tested with Gradle 4.10.2.

Usage

To use the plugin, include it in your build script:

```
buildscript {
    dependencies {
        classpath group: "com.liferay", name: "com.liferay.gradle.plugins.xsd.builder", version: "1.0.7"
    }

    repositories {
        maven {
            url "https://repository-cdn.liferay.com/nexus/content/groups/public"
        }
    }
}

apply plugin: "com.liferay.xsd.builder"
```

The XSD Builder plugin automatically applies the `java` plugin.

Since the plugin automatically resolves the Liferay Service Builder library as a dependency, you have to configure a repository that hosts the library and its transitive dependencies. The Liferay CDN repository hosts them all:

```
repositories {
    maven {
        url "https://repository-cdn.liferay.com/nexus/content/groups/public"
    }
}
```

Tasks

The plugin adds three tasks to your project:

Name | Depends On | Type | Description `buildXSD` | `buildXSDCompile` | `BuildXSDTask` | Generates XMLBeans bindings and compiles them in a JAR file. `buildXSDGenerate` | `cleanBuildXSDGenerate` | `JavaExec` | Invokes the XMLBeans Schema Compiler to generate Java types from XML Schema. `buildXSDCompile` | `buildXSDGenerate`, `cleanBuildXSDCompile` | `JavaCompile` | Compiles the generated Java types.

By default, the `buildXSD` task looks for XSD files in the `${project.projectDir}/xsd` directory, and saves the generated JAR file as `${project.projectDir}/lib/${project.archivesBaseName}-xbean.jar`.

If the `war` plugin is applied, the task looks for XSD files in the `${project.webAppDir}/WEB-INF/xsd` directory, and saves the generated JAR file as `${project.webAppDir}/WEB-INF/lib/${project.archivesBaseName}-xbean.jar`.

BuildXSDTask

Tasks of type `BuildXSDTask` extend `Zip`. They also have the following properties set by default:

Property Name | Default Value `appendix` | `"xbean"` `extension` | `"jar"` `version` | `null`

For each task of type `BuildXSDTask`, the following helper tasks are created:

- `${buildXSDTask.name}Compile`
- `${buildXSDTask.name}Generate`

Task Properties Property Name | Type | Default Value | Description `inputDir` | `File` | `null` | A directory containing XSD files from which to generate Apache XMLBeans bindings.

The properties of type `File` support any type that can be resolved by `project.file`.

Additional Configuration

There are additional configurations that can help you use the XSD Builder.

Apache XMLBeans Dependency

By default, the XSD Builder Gradle plugin creates a configuration called `xsdBuilder` and adds a dependency to the 2.5.0 version of Apache XMLBeans. It is possible to override this setting and use a specific version of the library by manually adding a dependency to the `xsdBuilder` configuration:

```
dependencies {
    xsdBuilder group: "org.apache.xmlbeans", name: "xmlbeans", version: "2.6.0"
}
```

131.30 Felix Gogo Shell

To interact with Liferay DXP's module framework on a local server machine, you can use the Felix Gogo shell within Blade CLI.

Here's the command syntax:

```
blade sh <gogoShellCommand>
```

If you're not using Blade CLI, you can start the Gogo shell from a local telnet session.

```
telnet localhost 11311
```

To disconnect the session, execute the `disconnect` command.

Warning: Commands `shutdown`, `close`, and `exit` stop the OSGi framework. So make sure to use the `disconnect` command to end the telnet Gogo Shell session.

Here are some useful Gogo shell commands:

`help`: lists all the available Gogo shell commands. Notice that each command has two parts to its name, separated by a colon. For example, the full name of the `help` command is `felix:help`. The first part is the command scope while the second part is the command function. The scope allows commands with the same name to be disambiguated. E.g., scope allows the `felix:refresh` command to be distinguished from the `equinox:refresh` command.

`help [COMMAND_NAME]`: lists information about a specific command including a description of the command, the scope of the command, and information about any flags or parameters that can be supplied when invoking the command.

`lb`: lists all of the bundles installed in Liferay's module framework. Use the `-s` flag to list the bundles using the bundles' symbolic names.

`b [BUNDLE_ID]`: lists information about a specific bundle including the bundle's symbolic name, bundle ID, data root, registered (provided) and used services, imported and exported packages, and more

`headers [BUNDLE_ID]`: lists metadata about the bundle from the bundle's `MANIFEST.MF` file

`diag [BUNDLE_ID]`: lists information about why the specified bundle is not working (e.g., unresolved dependencies, etc.)

`packages [PACKAGE_NAME]`: lists all of the named package's dependencies

`scr:list`: lists all of the components registered in the module framework (*scr* stands for service component runtime) `scr:info [COMPONENT_NAME]`: lists information about a specific component including the component's description, services, properties, configuration, references, and more.

`services`: lists all of the services that have been registered in Liferay's module framework

`inspect capability service [BUNDLE_ID]`: lists services exposed by a bundle

`install [PATH_TO_JAR_FILE]`: installs the specified bundle into Liferay's module framework

`start [BUNDLE_ID]`: starts the specified bundle

`stop [BUNDLE_ID]`: stops the specified bundle

`uninstall [BUNDLE_ID]`: uninstalls the specified bundle from Liferay's module framework

`system:getproperties`: lists all of the system properties

For more information about the Gogo shell, please visit http://felix.apache.org/documentation/subprojects/apache-felix-gogo.html.

Chapter 132

Maven

Liferay provides plugins that you can apply to your Maven project. This reference documentation describes configuration properties for your Maven project's `pom.xml` for each plugin. If you're looking for instructions on using Maven with your Liferay modules, see the Maven tutorials.

132.1 Bundle Support Plugin

The Bundle Support plugin lets you use Liferay Workspace as a Maven project. For more information on how a Maven Workspace works and the features it provides, see the Maven Workspace tutorial.

Usage

To use the plugin, include it in your project's root `pom.xml` file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.portal.tools.bundle.support</artifactId>
            <version>3.2.5</version>
            <executions>
                <execution>
                    <id>clean</id>
                    <goals>
                        <goal>clean</goal>
                    </goals>
                    <phase>clean</phase>
                    <configuration>
                    </configuration>
                </execution>
                <execution>
                    <id>deploy</id>
                    <goals>
                        <goal>deploy</goal>
                    </goals>
                    <phase>pre-integration-test</phase>
                    <configuration>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        ...
    </plugins>
</build>
```

Goals

The plugin adds five Maven goals to your project:

Name | Description bundle-support:clean | Deletes a file from the `deploy` directory of a Liferay bundle. bundle-support:create-token | Creates a token used to validate your user credentials when downloading a DXP bundle. bundle-support:deploy | Deploys the Maven project to the specified Liferay DXP bundle. bundle-support:dist | Creates a distributable Liferay DXP bundle archive file (e.g., ZIP). bundle-support:init | Downloads and installs the specified Liferay DXP version.

clean Goal's Available Parameters

You can set the following parameters in the `clean` execution's `<configuration>` section of the POM:

Parameter Name | Type | Default Value | Description `liferayHome` | `String` | `bundles` | The directory where your Liferay DXP instance resides. This can be specified from the command line as `-DliferayHome=`. `fileName` | `String` | `${project.artifactId}.${project.packaging}` | The name of the file to delete from your bundle.

create-token Goal's Available Parameters

You can change the default parameter values of the `create-token` goal by creating an `<execution>` section containing `<configuration>` tags. For example,

```
<execution>
    <id>create-token</id>
    <goals>
        <goal>create-token</goal>
    </goals>
    <configuration>
    </configuration>
</execution>
```

You can set the following parameters in the `create-token` execution's `<configuration>` section of the POM:

Parameter Name | Type | Default Value | Description `emailAddress` | `String` | `null` | The email address to use when downloading a DXP bundle. This email address must match the one registered for your DXP subscription. `force` | `boolean` | `false` | Whether to override the existing token with a newly generated one. `password` | `String` | `null` | The password to use when downloading a DXP bundle. This password must match the one registered for your DXP subscription. `passwordFile` | `File` | `null` | The file to hold your password used when downloading a DXP bundle. `tokenFile` | `File` | `${user.home}/.liferay/token` | The file to hold the Liferay bundle authentication token. `tokenUrl` | `URL` | `https://releases-cdn.liferay.com/portal/7.0.6-ga7/liferay-ce-portal-tomcat-7.0-ga7-20180507111753223.zip` | The URL pointing to the bundle Zip to download.

After executing the `create-token` goal, you're prompted for your email address and password, both of which are used to generate your token. It's recommended to configure your email and password from the command line rather than specifying them in your POM file.

deploy Goal's Available Parameters

You can set the following parameters in the `deploy` execution's `<configuration>` section of the POM:

Parameter Name | Type | Default Value | Description `liferayHome` | `String` | `bundles` | The directory where your Liferay DXP instance resides. This can be specified from the command line as `-DliferayHome=`.

deployFile | File | ${project.build.directory}/${project.build.finalName}.${project.packaging} | The packaged file (e.g., JAR) to deploy to the Liferay bundle. outputFileName | String | ${project.artifactId}.${project.packaging | The name of the output file.

dist Goal's Available Parameters

You can change the default parameter values of the dist goal by creating an <execution> section containing <configuration> tags. For example,

```
<execution>
    <id>dist</id>
    <goals>
        <goal>dist</goal>
    </goals>
    <configuration>
    </configuration>
</execution>
```

You can set the following parameters in the dist execution's <configuration> section of the POM:

Parameter Name | Type | Default Value | Description liferayHome | String | bundles | The directory where your Liferay DXP instance resides. This can be specified from the command line as -DliferayHome=. archiveFileName | String | null | The name for the generated archive file. cacheDir | File | ${user.home}/.liferay/bundles | The directory where the downloaded bundle Zip files are stored. configs | String | configs | The directory that contains the configuration files. deployFile | File |${project.build.directory}/${project.build.finalName}.${project.packaging} | The packaged file (e.g., JAR) to deploy to the Liferay bundle. environment | String | ${liferay.workspace.environment} | The environment of your Liferay home deployment. (e.g., common, dev, local, prod, and uat). format | String | zip | The format type to use when packaging the Liferay bundle as an archive. includeFolder | boolean | true | Whether to add a parent folder to the archive. outputFileName | String | ${project.artifactId}.${project.packaging} | The path to the archive file. password | String | null | The password if your Liferay bundle's URL requires authentication. stripComponents | int | 1 | The number of directories to strip when expanding your bundle. token | boolean | false | Whether to use a token to download a Liferay DXP bundle. This should be set to true when downloading a DXP bundle. tokenFile | File | ${user.home}/.liferay/token | The file to hold the Liferay bundle authentication token. url | URL | ${liferay.workspace.bundle.url} | The URL of the Liferay bundle to expand. userName | String | null | The user name if your Liferay bundle's URL requires authentication.

init Goal's Available Parameters

You can change the default parameter values of the init goal by creating an <execution> section containing <configuration> tags. For example,

```
<execution>
    <id>init</id>
    <goals>
        <goal>init</goal>
    </goals>
    <configuration>
    </configuration>
</execution>
```

You can set the following parameters in the init execution's <configuration> section of the POM:

Parameter Name | Type | Default Value | Description liferayHome | String | bundles | The directory where your Liferay DXP instance resides. This can be specified from the command line as -DliferayHome=.

cacheDir | File | ${user.home}/.liferay/bundles | The directory where the downloaded bundle Zip files are stored. configs | String | configs | The directory that contains the configuration files. environment | String | ${liferay.workspace.environment} | The environment with the settings appropriate for current development (e.g., common, dev, local, prod, and uat). password | String | null | The password if your Liferay bundle's URL requires authentication. stripComponents | int | 1 | The number of directories to strip when expanding your bundle. token | boolean | false | Whether to use a token to download a Liferay DXP bundle. This should be set to true when downloading a DXP bundle. tokenFile | File | ${user.home}/.liferay/token | The file to hold the Liferay bundle authentication token. url | URL | ${liferay.workspace.bundle.url} | The URL of the Liferay bundle to expand. userName | String | null | The user name if your Liferay bundle's URL requires authentication.

132.2 CSS Builder Plugin

The CSS Builder plugin lets you compile Sass files in your project.

Usage

To use the plugin, include it in your project's root pom.xml file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.css.builder</artifactId>
            <version>3.0.0</version>
            <executions>
                <execution>
                    <id>default-build</id>
                    <phase>compile</phase>
                    <goals>
                        <goal>build</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the CSS Builder configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description css-builder:build | Compiles the Sass files in the project.

Available Parameters

You can set the following parameters in the <configuration> section of the POM:

Parameter Name | Type | Default Value | Description appendCssImportTimestamps | boolean | true | Whether to append the current timestamp to the URLs in the @import CSS at-rules. baseDir | File | "src/META-INF/resources" | The base directory that contains the SCSS files to compile. dirNames | List<String> | ["/"] |

The name of the directories, relative to baseDir, which contain the SCSS files to compile. generateSourceMap | boolean | false | Whether to generate source maps for easier debugging. importDir | File | null | The META-INF/resources directory of the Liferay Frontend Common CSS artifact. This is required in order to make Bourbon and other CSS libraries available to the compilation. outputDirName | String | ".sass-cache/" | The name of the sub-directories where the SCSS files are compiled to. For each directory that contains SCSS files, a sub-directory with this name is created. precision | int | 9 | The numeric precision of numbers in Sass. rtlExcludedPathRegexps | List<String> | | The SCSS file patterns to exclude when converting for right-to-left (RTL) support. sassCompilerClassName | String | "jni" | The type of Sass compiler to use. Supported values are "jni" and "ruby". The Ruby Sass compiler requires com.liferay.sass.compiler.ruby.jar, com.liferay.ruby.gems.jar, and jruby-complete.jar to be added to the classpath.

You can also manage the com.liferay.frontend.css.common default theme dependency provided by the CSS Builder in your pom.xml. This can be modified by adding it as a project dependency:

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.frontend.css.common</artifactId>
            <version>3.0.1</version>
            <scope>provided</scope>
        </dependency>
        ...
    </dependencies>
</project>
```

There are additional Liferay theme-related dependencies you can manage this way that are provided by the Theme Builder. See this section for more information.

132.3 DB Support Plugin

The DB Support plugin lets you run the Liferay DB Support tool to execute certain actions on a local Liferay DXP database. The following actions are available:

- Cleans the Liferay database from the Service Builder tables and rows of a module.

Usage

To use the plugin, include it in your project's pom.xml file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.portal.tools.db.support</artifactId>
            <version>1.0.6</version>
            <configuration>
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>org.hsqldb</groupId>
                    <artifactId>hsqldb</artifactId>
                    <version>2.4.0</version>
                </dependency>
            </dependencies>
        </plugin>
    ...
    </plugins>
</build>
```

Also notice the configured plugin dependency. You must configure the JDBC driver used by your Liferay DXP bundle so the DB Support plugin can properly manage your database. Replace the HSQLDB driver listed above with your custom database's JDBC driver.

Goals

The plugin adds one Maven goal to your project:

Name | Description `db-support:clean-service-builder` | Cleans the Liferay DXP database from the Service Builder tables and rows of a module.

Available Parameters

You can set the following parameters in the `<configuration>` section of the POM:

Parameter Name | Type | Default Value | Description `password` | `String` | `jdbc.default.password` | The user password for connecting to the Liferay DXP database. `propertiesFile` | `File` | `null` | The `portal-ext.properties` file which contains the JDBC settings for connecting to the Liferay DXP database. `serviceXmlFile` | `File` | `null` | The `service.xml` file of the module. `servletContextName` | `String` | `null` | The servlet context name (usually the value of the `Bundle-Symbolic-Name` manifest header) of the module. `url` | `String` | `jdbc.default.url` | The JDBC URL for connecting to the Liferay DXP database. `userName` | `String` | `jdbc.default.username` | The user name for connecting to the Liferay DXP database.

132.4 Deployment Helper Plugin

The Deployment Helper plugin lets you create a cluster deployable WAR from your OSGi artifacts.

Usage

To use the plugin, include it in your project's root `pom.xml` file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.deployment.helper</artifactId>
            <version>1.0.4</version>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the Deployment Helper configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description `deployment-helper:build` | Builds a WAR which contains one or more files that are copied once the WAR is deployed.

Available Parameters

You can set the following parameters in the <configuration> section of the POM:

Parameter Name | Type | Default Value | Description deploymentFileNames | String | null | The files or directories to include in the WAR and copy once the WAR is deployed. If a directory is added to this collection, all the JAR files contained in the directory are included in the WAR. deploymentPath | String | null | The directory to which the included files are copied. outputFileName | String | null | The WAR file to build.

132.5 Javadoc Formatter Plugin

The Javadoc Formatter plugin lets you format project Javadoc comments. The tool lets you generate:

- Default @author tags to all classes.
- Comment stubs to classes, fields, and methods.
- Missing @Override annotations.
- An XML representation of the Javadoc comments, which can be used by tools in order to index the Javadocs of the project.

Usage

To use the plugin, include it in your project's root pom.xml file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.javadoc.formatter</artifactId>
            <version>1.0.32</version>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the Javadoc Formatter configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description javadoc-formatter:format | Runs the Liferay Javadoc Formatter to format files.

Available Parameters

You can set the following parameters in the <configuration> section of the POM:

Parameter Name | Type | Default Value | Description author | String | "Brian Wing Shun Chan" | The value of the @author tag to add at class level if missing. generateXml | boolean | false | Whether to generate a XML representation of the Javadoc comments. The XML files are generated in the src/main/resources directory only if the Java files are contained in src/main/java. initializeMissingJavadocs | boolean | false | Whether to add comment stubs at the class, field, and method levels. If false, only the class-level @author is added. inputDirName | String | "./" | The root directory to begin searching for Java files to format. limits | String[] | [] | The Java file name patterns, relative to the working directory, to include when formatting

Javadoc comments. The patterns must be specified without the `.java` file type suffix. If empty, all Java files are formatted. `outputFilePrefix` | `String` | `"javadocs"` | The file name prefix of the XML representation of the Javadoc comments. If `generateXML` is `false`, this property is not used. `updateJavadocs` | `boolean` | `false` | Whether to fix existing comment blocks by adding missing tags.

132.6 Lang Builder Plugin

The Lang Builder plugin lets you sort and translate the language keys in your project.

Usage

To use the plugin, include it in your project's root `pom.xml` file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.lang.builder</artifactId>
            <version>1.0.31</version>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the Lang Builder configuration here.

Goals

The plugin adds one Maven goal to your project:
Name | Description `lang-builder:build` | Runs Liferay Lang Builder to translate language property files.

Available Parameters

You can set the following parameters in the `<configuration>` section of the POM:
Parameter Name | Type | Default Value | Description `excludedLanguageIds` | `String[]` | `{"da", "de", "fi", "ja", "nl", "pt_PT", "sv"}` | The language IDs to exclude in the automatic translation. `langDirName` | `String` | `"src/content"` | The directory where the language properties files are saved. `langFileName` | `String` | `"Language"` | The file name prefix of the language properties files (e.g., `Language_it.properties`). `plugin` | `boolean` | `true` | Whether to check for duplicate language keys between the project and the portal. `portalLanguagePropertiesFileName` | `String` | `null` | The `Language.properties` file of the portal. `translate` | `boolean` | `true` | Whether to translate the language keys and generate a language properties file for each locale that's supported by Liferay DXP. `translateSubscriptionKey` | `String` | `null` | The subscription key for Microsoft Translation integration. Subscription to the Translator Text Translation API on Microsoft Cognitive Services is required. Basic subscriptions, up to 2 million characters a month, are free.

132.7 Service Builder Plugin

The Service Builder plugin lets you generate a service layer defined in a Service Builder `service.xml` file. Visit the Using Service Builder in a Maven Project tutorial to learn more about applying Service Builder to your Maven project.

Usage

To use the plugin, include it in your project's root `pom.xml` file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.portal.tools.service.builder</artifactId>
            <version>1.0.292</version>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the Service Builder configuration here.

Goals

The plugin adds one Maven goal to your project:
Name | Description `service-builder:build` | Runs the Liferay Service Builder.

Available Parameters

You can set the following parameters in the `<configuration>` section of the POM:
Parameter Name | Type | Default Value | Description `apiDirName` | `String` | `"../portal-kernel/src"` | A directory where the service API Java source files are generated. `autoImportDefaultReferences` | `boolean` | `true` | Whether to automatically add default references, like `com.liferay.portal.ClassName`, `com.liferay.portal.Resource` and `com.liferay.portal.User`, to the services. `autoNamespaceTables` | `boolean` | `null` | Whether to prefix table names by the namespace specified in the `service.xml` file. `beanLocatorUtil` | `String` | `"com.liferay.portal.kernel.bean.PortalBeanLocatorUtil"` | The fully qualified class name of a bean locator class to use in the generated service classes. `buildNumber` | `long` | `1` | A specific value to assign the `build.number` property in the `service.properties` file. `buildNumberIncrement` | `boolean` | `true` | Whether to automatically increment the `build.number` property in the `service.properties` file by one at every service generation. `databaseNameMaxLength` | `int` | `30` | The upper bound for database table and column name lengths to ensure it works on all databases. `hbmFileName` | `String` | `"src/META-INF/portal-hbm.xml"` | A Hibernate Mapping file to generate. `implDirName` | `String` | `"src"` | A directory where the service Java source files are generated. `inputFileName` | `String` | `"service.xml"` | The project's `service.xml` file. `modelHintsConfigs` | `String` | `"classpath*:META-INF/portal-model-hints.xml, META-INF/portal-model-hints.xml, classpath*:META-INF/ext-model-hints.xml, classpath*:META-INF/portlet-model-hints.xml"` | Paths to the model hints files for Liferay Service Builder to use in generating the service layer. `modelHintsFileName` | `String` | `"src/META-INF/portal-model-hints.xml"` | A model hints file for the project. `osgiModule` | `boolean` | `null` | Whether to generate the service layer for OSGi modules. `pluginName` | `String` | `null` | If specified, a plugin can enable additional generation features, such as `Clp` class generation, for non-OSGi modules. `propsUtil` | `String` | `"com.liferay.portal.util.PropsUtil"` | The fully qualified class name of the service properties util class to generate. `readOnlyPrefixes` | `String` | `"fetch, get, has, is, load, reindex, search"` | Prefixes of methods to consider read-only. `resourceActionsConfigs` | `String` | `"META-INF/resource-actions/default.xml, resource-actions/default.xml"` | Paths to the resource actions files for Liferay Service Builder to use in generating the service layer. `resourcesDirName` | `String` | `"src"` | A directory where the service non-Java files are generated. `springFileName` | `String` | `"src/META-INF/portal-spring.xml"` | A service Spring file to

generate. springNamespaces | String | "beans" | Namespaces of Spring XML Schemas to add to the service Spring file. sqlDirName | String | "../sql" | A directory where the SQL files are generated. sqlFileName | String | "portal-tables.sql" | A name (relative to sqlDir) for the file in which the SQL table creation instructions are generated. sqlIndexesFileName | String | "indexes.sql" | A name (relative to sqlDir) for the file in which the SQL index creation instructions are generated. sqlSequencesFileName | String | "sequences.sql" | A name (relative to sqlDir) for the file in which the SQL sequence creation instructions are generated. targetEntityName | String | null | If specified, it's the name of the entity for which Liferay Service Builder should generate the service. testDirName | String | "test/integration" | If specified, it's a directory where integration test Java source files are generated.

132.8 Source Formatter Plugin

The Source Formatter plugin formats project files according to Liferay's source formatting standards. For more documentation on Source Formatter specific functionality, visit the tool's documentation folder.

Usage

To use the plugin, include it in your project's root pom.xml file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.source.formatter</artifactId>
            <version>1.0.885</version>
            <executions>
                <execution>
                    <phase>process-sources</phase>
                    <goals>
                        <goal>format</goal>
                    </goals>
                </execution>
            </executions>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the Source Formatter configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description source-formatter:format | Runs the Liferay Source Formatter to format source formatting errors.

Available Parameters

You can set the following parameters in the <configuration> section of the POM:

Parameter Name | Type | Default Value | Description autoFix | boolean | true | Whether to automatically fix source formatting errors. baseDir | String | "./" | The Source Formatter base directory. *(Read-only)* fileNames | String[] | null | The file names to format, relative to the project directory. If null, all files contained in `baseDir` will be formatted. `formatCurrentBranch` | `boolean` | `false` | Whether to format only the files contained in `baseDir` that are added or modified in the current Git branch. `formatLatestAuthor` | `boolean` | `false` | Whether to format only the files contained in `baseDir` that are added or modified in the latest Git commits of the same author. `formatLocalChanges` | `boolean` | `false` | Whether to format only the unstaged files contained in `baseDir`. `gitWorkingBranchName` | `String` | `"master"` | The Git working branch name. `includeSubrepositories` | `boolean` | `false` | Whether to format files that are in read-only subrepositories. `maxLineLength` | `int` | `80` | The maximum number of characters allowed in Java files. `printErrors` | `boolean` | `true` | Whether to print formatting errors on the Standard Output stream. `processorThreadCount` | `int` | `5` | The number of threads used by Source Formatter. `showDocumentation` | `boolean` | `false` | Whether to show the documentation for the source formatting issues, if present. `throwException` | `boolean` | `false` | Whether to fail the build if formatting errors are found.

132.9 Theme Builder Plugin

The Theme Builder plugin lets you build Liferay theme files in your project. Visit the Building Themes in a Maven Project tutorial to learn more about applying Theme Builder to your Maven project.

Usage

To use the plugin, include it in your project's root `pom.xml` file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.portal.tools.theme.builder</artifactId>
            <version>1.1.7</version>
            <executions>
                <execution>
                    <phase>generate-resources</phase>
                    <goals>
                        <goal>build</goal>
                    </goals>
                    <configuration>
                    </configuration>
                </execution>
            </executions>
        </plugin>
        ...
    </plugins>
</build>
```

You can view an example POM containing the Theme Builder configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description `theme-builder:build` | Builds the theme files.

Available Parameters

You can set the following parameters in the `<configuration>` section of the POM:

Parameter Name | Type | Default Value | Description `diffsDir` | `File` | `${maven.war.src}` | The directory that contains the files to copy over the parent theme. `name` | `String` | `${project.artifactId}` | The name of the new theme. `outputDir` | `File` | `${project.build.directory}/${project.build.finalName}` | The directory where to build the theme. `parentDir` | `File` | `null` | The directory of the parent theme. `parentName` | `String` | `null` | The name of the parent theme. `templateExtension` | `String` | `"ftl"` | The extension of the template files, usually `"ftl"` or `"vm"`. `unstyledDir` | `File` | `null` | The directory of Liferay Frontend Theme Unstyled.

You can also manage the `com.liferay.frontend.theme.styled` and `com.liferay.frontend.theme.unstyled` default theme dependencies provided by the Theme Builder in your `pom.xml`. They can be modified by adding them as project dependencies:

```
<project>
    ...
    <dependencies>
        ...
        <dependency>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.frontend.theme.styled</artifactId>
            <version>3.0.4</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.frontend.theme.unstyled</artifactId>
            <version>3.0.4</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

There is an additional Liferay theme-related dependency you can manage this way that's provided by the CSS Builder. See this section for more information.

132.10 TLD Formatter Plugin

The TLD Formatter plugin lets you format a project's TLD files.

Usage

To use the plugin, include it in your project's root `pom.xml` file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.tld.formatter</artifactId>
            <version>1.0.5</version>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the TLD Formatter configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description `tld-formatter:format` | Runs the Liferay TLD Formatter to format files.

Available Parameters

You can set the following parameters in the <configuration> section of the POM:

Parameter Name | Type | Default Value | Description baseDirName | String | "./" | The base directory to begin searching for TLD files to format. plugin | boolean | true | Whether to format all the TLD files contained in the working directory. If false, all liferay-portlet-ext.tld files are ignored.

132.11 WSDD Builder Plugin

The WSDD Builder plugin lets you generate the Apache Axis Web Service Deployment Descriptor (WSDD) files from a Service Builder service.xml file.

Usage

To use the plugin, include it in your project's root pom.xml file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.portal.tools.wsdd.builder</artifactId>
            <version>1.0.10</version>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the WSDD Builder configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description wsdd-builder:build | Runs the Liferay WSDD Builder to generate the WSDD files.

Available Parameters

You can set the following parameters in the <configuration> section of the POM:

Parameter Name | Type | Default Value | Description classPath | String | null | The classpath that the Liferay WSDD Builder uses to generate WSDD files. inputFileName | String | "service.xml" | The file from which to generate the WSDD files. outputDirName | String | "src" | The directory where the *_deploy.wsdd and *_undeploy.wsdd files are generated. serverConfigFileName | String | "server-config.wsdd" | The file to generate. serviceNamespace | String | "Plugin" | The namespace for the WSDD Service.

132.12 XML Formatter Plugin

The XML Formatter plugin lets you format a project's XML files.

Usage

To use the plugin, include it in your project's root `pom.xml` file:

```
<build>
    <plugins>
    ...
        <plugin>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.xml.formatter</artifactId>
            <version>1.0.5</version>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the XML Formatter configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description `xml-formatter:format` | Runs the Liferay XML Formatter to format the project files.

Available Parameters

You can set the following parameters in the `<configuration>` section of the POM:

Parameter Name | Type | Default Value | Description `fileName` | `String` | `null` | The XML file to format. This plugin only lets you format one XML file at a time. `stripComments` | `boolean` | `false` | Whether to remove all the comments from the XML file.

Chapter 133

Project Templates

Liferay provides project templates that you can use to generate starter projects formatted in an opinionated way. These templates can be used by most build tools (e.g., Gradle, Maven, Liferay @ide@) to generate your desired project structure.

Some popular project templates include

- API project
- Fragment project
- MVC Portlet project
- Service Builder project
- Template Context Contributor project
- Theme project (WAR)
- etc.

If you're using Blade CLI, execute the following command to display a full list of project templates:

```
blade create -l
```

If you're using Maven, you can view and use the project templates as Maven archetypes. Execute the following command to list them:

```
mvn archetype:generate -Dfilter=liferay
```

Archetypes with the `com.liferay.project.templates` prefix are the latest templates offered by Liferay.

If you're using Liferay @ide@, navigate to *File* → *New* → *Liferay Module Project* and view the project templates from the *Project Template Name* drop-down menu.

In this section of reference articles, each project template is outlined with the appropriate generation command and folder structure. Visit the project template article you're most interested in to start building your own project!

133.1 Activator Template

In this article, you'll learn how to create a Liferay activator as a Liferay module. To create a Liferay activator via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t activator -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.activator \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `activator`. Suppose you want to create an activator project called `my-activator-project` with a package name of `com.liferay.docs.activator` and a class name of `Activator`. You could run the following command to accomplish this:

```
blade create -t activator -v 7.0 -p com.liferay.docs.activator -c Activator my-activator-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.activator \
    -DgroupId=com.liferay \
    -DartifactId=my-activator-project \
    -Dpackage=com.liferay.docs.activator \
    -Dversion=1.0 \
    -DclassName=Activator \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

Note that in your class, you're implementing the `org.osgi.framework.BundleActivator` interface.

After running the command above, your project's directory structure looks like this:

- `my-activator-project`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/activator`
      - `Activator.java`
  - `bnd.bnd`
  - `build.gradle`
  - `[gradlew|pom.xml]`

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.2 API Template

In this tutorial, you'll learn how to create a Liferay API as a Liferay module. To create a Liferay API via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t api -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.api \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `api`. The `api` template creates a simple `api` module with an empty public interface. For example, suppose you want to create an API project called `my-api-project` with a package name of `com.liferay.docs.api` and a class name of `MyApi`. You could run the following command to accomplish this:

```
blade create -t api -v 7.0 -p com.liferay.docs -c MyApi my-api-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.api \
    -DgroupId=com.liferay \
    -DartifactId=my-api-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=MyApi \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-api-project`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/api`

            - MyApi.java

            - resources
            - com/liferay/docs/api
            - packageinfo

    - bnd.bnd
    - build.gradle
    - [gradlew|pom.xml]

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.3 Control Menu Entry Template

In this article, you'll learn how to create a Liferay Control Menu entry as a Liferay module. To create a Liferay Control Menu entry via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t control-menu-entry -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.control.menu.entry \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `control-menu-entry`. Suppose you want to create a control menu entry project called `my-control-menu-entry-project` with a package name of `com.liferay.docs.entry.control.menu` and a class name of `SampleProductNavigationControlMenuEntry`. You could run the following command to accomplish this:

```
blade create -t control-menu-entry -v 7.0 -p com.liferay.docs.entry -c Sample my-control-menu-entry-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.control.menu.entry \
    -DgroupId=com.liferay \
    -DartifactId=my-control-menu-entry-project \
    -Dpackage=com.liferay.docs.entry \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure would look like this:

- my-control-menu-entry-project
  - gradle (only in Blade CLI generated projects)
    - wrapper
      - gradle-wrapper.jar
      - gradle-wrapper.properties
  - src
    - main
      - java
      - com/liferay/docs/entry/control/menu
      - SampleProductNavigationControlMenuEntry.java

      - resources
      - content
      - Language.properties
  - bnd.bnd
  - build.gradle
  - [gradlew|pom.xml]

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the control-menu-entry sample project for a more expanded sample of a Control Menu entry. Likewise, see the Customizing the Control Menu tutorial for instructions on customizing a Control Menu entry project.

133.4 Form Field Template

In this article, you'll learn how to create a Liferay form field as a Liferay module. To create a Liferay form field via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t form-field -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.form.field \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `form-field`. Suppose you want to create a form field project called `my-form-field-project` with a package name of `com.liferay.docs.form.field` and a class name prefix of `MyFormField`. You could run one of the following commands to accomplish this:

```
blade create -t form-field -v 7.0 -p com.liferay.docs -c MyFormField my-form-field-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.form.field \
    -DgroupId=com.liferay \
    -DartifactId=my-form-field-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=MyFormField \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-form-field-project`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/form/field`
      - `MyFormFieldDDMFormFieldRenderer.java`
      - `MyFormFieldDDMFormFieldType.java`

      - `resources`
      - `content`
      - `Language.properties`

      - `META-INF`
      - `resources`
      - `config.js`
      - `my-form-field-project.soy`
      - `my-form-field-project_field.js`
  - `bnd.bnd`
  - `build.gradle`
  - `[gradlew|pom.xml]`

The generated module is a working form field and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.5 Fragment Template

In this article, you'll learn how to create a Liferay fragment as a Liferay module. You can learn more about fragment modules in the Declaring a Fragment Host article and in section 3.14 of the OSGi Alliance's core specification document.

To create a Liferay fragment via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t fragment -v 7.0 [-h hostBundleName] [-H hostBundleVersion] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.fragment \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `fragment`. Suppose you want to create a fragment project called `my-fragment-project` with a host bundle symbolic name of `com.liferay.login.web` and host bundle version of `1.0.0`. You could run the following command to accomplish this:

```
blade create -t fragment -v 7.0 -h com.liferay.login.web -H 1.0.0 my-fragment-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.fragment \
    -DgroupId=com.liferay \
    -DartifactId=my-fragment-project \
    -Dversion=1.0 \
    -Dpackage= \
    -DhostBundleSymbolicName=com.liferay.login.web \
    -DhostBundleVersion=1.0.0 \
    -DliferayVersion=7.0
```

The folder structure is created, but there are no files. The only files created are the `bnd.bnd` and `build.gradle` files, which specify your host bundle and its information, and your build tool's files. After running the command above, your project's directory structure looks like this:

- `my-fragment-project`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`

    * main

        · java -resources -META-INF -resources

  - bnd.bnd
  - build.gradle
  - [gradlew|pom.xml]

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.6 FreeMarker Portlet Template

In this article, you'll learn how to create a Liferay FreeMarker portlet application as a Liferay module. To create a Liferay FreeMarker portlet application via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t freemarker-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.freemarker.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `freemarker-portlet`. Suppose you want to create a FreeMarker portlet project called `my-freemarker-portlet-project` with a package name of `com.liferay.docs.freemarkerportlet` and a class name of `MyFreemarkerPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.util.bridges.freemarker.FreeMarkerPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t freemarker-portlet -v 7.0 -p com.liferay.docs.freemarkerportlet -c MyFreemarkerPortlet my-freemarker-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.freemarker.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-freemarker-portlet-project \
    -Dpackage=com.liferay.docs.freemarkerportlet \
    -Dversion=1.0 \
    -DclassName=MyFreemarkerPortlet \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- my-freemarker-portlet-project
    - gradle (only in Blade CLI generated projects)
        - wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        - main
            - java
            - com/liferay/docs/freemarkerportlet
            - constants
            - MyFreemarkerPortletKeys.java

            - portlet
            - MyFreemarkerPortlet.java

            - resources
            - content
            - Language.properties

            - META-INF
            - resources
            - css
            - main.scss

            - templates
            - init.ftl
            - view.ftl
    - bnd.bnd
    - build.gradle
    - [gradlew|pom.xml]

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.7 Layout Template

In this article, you'll learn how to create a Liferay layout template as a WAR project. To create a Liferay layout template via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t layout-template -v 7.0 projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.layout.template \
    -DartifactId=[projectName] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `layout-template`. Suppose you want to create a layout template project called `my-layout-template-project`. You could run one of the following commands to accomplish this:

```
blade create -t layout-template -v 7.0 my-layout-template-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.layout.template \
    -DgroupId=com.liferay \
    -DartifactId=my-layout-template-project \
    -Dversion=1.0 \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-layout-template-project`
    - `gradle` (only in Blade CLI generated projects)
        - `wrapper`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`
    - `src`
        - `main`
            - `webapp`
            - `WEB-INF`
            - `liferay-layout-templates.xml`
            - `liferay-plugin-package.properties`

            - `my-layout-template-project.png`
            - `my-layout-template-project.tpl`
    - `build.gradle`
    - `[gradlew|pom.xml]`

The generated WAR is a working layout template and is deployable to a Liferay DXP instance. To build upon the generated layout template, modify the project by adding logic and additional files to the folders outlined above.

133.8 MVC Portlet Template

In this article, you'll learn how to create a Liferay MVC portlet application as a Liferay module. To create a Liferay MVC portlet application via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t mvc-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.mvc.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `mvc-portlet`. Suppose you want to create an MVC portlet project called `my-mvc-portlet-project` with a package name of `com.liferay.docs.mvcportlet` and a class name of `MyMvcPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t mvc-portlet -v 7.0 -p com.liferay.docs.mvcportlet -c MyMvcPortlet my-mvc-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.mvc.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-mvc-portlet-project \
    -Dpackage=com.liferay.docs.mvcportlet \
    -Dversion=1.0 \
    -DclassName=MyMvcPortlet \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-mvc-portlet-project`
    - `gradle` (only in Blade CLI generated projects)
        - `wrapper`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`
    - `src`
        - `main`

    - java
    - com/liferay/docs/mvcportlet
    - MyMvcPortlet.java

    - resources
    - content
    - Language.properties

    - META-INF
    - resources
    - init.jsp
    - view.jsp

- bnd.bnd
- build.gradle
- [gradlew|pom.xml]

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.9 npm Angular Portlet Template

In this article, you'll learn how to create an npm Angular portlet as a Liferay module. To create an npm Angular portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t npm-angular-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.angular.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

The template for this kind of project is `npm-angular-portlet`. Suppose you want to create an npm Angular portlet project called `my-npm-angular-portlet` with a package name of `com.liferay.npm.angular` and a class name of `MyNpmAngularPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t npm-angular-portlet -v 7.0 -p com.liferay.npm.angular -c MyNpmAngularPortlet my-npm-angular-portlet
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.angular.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-npm-angular-portlet \
    -Dpackage=com.liferay.npm.angular \
    -Dversion=1.0 \
    -DclassName=MyNpmAngularPortlet \
    -DpackageJsonVersion=1.0.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-npm-angular-portlet`
  - `[gradle|.mvn]`
    - `wrapper`
      - `[gradle|maven]-wrapper.jar`
      - `[gradle|maven]-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/npm/angular`
      - `constants`
      - `MyNpmAngularPortletKeys.java`

      - `portlet`
      - `MyNpmAngularPortlet.java`

      - `resources`
      - `content`
      - `Language.properties`

      - `META-INF`
      - `resources`
      - `js`
      - `app`
      - `app.component.ts`
      - `app.module.ts`

      - `angular-loader.ts`
      - `main.ts`

      - `init.jsp`

        · view.jsp

    – .babelrc
    – .npmbundlerrc
    – bnd.bnd
    – [build.gradle|pom.xml]
    – [gradlew|mvnw]
    – package.json
    – tsconfig.json

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated portlet, modify the project by adding logic and additional files to the folders outlined above.

133.10 npm Billboard.js Portlet Template

In this article, you'll learn how to create an npm Billboard.js portlet as a Liferay module. To create an npm Billboard.js portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t npm-billboardjs-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.billboardjs.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

The template for this kind of project is `npm-billboardjs-portlet`. Suppose you want to create an npm Billboard.js portlet project called `my-npm-billboardjs-portlet` with a package name of `com.liferay.npm.billboardjs` and a class name of `MyNpmBillboardjsPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t npm-billboardjs-portlet -v 7.0 -p com.liferay.npm.billboardjs -c MyNpmBillboardjsPortlet my-npm-billboardjs-portlet
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.billboardjs.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-npm-billboardjs-portlet \
    -Dpackage=com.liferay.npm.billboardjs \
    -Dversion=1.0 \
    -DclassName=MyNpmBillboardjsPortlet \
    -DpackageJsonVersion=1.0.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-npm-billboardjs-portlet`
  - `[gradle|.mvn]`
    - `wrapper`
      - `[gradle|maven]-wrapper.jar`
      - `[gradle|maven]-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/npm/billboardjs`
      - `constants`
      - `MyNpmBillboardjsPortletKeys.java`

      - `portlet`
      - `MyNpmBillboardjsPortlet.java`

      - `resources`
      - `content`
      - `Language.properties`

      - `META-INF`
      - `resources`
      - `js`
      - `data.json`
      - `index.es.js`

      - `init.jsp`
      - `view.jsp`
  - `.babelrc`
  - `.npmbundlerrc`
  - `bnd.bnd`
  - `[build.gradle|pom.xml]`
  - `[gradlew|mvnw]`
  - `package.json`

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated portlet, modify the project by adding logic and additional files to the folders outlined above.

133.11 npm Isomorphic Portlet Template

In this article, you'll learn how to create an npm Isomorphic portlet as a Liferay module. To create an npm Isomorphic portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t npm-isomorphic-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.isomorphic.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

The template for this kind of project is `npm-isomorphic-portlet`. Suppose you want to create an npm Isomorphic portlet project called `my-npm-isomorphic-portlet` with a package name of `com.liferay.npm.isomorphic` and a class name of `MyNpmIsomorphicPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t npm-isomorphic-portlet -v 7.0 -p com.liferay.npm.isomorphic -c MyNpmIsomorphicPortlet my-npm-isomorphic-portlet
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.isomorphic.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-npm-isomorphic-portlet \
    -Dpackage=com.liferay.npm.isomorphic \
    -Dversion=1.0 \
    -DclassName=MyNpmIsomorphicPortlet \
    -DpackageJsonVersion=1.0.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-npm-isomorphic-portlet`

- [gradle|.mvn]

  * wrapper

    · [gradle|maven]-wrapper.jar
    · [gradle|maven]-wrapper.properties

- src

  * main

    · java
    · com/liferay/npm/isomorphic
    · constants
    · MyNpmIsomorphicPortletKeys.java

    · portlet
    · MyNpmIsomorphicPortlet.java

    · resources
    · content
    · Language.properties

    · META-INF
    · resources
    · js
    · index.es.js

    · init.jsp
    · view.jsp

- .babelrc
- .npmbundlerrc
- bnd.bnd
- [build.gradle|pom.xml]
- [gradlew|mvnw]
- package.json

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated portlet, modify the project by adding logic and additional files to the folders outlined above.

133.12 npm jQuery Portlet Template

In this article, you'll learn how to create an npm jQuery portlet as a Liferay module. To create an npm jQuery portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t npm-jquery-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.jquery.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

The template for this kind of project is `npm-jquery-portlet`. Suppose you want to create an npm jQuery portlet project called `my-npm-jquery-portlet` with a package name of `com.liferay.npm.jquery` and a class name of `MyNpmjQueryPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t npm-jquery-portlet -v 7.0 -p com.liferay.npm.jquery -c MyNpmjQueryPortlet my-npm-jquery-portlet
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.jquery.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-npm-jquery-portlet \
    -Dpackage=com.liferay.npm.jquery \
    -Dversion=1.0 \
    -DclassName=MyNpmjQueryPortlet \
    -DpackageJsonVersion=1.0.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-npm-jquery-portlet`
  - `[gradle|.mvn]`
    - `wrapper`
      - `[gradle|maven]-wrapper.jar`
      - `[gradle|maven]-wrapper.properties`
  - `src`
    - `main`
      - `java`

            · com/liferay/npm/jquery
            · constants
            · MyNpmjQueryPortletKeys.java

            · portlet
            · MyNpmjQueryPortlet.java

            · resources
            · content
            · Language.properties

            · META-INF
            · resources
            · js
            · index.es.js

            · init.jsp
            · view.jsp

- .babelrc
- .npmbundlerrc
- bnd.bnd
- [build.gradle|pom.xml]
- [gradlew|mvnw]
- package.json

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated portlet, modify the project by adding logic and additional files to the folders outlined above.

133.13 npm Metal.js Portlet Template

In this article, you'll learn how to create an npm Metal.js portlet as a Liferay module. To create an npm Metal.js portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t npm-metaljs-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.metaljs.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

The template for this kind of project is `npm-metaljs-portlet`. Suppose you want to create an npm Metal.js portlet project called `my-npm-metaljs-portlet` with a package name of `com.liferay.npm.metaljs` and a class name of `MyNpmMetaljsPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t npm-metaljs-portlet -v 7.0 -p com.liferay.npm.metaljs -c MyNpmMetaljsPortlet my-npm-metaljs-portlet
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.metaljs.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-npm-metaljs-portlet \
    -Dpackage=com.liferay.npm.metaljs \
    -Dversion=1.0 \
    -DclassName=MyNpmMetaljsPortlet \
    -DpackageJsonVersion=1.0.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-npm-metaljs-portlet`
  - `[gradle|.mvn]`
    - `wrapper`
      - `[gradle|maven]-wrapper.jar`
      - `[gradle|maven]-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/npm/metaljs`
      - `constants`
      - `MyNpmMetaljsPortletKeys.java`

      - `portlet`
      - `MyNpmMetaljsPortlet.java`

      - `resources`
      - `content`
      - `Language.properties`

            - META-INF
            - resources
            - js
            - HelloWorld.soy
            - index.es.js
            - init.jsp
            - view.jsp

    - .babelrc
    - .npmbundlerrc
    - bnd.bnd
    - [build.gradle|pom.xml]
    - [gradlew|mvnw]
    - package.json

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated portlet, modify the project by adding logic and additional files to the folders outlined above.

133.14 npm Portlet Template

In this article, you'll learn how to create an npm portlet as a Liferay module. To create an npmportlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t npm-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

The template for this kind of project is `npm-portlet`. Suppose you want to create an npm portlet project called `my-npm-portlet` with a package name of `com.liferay.npm` and a class name of `MyNpmPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t npm-portlet -v 7.0 -p com.liferay.npm -c MyNpmPortlet my-npm-portlet
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-npm-portlet \
    -Dpackage=com.liferay.npm \
    -Dversion=1.0 \
    -DclassName=MyNpmPortlet \
    -DpackageJsonVersion=1.0.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-npm-portlet`
  - `[gradle|.mvn]`
    - `wrapper`
      - `[gradle|maven]-wrapper.jar`
      - `[gradle|maven]-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/npm`
      - `constants`
      - `MyNpmPortletKeys.java`

      - `portlet`
      - `MyNpmPortlet.java`

      - `resources`
      - `content`
      - `Language.properties`

      - `META-INF`
      - `resources`
      - `js`
      - `index.es.js`

      - `init.jsp`
      - `view.jsp`
  - `.babelrc`
  - `.npmbundlerrc`
  - `bnd.bnd`
  - `[build.gradle|pom.xml]`
  - `[gradlew|mvnw]`

- package.json

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated portlet, modify the project by adding logic and additional files to the folders outlined above.

133.15 npm React Portlet Template

In this article, you'll learn how to create an npm React portlet as a Liferay module. To create an npm React portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t npm-react-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.react.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

The template for this kind of project is `npm-react-portlet`. Suppose you want to create an npm React portlet project called `my-npm-react-portlet` with a package name of `com.liferay.npm.react` and a class name of `MyNpmReactPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t npm-react-portlet -v 7.0 -p com.liferay.npm.react -c MyNpmReactPortlet my-npm-react-portlet
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.react.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-npm-react-portlet \
    -Dpackage=com.liferay.npm.react \
    -Dversion=1.0 \
    -DclassName=MyNpmReactPortlet \
    -DpackageJsonVersion=1.0.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- my-npm-react-portlet
  - [gradle|.mvn]
    - wrapper
      - [gradle|maven]-wrapper.jar
      - [gradle|maven]-wrapper.properties
  - src
    - main
      - java
      - com/liferay/npm/react
      - constants
      - MyNpmReactPortletKeys.java

      - portlet
      - MyNpmReactPortlet.java

      - resources
      - content
      - Language.properties

      - META-INF
      - resources
      - js
      - index.es.js

      - init.jsp
      - view.jsp
  - .babelrc
  - .npmbundlerrc
  - bnd.bnd
  - [build.gradle|pom.xml]
  - [gradlew|mvnw]
  - package.json

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated portlet, modify the project by adding logic and additional files to the folders outlined above.

133.16 npm Vue.js Portlet Template

In this article, you'll learn how to create an npm Vue.js portlet as a Liferay module. To create an npm Vue.js portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t npm-vuejs-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.vuejs.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

The template for this kind of project is `npm-vuejs-portlet`. Suppose you want to create an npm Vue.js portlet project called `my-npm-vuejs-portlet` with a package name of `com.liferay.npm.vuejs` and a class name of `MyNpmVuejsPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t npm-vuejs-portlet -v 7.0 -p com.liferay.npm.vuejs -c MyNpmVuejsPortlet my-npm-vuejs-portlet
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.npm.vuejs.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-npm-vuejs-portlet \
    -Dpackage=com.liferay.npm.vuejs \
    -Dversion=1.0 \
    -DclassName=MyNpmVuejsPortlet \
    -DpackageJsonVersion=1.0.0 \
    -DliferayVersion=7.0
```

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.
After running the command above, your project's directory structure looks like this:

- `my-npm-vuejs-portlet`
    - `.mvn` (only in Maven Blade CLI generated projects)
        - `wrapper`

        · maven-wrapper.jar
        · maven-wrapper.properties

  – src

      * main

        · java
        · com/liferay/npm/vuejs
        · constants
        · MyNpmVuejsPortletKeys.java

        · portlet
        · MyNpmVuejsPortlet.java

        · resources
        · content
        · Language.properties

        · META-INF
        · resources
        · lib
        · index.es.js

        · init.jsp
        · view.jsp

  – .babelrc
  – .npmbundlerrc
  – bnd.bnd
  – [build.gradle|pom.xml]
  – mvnw (only in Maven Blade CLI generated projects)
  – mvnw.cmd (only in Maven Blade CLI generated projects)
  – package.json

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated portlet, modify the project by adding logic and additional files to the folders outlined above.

133.17 Panel App Template

In this article, you'll learn how to create a Liferay panel app and category as a Liferay module. To create a Liferay panel app and category via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t panel-app -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.panel.app \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `panel-app`. Suppose you want to create a panel app project called `my-panel-app-project` with a package name prefix of `com.liferay.docs` and a class name prefix of `Sample`. You could run the following command to accomplish this:

```
blade create -t panel-app -v 7.0 -p com.liferay.docs -c Sample my-panel-app-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.panel.app \
    -DgroupId=com.liferay \
    -DartifactId=my-panel-app-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure would look like this

- `my-panel-app-project`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/`
      - `application/list`
      - `SamplePanelApp.java`
      - `SamplePanelCategory.java`

      - `constants`
      - `SamplePanelCategoryKeys.java`
      - `SamplePortletKeys.java`

      - `portlet`

    - SamplePortlet.java
    - resources
    - content
    - Language.properties
    - META-INF
    - resources
    - init.jsp
    - view.jsp
- bnd.bnd
- build.gradle
- [gradlew|pom.xml]

The generated module is functional and is deployable to a Liferay DXP instance. The generated module, by default, creates a panel category with a panel app in Liferay DXP's Product Menu. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the Customizing the Product Menu tutorial for instructions on customizing a panel app project.

133.18 Portlet Configuration Icon

In this article, you'll learn how to create a Liferay portlet configuration icon as a Liferay module. To create a portlet configuration icon via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t portlet-configuration-icon -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.configuration.icon \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `portlet-configuration-icon`. Suppose you want to create a portlet configuration icon project called `my-portlet-config-icon` with a package name of `com.liferay.docs.portlet.configuration.icon` and a class name of `SamplePortletConfigurationIcon`. You could run the following command to accomplish this:

```
blade create -t portlet-configuration-icon -v 7.0 -p com.liferay.docs -c Sample my-portlet-config-icon
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.configuration.icon \
    -DgroupId=com.liferay \
    -DartifactId=my-portlet-config-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure would look like this

- `my-portlet-config-icon`
    - `gradle` (only in Blade CLI generated projects)
        * `wrapper`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`
    - `src`
        * `main`
            - `java`
            - `com/liferay/docs/portlet/configuration/icon`
            - `SamplePortletConfigurationIcon.java`

            - `resources`
            - `content`
            - `Language.properties`
    - `bnd.bnd`
    - `build.gradle`
    - `[gradlew|pom.xml]`

The generated module is functional and is deployable to a Liferay DXP instance. The generated module, by default, creates a sample link in the Hello World portlet's Options menu. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the portlet-configuration-icon sample project for a more expanded sample of a portlet configuration icon.

133.19 Portlet Template

In this article, you'll learn how to create a Liferay portlet application as a Liferay module. To create a Liferay portlet application via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `portlet`. Suppose you want to create a portlet project called `my-portlet-project` with a package name of `com.liferay.docs.portlet` and a class name of `MyPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `javax.portlet.GenericPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t portlet -v 7.0 -p com.liferay.docs.portlet -c MyPortlet my-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-portlet-project \
    -Dpackage=com.liferay.docs.portlet \
    -Dversion=1.0 \
    -DclassName=MyPortlet \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-portlet-project`
    - `gradle` (only in Blade CLI generated projects)
        - `wrapper`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`
    - `src`
        - `main`
            - `java`
            - `com/liferay/docs/portlet`
            - `MyPortlet.java`
    - `bnd.bnd`
    - `build.gradle`
    - `[gradlew|pom.xml]`

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.20 Portlet Provider Template

In this article, you'll learn how to create a Liferay portlet provider as a Liferay module. To create a Liferay portlet provider via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t portlet-provider -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.provider \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `portlet-provider`. Suppose you want to create a portlet provider project called `my-portlet-provider-project` with a package name of `com.liferay.docs.portlet` and a class name prefix of `Sample`. You could run the following command to accomplish this:

```
blade create -t portlet-provider -v 7.0 -p com.liferay.docs -c Sample my-portlet-provider-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.provider \
    -DgroupId=com.liferay \
    -DartifactId=my-portlet-provider-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure would look like this

- `my-portlet-provider-project`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/portlet`

- SampleAddPortletProvider.java
- SamplePortlet.java

- resources
- META-INF
- resources
- init.jsp
- view.jsp

- bnd.bnd
- build.gradle
- [gradlew|pom.xml]

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the Providing Portlets to Manage Requests tutorial for instructions on customizing a portlet provider project.

133.21 Portlet Toolbar Contributor Template

In this article, you'll learn how to create a Liferay portlet toolbar contributor as a Liferay module. To create a portlet toolbar contributor entry via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t portlet-toolbar-contributor -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.toolbar.contributor \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `portlet-toolbar-contributor`. Suppose you want to create a portlet toolbar contributor project called `my-portlet-toolbar-contributor` with a package name of `com.liferay.docs.portlet.toolbar.contributor` and a class name of `SamplePortletToolbarContributor`. You could run the following command to accomplish this:

```
blade create -t portlet-toolbar-contributor -v 7.0 -p com.liferay.docs -c Sample my-portlet-toolbar-contributor
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.portlet.toolbar.contributor \
    -DgroupId=com.liferay \
    -DartifactId=my-portlet-toolbar-contributor \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure would look like this

- `my-portlet-toolbar-contributor`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/portlet/toolbar/contributor`
      - `SamplePortletToolbarContributor.java`

      - `resources`
      - `content`
      - `Language.properties`
  - `bnd.bnd`
  - `build.gradle`
  - `[gradlew|pom.xml]`

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. This generated project, by default, creates a new button on the Hello World portlet's toolbar. You can visit the portlet-toolbar-contributor sample project for a more expanded sample of a portlet toolbar contributor.

133.22 REST Template

In this article, you'll learn how to create a Liferay RESTful web service packaged in a Liferay module. To create a Liferay RESTful web service via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t rest -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.rest \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `rest`. Suppose you want to create a RESTful web service project called `my-rest-project` with a package name of `com.liferay.docs.application` and a class name prefix of `Rest`. You could run one of the following commands to accomplish this:

```
blade create -t rest -v 7.0 -p com.liferay.docs -c Rest my-rest-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.rest \
    -DgroupId=com.liferay \
    -DartifactId=my-rest-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Rest \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-rest-project`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/application`
      - `RestApplication.java`

      - `resources`
      - `configuration`
      - `com.liferay.portal.remote.cxf.common.configuration.CXFEndpointPublisherConfiguration-cxf`
      - `com.liferay.portal.remote.rest.extender.configuration.RestExtenderConfiguration-rest`
  - `bnd.bnd`
  - `build.gradle`
  - `[gradlew|pom.xml]`

The generated module is a working RESTful web service and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.23 Service Template

In this article, you'll learn how to create a Liferay service as a Liferay module. To create a Liferay service via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t service -v 7.0 [-p packageName] [-c className] [-s serviceName] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DserviceName=[serviceName] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `service`. Suppose you want to create a service project called `my-service-project` with a package name of `com.liferay.docs.service` and a class name of `Service`. Also, you'd like to create a service of type `com.liferay.portal.kernel.events.LifecycleAction` that also implements that same service. You could run the following command to accomplish this:

```
blade create -t service -v 7.0 -p com.liferay.docs.service -c Service -s com.liferay.portal.kernel.events.LifecycleAction  my-service-
project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service \
    -DgroupId=com.liferay \
    -DartifactId=my-service-project \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Service \
    -DclassName=com.liferay.portal.kernel.events.LifecycleAction \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure would look like this

- `my-service-project`
    - `gradle` (only in Blade CLI generated projects)
        - `wrapper`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`
    - `src`
        - `main`

        - java
        - com/liferay/docs/service
        - Service.java

  - bnd.bnd
  - build.gradle
  - [gradlew|pom.xml]

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.24 Service Builder Template

In this article, you'll learn how to create a Liferay portlet application that uses Service Builder as Liferay modules. To create a Liferay Service Builder project via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t service-builder -v 7.0 [-p packageName] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service.builder \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DapiPath=[apiPath] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `service-builder`. Suppose you want to create a Service Builder project called tasks with a package name of `com.liferay.docs.tasks`. You could run the following command to accomplish this:

```
blade create -t service-builder -v 7.0 -p com.liferay.docs.tasks tasks
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service.builder \
    -DgroupId=com.liferay \
    -DartifactId=tasks \
    -Dpackage=com.liferay.docs.tasks \
    -Dversion=1.0 \
    -DapiPath=com.liferay.api.path \
    -DliferayVersion=7.0
```

This task creates the `tasks-api` and `tasks-service` folders. In many cases, a Service Builder project also requires a `-web` folder to hold, for example, portlet classes. This should be created manually. After running the command above, your project's directory structure looks like this:

- tasks

- `gradle` (only in Blade CLI generated projects)
    - `wrapper`
        - `gradle-wrapper.jar`
        - `gradle-wrapper.properties`
- `tasks-api`
    - `bnd.bnd`
    - `build.gradle`
- `tasks-service`
    - `bnd.bnd`
    - `build.gradle`
    - `service.xml`
- `build.gradle`
- `[gradlew|pom.xml]`
- `settings.gradle`

To generate your service and API classes for the `*-api` and `*-service` modules, replace the `service.xml` file in the `*-service` module. Depending on your build tool, you can build your services by executing

```
blade gw buildService
```

or

```
mvn service-builder:build
```

from the tasks root directory. Note that `blade gw` only works if the Gradle wrapper can be detected. To ensure the availability of the Gradle wrapper, be sure to work in a Liferay Workspace.

The `mvn service-builder:build` command only works if you're using the `com.liferay.portal.tools.service.builder` plugin version 1.0.145+. Maven projects using an earlier version of the Service Builder plugin should update their POM accordingly.

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

Related Topics

Running Service Builder and Understanding the Generated Code

Using Service Builder in a Maven Project

Service Builder with Maven

133.25 Service Wrapper Template

In this article, you'll learn how to create a Liferay service wrapper as a Liferay module. To create a Liferay service wrapper via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t service-wrapper -v 7.0 [-p packageName] [-c className] [-s serviceWrapperClass] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service.wrapper \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DserviceWrapperClass=[serviceWrapperClass] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `service-wrapper`. Suppose you want to create a service wrapper project called `service-override` with a package name of `com.liferay.docs.serviceoverride` and a class name of `UserLocalServiceOverride`. Also, you'd like to create a service of type `com.liferay.portal.kernel.service.ServiceWrapper` that extends the `com.liferay.portal.service.UserLocalServiceWrapper` class. You could run the following command to accomplish this:

```
blade create -t service-wrapper -v 7.0 -p com.liferay.docs.serviceoverride -c UserLocalServiceOverride -s com.liferay.portal.kernel.service.UserLocalService
override
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.service.wrapper \
    -DgroupId=com.liferay \
    -DartifactId=service-override \
    -Dpackage=com.liferay.docs.serviceoverride \
    -Dversion=1.0 \
    -DclassName=UserLocalServiceOverride \
    -DserviceWrapperClass=com.liferay.portal.kernel.service.UserLocalServiceWrapper \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*.

After running the command above, your project's directory structure looks like this:

- `service-override`
    - `gradle` (only in Blade CLI generated projects)
        - `wrapper`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`

  - src

    * main

      · java
      · com/liferay/docs/serviceoverride
      · UserLocalServiceOverride.java

  - bnd.bnd
  - build.gradle
  - [gradlew|pom.xml]

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.26 Simulation Panel Entry Template

In this article, you'll learn how to create a Liferay simulation panel entry as a Liferay module. To create a simulation panel entry via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t simulation-panel-entry -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.simulation.panel.entry \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `simulation-panel-entry`. Suppose you want to create a simulation panel entry project called `my-simulation-panel-entry` with a package name of `com.liferay.docs.application.list` and a class name of `SampleSimulationPanelApp`. You could run the following command to accomplish this:

```
blade create -t simulation-panel-entry -v 7.0 -p com.liferay.docs -c Sample my-simulation-panel-entry
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.simulation.panel.entry \
    -DgroupId=com.liferay \
    -DartifactId=my-simulation-panel-entry \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure would look like this

- `my-simulation-panel-entry`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/application/list`
      - `SampleSimulationPanelApp.java`

      - `resources`
      - `content`
      - `Language.properties`

      - `META-INF`
      - `resources`
      - `simulation_panel.jsp`
  - `bnd.bnd`
  - `build.gradle`
  - `[gradlew|pom.xml]`

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the simulation-panel-app sample project for a more expanded sample of a control menu entry. Likewise, see the Extending the Simulation Menu tutorial for instructions on customizing a simulation panel entry project.

133.27 Soy Portlet Template

In this article, you'll learn how to create a Soy portlet application as a Liferay module. To create a Soy portlet as a module via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t soy-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.soy.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `soy-portlet`. Suppose you want to create an Soy portlet project called `my-soy-portlet-project` with a package name of `com.liferay.docs.soyportlet` and a class name of `MySoyPortlet`. Also, you'd like to create a service of type `javax.portlet.Portlet` that extends the `com.liferay.portal.portlet.bridge.soy.SoyPortlet` class. Here, *service* means an OSGi service, not a Liferay API. Another way to say *service type* is to say *component type*. You could run the following command to accomplish this:

```
blade create -t soy-portlet -v 7.0 -p com.liferay.docs.soyportlet -c MySoyPortlet my-soy-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.soy.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-soy-portlet-project \
    -Dpackage=com.liferay.docs.soyportlet \
    -Dversion=1.0 \
    -DclassName=MySoyPortlet \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-soy-portlet-project`
    - `gradle` (only in Blade CLI generated projects)
        - `wrapper`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`
    - `src`
        - `main`
            - `java`
            - `com/liferay/docs/soyportlet`
            - `constants`
            - `MySoyPortletKeys.java`

            - `portlet`
            - `action`
            - `MySoyPortletNavigationMVCRenderCommand.java`

            - `MySoyPortlet.java`

            - `resources`
            - `content`
            - `Language.properties`

        - META-INF
        - resources
        - Footer.es
        - Footer.soy
        - Header.es
        - Header.soy
        - Navigation.es
        - Navigation.soy
        - View.es
        - View.soy

    - bnd.bnd
    - build.gradle
    - package.json
    - [gradlew|pom.xml]

The generated module is a working application and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above.

133.28 Spring MVC Portlet Template

In this article, you'll learn how to create a Liferay Spring MVC portlet application as a WAR. To create a Liferay Spring MVC portlet via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t spring-mvc-portlet -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.spring.mvc.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `spring-mvc-portlet`. Suppose you want to create a Spring MVC portlet project called `my-spring-mvc-portlet-project` with a package name of `com.liferay.docs.springmvcportlet` and a class name of `MySpringMvcPortlet`. Also, you'd like to create a Spring-annotated portlet class named `MySpringMvcPortletViewController`.

```
blade create -t spring-mvc-portlet -v 7.0 -p com.liferay.docs.springmvcportlet -c MySpringMvcPortlet my-spring-mvc-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.spring.mvc.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-spring-mvc-portlet-project \
    -Dpackage=com.liferay.docs.springmvcportlet \
    -Dversion=1.0 \
    -DclassName=MySpringMvcPortlet \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure looks like this:

- `my-spring-mvc-portlet-project`
    - `gradle` (only in Blade CLI generated projects)
        - `wrapper`
            - `gradle-wrapper.jar`
            - `gradle-wrapper.properties`
    - `src`
        - `main`
            - `java`
            - `com/liferay/docs/springmvcportlet/portlet`
            - `MySpringMvcPortletViewController`

            - `resources`
            - `content`
            - `Language.properties`

            - `webapp`
            - `css`
            - `main.scss`

            - `WEB-INF`
            - `jsp`
            - `init.jsp`
            - `view.jsp`

            - `spring-context`
            - `portlet`
            - `my-spring-mvc-portlet-project.xml`

            - `portlet-application-context.xml`

            - `tld`
            - `liferay-portlet.tld`
            - `liferay-portlet-ext.tld`
            - `liferay-security.tld`
            - `liferay-theme.tld`
            - `liferay-ui.tld`
            - `liferay-util.tld`

            · liferay-display.xml
            · liferay-plugin-package.properties
            · liferay-portlet.xml
            · portlet.xml
            · web.xml

            · icon.png

    – build.gradle
    – [gradlew|pom.xml]

The generated WAR is a working application and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the springmvc-portlet sample project for a more expanded sample of a Spring MVC portlet.

133.29 Template Context Contributor Template

In this article, you'll learn how to create a Liferay template context contributor as a Liferay module. To create a template context contributor via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t template-context-contributor -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.template.context.contributor \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `template-context-contributor`. Suppose you want to create a template context contributor project called `my-template-context-contributor` with a package name of `com.liferay.docs.theme.contributor` and a class name of `SampleTemplateContextContributor`. You could run the following command to accomplish this:

```
blade create -t template-context-contributor -v 7.0 -p com.liferay.docs -c Sample my-template-context-contributor
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.template.context.contributor \
    -DgroupId=com.liferay \
    -DartifactId=my-template-context-contributor \
    -Dpackage=com.liferay.docs \
    -Dversion=1.0 \
    -DclassName=Sample \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.0
```

After running the command above, your project's directory structure would look like this

- `my-template-context-contributor`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/theme/contributor`
      - `SampleTemplateContextContributor.java`
  - `bnd.bnd`
  - `build.gradle`
  - `[gradlew|pom.xml]`

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the template-context-contributor sample project for a more expanded sample of a template context contributor. Likewise, see the Context Contributors tutorial for instructions on customizing a template context contributor project.

133.30 Theme Template

In this article, you'll learn how to create a Liferay theme as a WAR project. To create a Liferay theme via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t theme -v 7.0 projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.theme \
    -DartifactId=[projectName] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `theme`. Suppose you want to create a theme project called `my-theme-project` as a WAR file. You could run the following command to accomplish this:

```
blade create -t theme -v 7.0 my-theme-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.theme \
    -DgroupId=com.liferay \
    -DartifactId=my-theme-project \
    -Dversion=1.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's folder structure looks like this:

- `my-theme-project`
  - `gradle` (only in Blade CLI generated projects)
    - `wrapper`
      - `gradle-wrapper.jar`
      - `gradle-wrapper.properties`
  - `src`
    - `main`
      - `resources`
      - `resources-importer`
      - `sitemap.json`

      - `webapp`
      - `css`
      - `_custom.scss`

      - `WEB-INF`
      - `liferay-plugin-package.properties`
      - `web.xml`
  - `build.gradle`
  - `[gradlew|pom.xml]`

The generated theme is functional and is deployable to a Liferay DXP instance. To build upon the generated project, modify the project by adding logic and additional files to the folders outlined above. You can visit the simple-theme project for a more expanded sample of a theme. Likewise, see the Themes and Layout Templates tutorial section for more information on creating themes.

133.31 Theme Contributor Template

In this article, you'll learn how to create a Liferay theme contributor as a Liferay module. To create a theme contributor via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t theme-contributor -v 7.0 [--contributor-type contributorType] [-p packageName] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.theme.contributor \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DcontributorType=[contributorType] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `theme-contributor`. Suppose you want to create a theme contributor project called `my-theme-contributor` with a package name of `com.liferay.docs.theme.contributor` and a contributor type of `my-contributor`. You could run the following command to accomplish this:

```
blade create -t theme-contributor -v 7.0 --contributor-type my-contributor -p com.liferay.docs.theme.contributor my-theme-contributor
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.theme.contributor \
    -DgroupId=com.liferay \
    -DartifactId=my-theme-contributor \
    -Dpackage=com.liferay.docs.theme.contributor \
    -Dversion=1.0 \
    -DcontributorType=my-contributor \
    -DliferayVersion=7.0
```

After running the command above, your project's folder structure would look like this:

- `my-theme-contributor`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/theme/contributor`

      - `resources/META-INF/resources`
      - `css`
      - `my-contributor`
      - `_body.scss`
      - `_control_menu.scss`
      - `_product_menu.scss`
      - `_simulation_panel.scss`

      - `my-contributor.scss`

      - `js`
      - `my-contributor.js`
  - `bnd.bnd`
  - `build.gradle` (only in Gradle Blade CLI generated projects)
  - `mvnw` (only in Maven Blade CLI generated projects)

    - `mvnw.cmd` (only in Maven Blade CLI generated projects)
    - `pom.xml` (only in Maven-related projects)

The generated module is functional and is deployable to a Liferay DXP instance. To build upon the generated app, modify the project by adding logic and additional files to the folders outlined above. You can visit the Blade Theme Contributor sample project for a more expanded sample of a theme contributor. Likewise, see the Theme Contributors tutorial for instructions on customizing a theme contributor project.

133.32 WAR Hook Template

In this article, you'll learn how to create a Liferay WAR hook project. To create a Liferay WAR hook via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t war-hook -v 7.0 [-p packageName] [-c className] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.hook \
    -DartifactId=[projectName]
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `war-hook`. Suppose you want to create a WAR hook project called `my-war-hook-project` with a package name of `com.liferay.docs` and a class name of `MyWarHook`. You could run the following command to accomplish this:

```
blade create -t war-hook -v 7.0 -p com.liferay.docs -c MyWarHook my-war-hook-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.hook \
    -DgroupId=com.liferay \
    -DartifactId=my-war-hook-project \
    -Dpackage=com.liferay.docs \
    -DclassName=MyWarHook \
    -Dversion=1.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's folder structure looks like this:

- `my-war-hook-project`
    - `[gradle|.mvn]`
        - `wrapper`
            - `[gradle|maven]-wrapper.jar`
            - `[gradle|maven]-wrapper.properties`

  - src

    * main

      · java
      · com/liferay/docs
      · MyWarHookLoginPostAction
      · MyWarHookStartupAction

      · resources
      · portal.properties

      · webapp
      · WEB-INF
      · liferay-hook.xml
      · liferay-plugin-package.properties
      · web.xml

  - [build.gradle|pom.xml]
  - [gradlew|mvnw]

The generated WAR hook is functional and is deployable to a Liferay DXP instance. To build upon the generated project, modify the project by adding logic and additional files to the folders outlined above. Deploying WAR hooks is supported for 7.0, however, it is recommended to optimize your WAR hooks to fragments or other applicable module projects. You can visit the Customizing section for info on how to do this for many project types. See the Customizing Liferay Portal section for more information on WAR hooks.

133.33 WAR MVC Portlet Template

In this article, you'll learn how to create a Liferay MVC portlet project as a WAR file. To create a Liferay MVC portlet project as a WAR via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t war-mvc-portlet -v 7.0 [-p packageName] projectName
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.mvc.portlet \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DliferayVersion=7.0
```

You can also insert the `-b maven` parameter in the Blade command to generate a Maven project using Blade CLI.

The template for this kind of project is `war-mvc-portlet`. Suppose you want to create a WAR MVC portlet project called `my-war-mvc-portlet-project` with a package name of `com.liferay.docs.war.mvc` and a class name of `MyWarMvcPortlet`. You could run the following command to accomplish this:

```
blade create -t war-mvc-portlet -v 7.0 -p com.liferay.docs.war.mvc my-war-mvc-portlet-project
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.war.mvc.portlet \
    -DgroupId=com.liferay \
    -DartifactId=my-war-mvc-portlet-project \
    -Dpackage=com.liferay.docs.war.mvc \
    -Dversion=1.0 \
    -DliferayVersion=7.0
```

After running the command above, your project's folder structure looks like this:

- `my-war-mvc-portlet-project`
  - `[gradle|.mvn]`
    - `wrapper`
      - `[gradle|maven]-wrapper.jar`
      - `[gradle|maven]-wrapper.properties`
  - `src`
    - `main`
      - `java`
      - `com/liferay/docs/war/mvc`

      - `resources`
      - `content`
      - `Language.properties`

      - `webapp`
      - `css`
      - `.sass-cache`
      - `main.css`
      - `main_rtl.css`

      - `main.scss`

      - `WEB-INF`
      - `tld`
      - `liferay-portlet.tld`
      - `liferay-portlet-ext.tld`
      - `liferay-security.tld`
      - `liferay-theme.tld`
      - `liferay-ui.tld`
      - `liferay-util.tld`

      - `liferay-display.xml`
      - `liferay-plugin-package.properties`
      - `liferay-portlet.xml`

        · `portlet.xml`
        · `web.xml`

        · `init.jsp`
        · `view.jsp`

  – `[build.gradle|pom.xml]`
  – `[gradlew|mvnw]`

The generated WAR MVC portlet is functional and is deployable to a Liferay DXP instance. To build upon the generated project, modify the project by adding logic and additional files to the folders outlined above. Deploying WAR MVC portlets is supported for 7.0, however, it is recommended to optimize your WAR portlet to a module project, if possible. You can visit the From Liferay Portal 6 to 7 section for info on how to do this.

Chapter 134

Sample Projects

Note: This section of articles does not provide documentation for *all* sample projects residing in the `liferay-blade-samples` repo. The documentation for these samples is in progress and will grow over time.

Liferay provides sample projects that target different integration points in Liferay DXP. These projects reside in the liferay-blade-samples Github repository and can be easily copy/pasted to your local environment. The sample projects are grouped into three different parent folders based on the build tools used to generate them:

- `gradle`
- `liferay-workspace`
- `maven`

Note: The Liferay Workspace folder stores WAR-type samples in a separate folder named wars. The Gradle and Maven tool folders mix WAR samples with the other sample types (apps, extensions, etc.).

For more information on these sample projects, visit the Liferay Sample Projects tutorial.

Chapter 135

Apps

This section focuses on Liferay sample applications. You can view these sample apps by visiting the apps folder corresponding to your preferred build tool:

- Gradle sample apps
- Liferay Workspace sample apps
- Maven sample apps

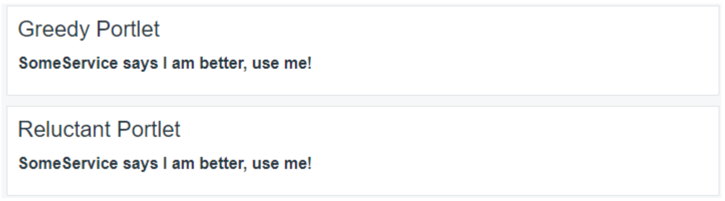
The following samples are documented:

- Greedy Policy Option Portlet
- Kotlin Portlet
- npm Samples
- Service Builder Samples
- Shared Language Keys
- Simulation Panel App
- Spring MVC Portlet

Visit a particular sample page to learn more!

135.1 Greedy Policy Option Application

The Greedy Policy Option sample provides two portlets that can be added to a Liferay DXP page: Greedy Portlet and Reluctant Portlet.



```
SomeService says I am better, use me!
```

```
SomeService says I am better, use me!
```

Figure 135.1: The Greedy Policy Option app provides two portlets that only print text. You'll dive deeper later to discover their interesting capabilities involving services.

These two portlets do not provide anything useful out-of-the-box. They are, however, very effective at demonstrating the ability to reference services using greedy and reluctant policy options. You'll learn how to do this later.

What API(s) and/or code components does this sample highlight?

This sample provides two modules referencing services using greedy and reluctant policy options.

- `service-reference`: Provides an OSGi service interface called `SomeService`, a default implementation of it, and portlets that refer to service instances. One portlet refers to new higher ranked instances of the service automatically. The other portlet is reluctant to use new higher ranked instances and continues to use its bound service. The reluctant portlet can, however, be configured dynamically to use other service instances.
- `higher-ranked-service`: Has a higher ranked `SomeService` implementation.

Here are each module's file structures:

- `service-reference/`
  - `bnd.bnd`
  - `configs/`
    - `com.liferay.blade.reluctant.vs.greedy.portlet.portlet.ReluctantPortlet.config` → `ReluctantPortlet` configuration file for Liferay DXP DE 7.0 Fix Pack 8 or later and Liferay CE Portal 7.0 GA4 or later
    - `com.liferay.blade.reluctant.vs.greedy.portlet.portlet.ReluctantPortlet.cfg` → `ReluctantPortlet` configuration file for Liferay DXP DE 7.0 Fix Packs earlier than Fix Pack 8 and Liferay CE Portal 7.0 GA3 or earlier
  - `src/main/java/com/liferay/blade/reluctant/vs/greedy/portlet/`
    - `api/`
      - `SomeService.java` → Service interface
    - `constants/`
      - `ReluctantPortletVsGreedyPortletKeys.java` → Portlet constants
    - `portlet/`
      - `DefaultSomeService.java` → Zero ranked service implementation
      - `GreedyPortlet.java` → Refers to `SomeService` using a greedy service policy option
      - `ReluctantPortletPortlet.java` → Refers to `SomeService` using a reluctant service policy option by default.
- `higher-ranked-service/`
  - `bnd.bnd`
  - `src/main/java/com/liferay/blade/reluctant/vs/greedy/svc/HigherRankedService.java` → Service implementation with service ranking value of `100`

How does this sample leverage the API(s) and/or code component?

Here are the things you can learn using the sample modules:

1. Binding a component's service reference to the highest ranked service instance that's available initially.
2. Deploying a module with a higher ranked service instance for binding to greedy references immediately.
3. Configuring a component to reference a different service instance dynamically.

Let's walk through the demonstration.

Binding a newly deployed component's service reference to the highest ranking service instance that's available initially

On deploying a component that references a service, it binds to the highest ranking service instance that matches its target filter (if specified).

The portlet classes refer to instances of interface `SomeService`. The `doSomething` method returns a `String`.

```
public interface SomeService {

    public String doSomething();

}
```

Class `DefaultSomeService` implements `SomeService`. Its `doSomething` method returns the `String` "I am Default!".

```
@Component
public class DefaultSomeService implements SomeService {

    @Override
    public String doSomething() {
        return "I am Default!";
    }

}
```

When module's portlets refer to `DefaultSomeService`, they display the `String` "I am Default!".

The `ReluctantPortlet` class's `SomeService` reference's policy option is the default: static and reluctant. This policy option keeps the reference bound to its current service instance unless that instance stops or the reference is reconfigured to refer to a different service instance.

```
@Component(
   immediate = true,
   property = {
       "com.liferay.portlet.display-category=category.sample",
       "com.liferay.portlet.instanceable=true",
       "javax.portlet.display-name=Reluctant Portlet",
       "javax.portlet.init-param.template-path=/",
       "javax.portlet.init-param.view-template=/view.jsp",
       "javax.portlet.name=" + ReluctantVsGreedyPortletKeys.Reluctant,
       "javax.portlet.resource-bundle=content.Language",
       "javax.portlet.security-role-ref=power-user,user"
   },
   service = Portlet.class
)
public class ReluctantPortlet extends MVCPortlet {
   @Override
   public void doView(
           RenderRequest renderRequest, RenderResponse renderResponse)
       throws IOException, PortletException {

       renderRequest.setAttribute("doSomething", _someService.doSomething());

       super.doView(renderRequest, renderResponse);
   }

   @Reference
   private SomeService _someService;

}
```

The `ReluctantPortlet`'s method `doView` sets render request attribute `doSomething` to the value returned from the `SomeService` instance's `doSomething` method (e.g., `DefaultService` returns "I am default!").

The `GreedyPortlet` class is similar to `ReluctantPortlet`, except its `SomeService` reference's policy option is static and greedy (i.e., `ReferencePolicyOption.GREEDY`).

```
public class GreedyPortlet extends MVCPortlet {

    @Override
    public void doView(
            RenderRequest renderRequest, RenderResponse renderResponse)
        throws IOException, PortletException {

        renderRequest.setAttribute("doSomething", _someService.doSomething());

        super.doView(renderRequest, renderResponse);
    }

    @Reference (policyOption = ReferencePolicyOption.GREEDY)
    private SomeService _someService;

}
```

The greedy policy option lets the component switch to using a higher ranked `SomeService` instance if one becomes active in the system. The section *Deploying a module with a higher ranked service instance for binding to greedy references immediately* demonstrates this portlet switching to a higher ranked service.

It's time to see this module's portlets and service in action.

1. Stop module `higher-ranked-service` if it's active.

2. Deploy the `service-reference` module.

3. Add the *Reluctant Portlet* from the *Add* → *Applications* → *Sample* category to a site page.

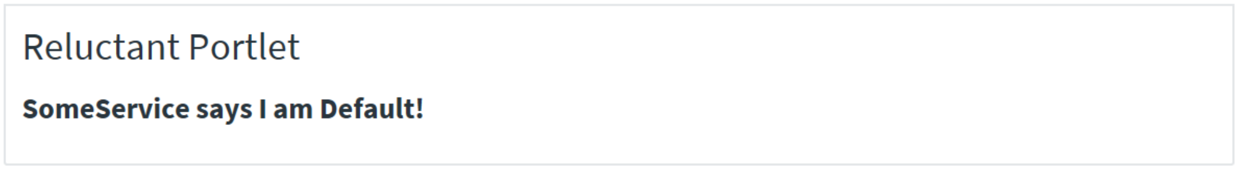
   The portlet displays the message "SomeService says I am default!"–whose latter part comes from the render request attribute set by the `DefaultService` instance.

4. Add the *Greedy Portlet* from the *Add* → *Applications* → *Sample* category to a site page.

   The portlet displays the message "SomeService says I am better, use me!". Both portlets are referencing a `DefaultService` instance.

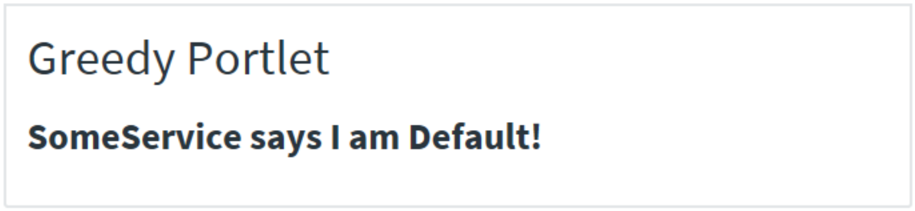
Since `DefaultService` is the only active `SomeService` instance in the system, the portlets refer to it for their `SomeService` fields.

The `DefaultService` and portlets *Reluctant Portlet* and *Greedy Portlet* are active. Let's activate a higher ranked `SomeService` instance and see how the portlets react to it.


Reluctant Portlet

SomeService says I am Default!

Figure 135.2: *Reluctant Portlet* displays the message “SomeService says I am default!”


Greedy Portlet

SomeService says I am Default!

Figure 135.3: *Greedy Portlet* displays the message “SomeService says I am better, use me!”

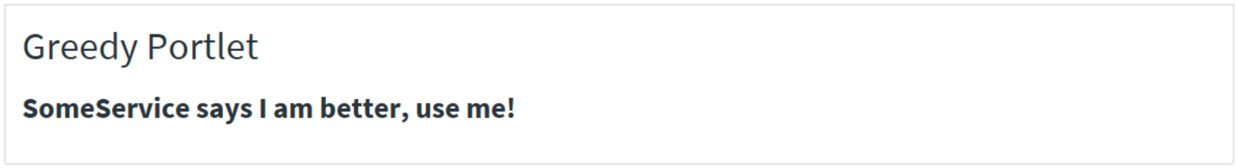
Deploying a module with a higher ranked service instance for binding to greedy references immediately

Module `higher-ranked-service` provides a `SomeService` implementation called `HigherRankedService`. `HigherRankedService`’s service ranking is `100`–that’s `100` more than `DefaultService`’s ranking `0`. Its `doSomething` method returns the `String` “I am better, use me!”.

1. Deploy the `higher-ranked-service` module.
2. Refresh your page that has the portlets *Reluctant Portlet* and *Greedy Portlet*.

Reluctant Portlet continues displaying message “SomeService says I am better, use me!”. It’s “reluctant” to unbind from the `DefaultService` instance and bind to the newly activated `HigherRankedService` service.

Greedy Portlet displays a new message “SomeService says I am better, use me!”. The part of the message “I am better, use me!” comes from the `HigherRankedService` instance to which it refers.


Greedy Portlet

SomeService says I am better, use me!

Figure 135.4: The *Greedy Portlet* is using a `HigherRankedService` instance

Next, learn how to bind the *Reluctant Portlet* to a `HigherRankedService` instance.

Configuring a component to reference a different service instance dynamically

The *Reluctant Portlet* is currently bound to a `DefaultService` instance. It’s “reluctant” to unbind from it and bind to a different service. OSGi Configuration Administration lets you reconfigure service references to filter on and bind to different service instances.

The `service-reference` module's configuration files and `com.liferay.blade.reluctant.vs.greedy.portlet.portlet.Relucta`
and `com.liferay.blade.reluctant.vs.greedy.portlet.portlet.ReluctantPortlet.cfg` configure the `ReluctantPortlet` component to use a `HigherRankedService` instance.

```
_someService.target=(component.name=com.liferay.blade.reluctant.vs.greedy.service.HigherRankedService)
```

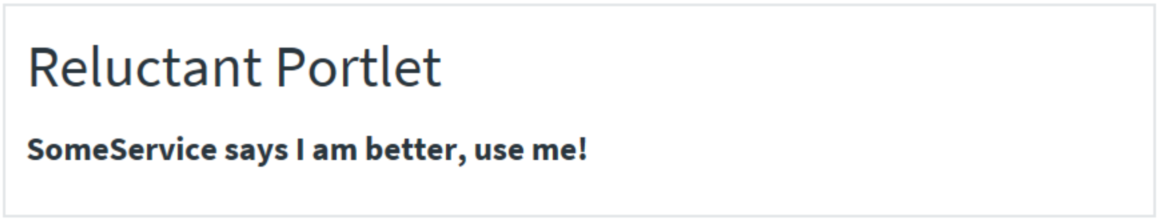
The service configuration filters on a service whose `component.name` is `com.liferay.blade.reluctant.vs.greedy.service.Hig`

Note: For deploying to Liferay DXP DE 7.0 Fix Pack 8 or later or Liferay CE Portal 7.0 GA4 or later, use file with suffix `.config`. For earlier versions, use the file with suffix `.cfg`.

Here are the steps to reconfigure `ReluctantPortlet` to use `HigherRankedService`:

1. Copy the configuration file to `[Liferay-Home]/osgi/configs`.
2. Refresh your browser.

Reluctant Portlet displays a new message "SomeService says I am better, use me!".


Reluctant Portlet

SomeService says I am better, use me!

Figure 135.5: *Reluctant Portlet* is using the `HigherRankedService` instance instead of a `DefaultService` instance.

Reluctant Portlet is using `HigherRankedService` instance instead of a `DefaultService` instance. You've configured *Reluctant Portlet* to use a `HigherRankedService` instance!

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

135.2 Kotlin Portlet

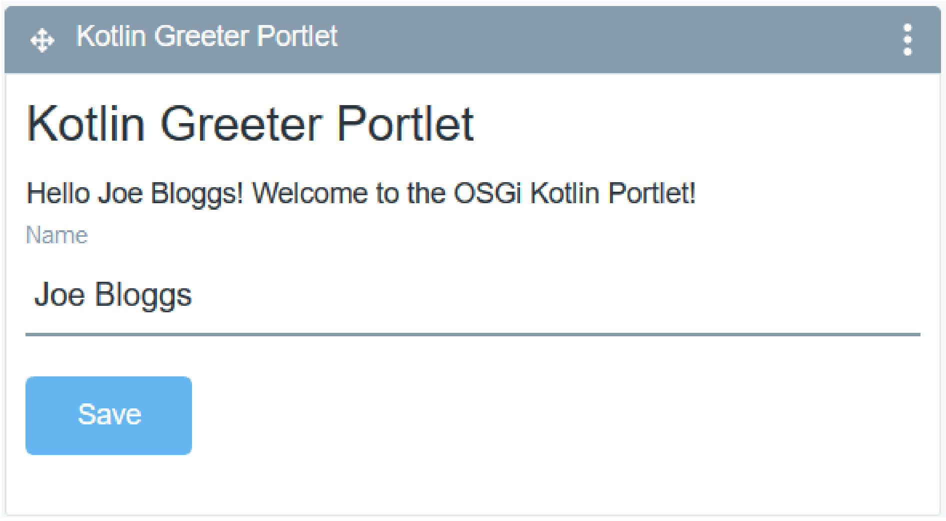
The Kotlin Portlet sample provides an input form that accepts a name. Once submitting a name, the portlet renders a greeting message.

What API(s) and/or code components does this sample highlight?

This sample highlights the use of the Kotlin programming language in conjunction with Liferay's MVC framework. Specifically, this sample leverages the MVCActionCommand interface.

How does this sample leverage the API(s) and/or code component?

This sample uses the MVC Action Command's `processAction(...)` method to process the inputted text (i.e., name). The text is set as an attribute in the `KotlinGreeterActionCommandKt.kt` class using an `ActionRequest` and then is retrieved in the JSP using a `RenderRequest`.



Kotlin Greeter Portlet

Hello Joe Bloggs! Welcome to the OSGi Kotlin Portlet!

Joe Bloggs

Figure 135.6: After saving the inputted name, it's is displayed as a greeting on the portlet page.

Where Is This Sample?

This sample is built with the following build tools:

- Gradle
- Liferay Workspace

CHAPTER 136

npm Samples

This section focuses on Liferay npm sample portlets built with Gradle. You can view these samples by visiting the gradle/apps/npm folder in the `liferay-blade-samples` Github repository.

The following npm samples are documented:

- Angular npm Portlet
- Billboard.js npm Portlet
- Isomorphic npm Portlet
- jQuery npm Portlet
- Metal.js npm Portlet
- React npm Portlet
- Simple npm Portlet
- Vue.js npm Portlet

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

Visit a particular sample page to learn more!

136.1 Angular npm Portlet

The Angular npm Portlet sample provides a portlet that uses the Angular framework to render its output.

This portlet showcases Angular's speed and performance when rendering a user interface.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.

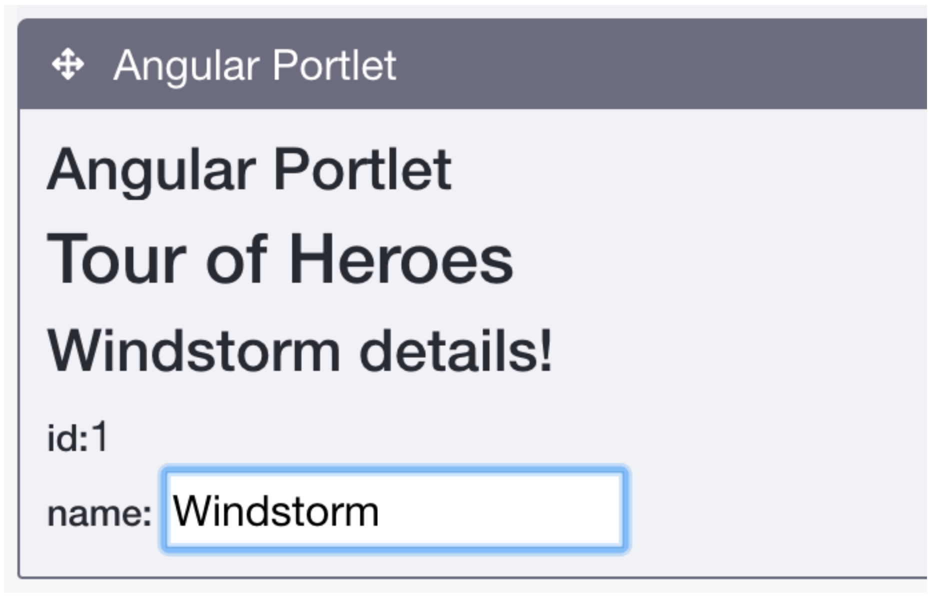


Figure 136.1: Type custom text in the field and watch it instantaneously displayed in the portlet.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

How does this sample leverage the API(s) and/or code component?

This sample uses the npm registry to download project dependencies and uses the liferay-npm-bundler tool to bundle the project dependencies inside the OSGi bundle JAR file.

To accomplish the bundling, the project's build process relies on a `build` script inside its `package.json` file:

```
"scripts": {
    "build": "tsc && liferay-npm-bundler"
},
```

Where Is This Sample?

This sample is built with the following build tool:

- Gradle

136.2 Billboard.js npm Portlet

The Billboard.js npm Portlet sample provides a portlet that uses the Billboard.js framework to render its output.

This portlet showcases the power of graphing by displaying a set of default charts and a more advanced custom chart. These are all built using Billboard.js.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

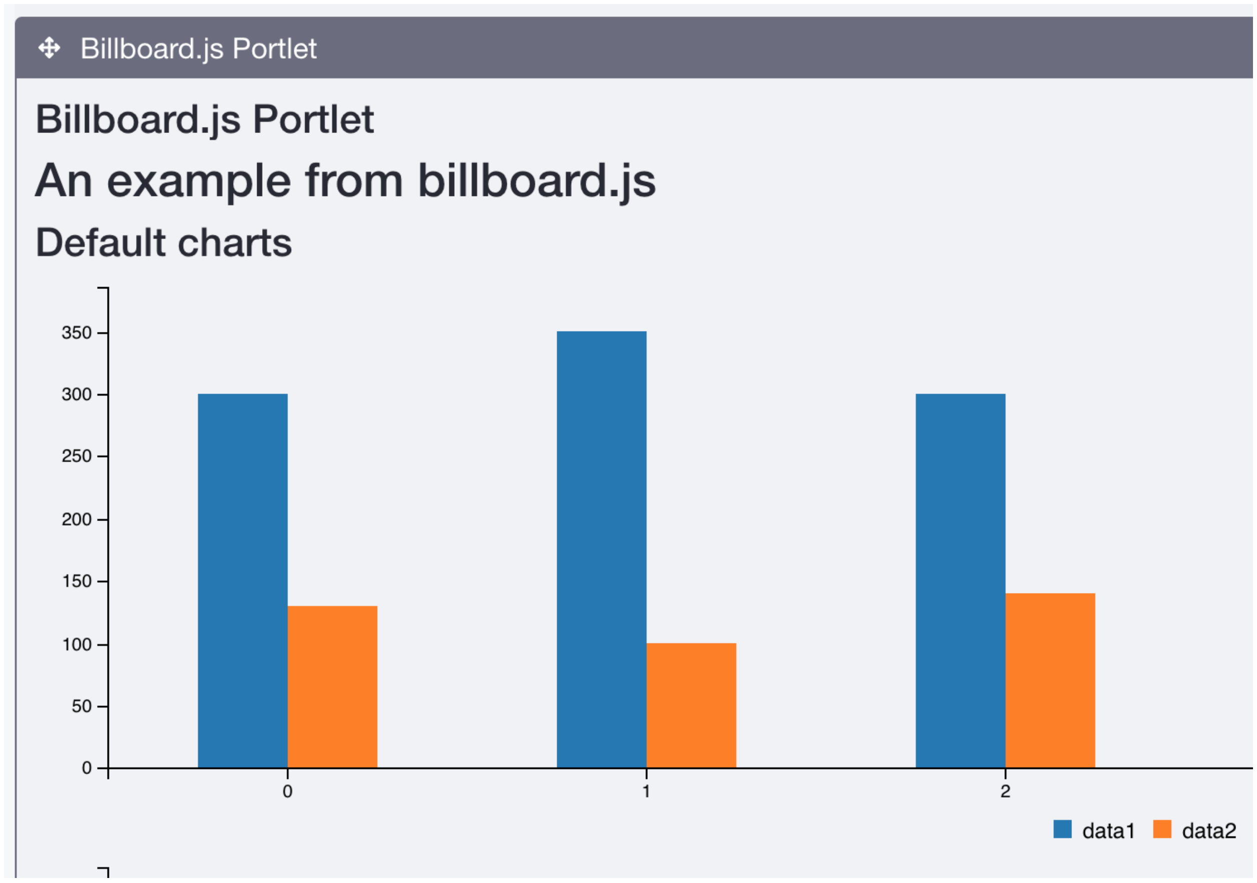


Figure 136.2: The Billboard.js npm Portlet shows off some nice looking graphs using Billboard.js.

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

How does this sample leverage the API(s) and/or code component?

This sample uses the npm registry to download project dependencies and uses the liferay-npm-bundler tool to bundle the project dependencies inside the OSGi bundle JAR file.

To accomplish the bundling, the project's build process relies on a `build` script inside its `package.json` file:

```
"scripts": {
    "build": "babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-INF/resources && liferay-npm-
bundler"
},
```

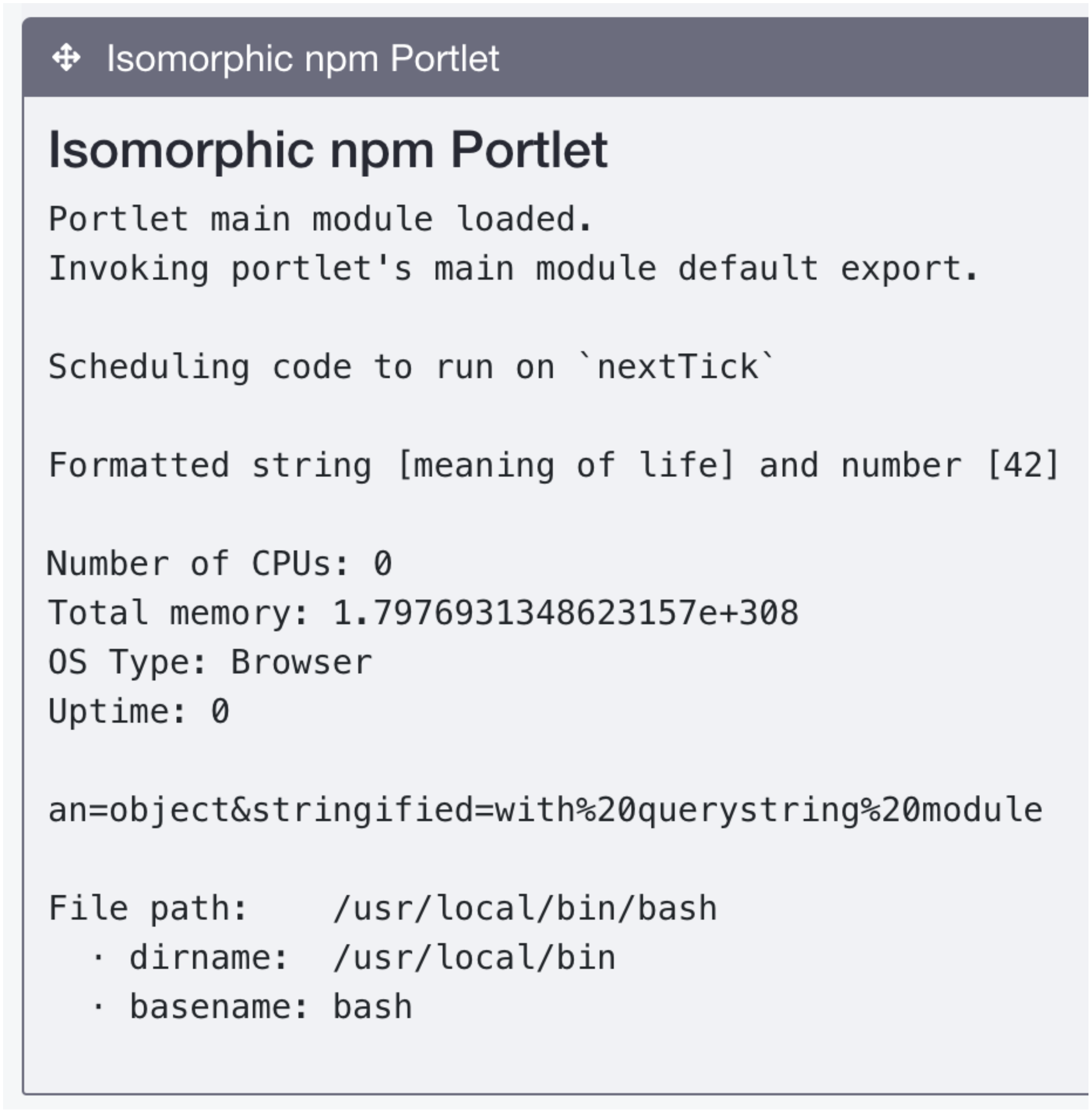
Where Is This Sample?

This sample is built with the following build tool:

- Gradle

136.3 Isomorphic npm Portlet

The Isomorphic npm Portlet sample provides a portlet that uses isomorphic code (i.e., can run from client and/or server side) on the client side.



Isomorphic npm Portlet

```
Portlet main module loaded.
Invoking portlet's main module default export.

Scheduling code to run on `nextTick`

Formatted string [meaning of life] and number [42]

Number of CPUs: 0
Total memory: 1.7976931348623157e+308
OS Type: Browser
Uptime: 0

an=object&stringified=with%20querystring%20module

File path:    /usr/local/bin/bash
   · dirname:  /usr/local/bin
   · basename: bash
```

Figure 136.3: This sample portlet displays the results of running code designed for the server in the browser.

This portlet showcases running code designed to execute in the server in the browser. Note that this portlet does **not** run JavaScript code in the server; it's executing isomorphic JavaScript code in the browser.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

You can do many things with isomorphic code. You can run it in

- the server only (e.g., Node.js)
- the client only (e.g., browser)
- both the server and client (e.g., Node.js + browser)

Isomorphic code cannot run server-side because Liferay DXP is Java based and cannot execute JavaScript that way. This sample portlet shows how Liferay's npm bundler can transform server-side code to make it work in the client (e.g., emulates some of Node.js' APIs in the client).

How does this sample leverage the API(s) and/or code component?

This sample uses the npm registry to download project dependencies and uses the liferay-npm-bundler tool to bundle the project dependencies inside the OSGi bundle JAR file.

To accomplish the bundling, the project's build process relies on a `build` script inside its `package.json` file:

```
"scripts": {
    "build": "babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-INF/resources && liferay-npm-bundler"
},
```

Where Is This Sample?

This sample is built with the following build tool:

- Gradle

136.4 jQuery npm Portlet

The jQuery npm Portlet sample provides a portlet that uses the jQuery framework to render its output.

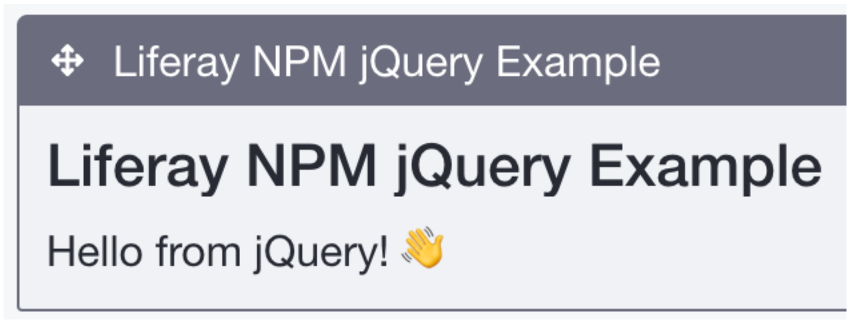
This portlet showcases the fast HTML document traversal jQuery offers.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.



Liferay NPM jQuery Example

Hello from jQuery! 👋

Figure 136.4: Clicking on the portlet's hand symbol displays a message.

How does this sample leverage the API(s) and/or code component?

This sample uses the npm registry to download project dependencies and uses the liferay-npm-bundler tool to bundle the project dependencies inside the OSGi bundle JAR file.

To accomplish the bundling, the project's build process relies on a `build` script inside its `package.json` file:

```
"scripts": {
    "build": "babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-INF/resources && liferay-npm-
bundler"
},
```

Where Is This Sample?

This sample is built with the following build tool:

- Gradle

136.5 Metal.js npm Portlet

The Metal.js npm Portlet sample provides a portlet that uses the Metal.js framework to render its output.

This portlet displays a Metal.js based dialog that has been rendered using SOY templates.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.

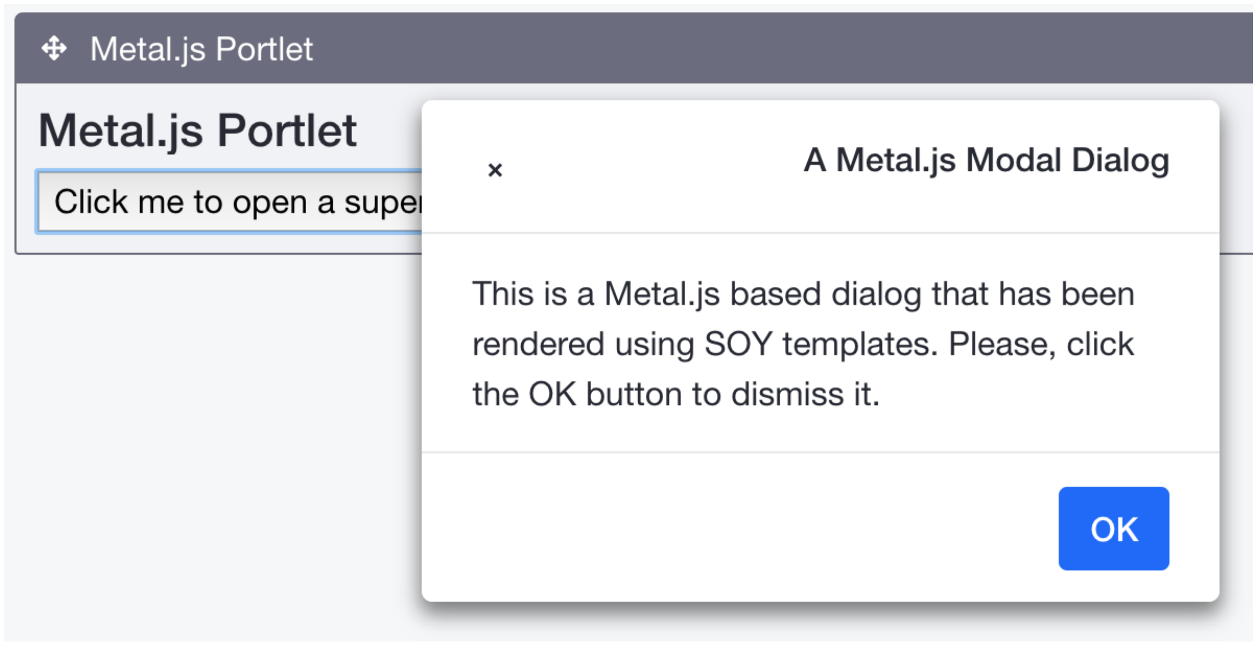
What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

How does this sample leverage the API(s) and/or code component?

This sample uses the npm registry to download project dependencies and uses the liferay-npm-bundler tool to bundle the project dependencies inside the OSGi bundle JAR file.

To accomplish the bundling, the project's build process relies on a `build` script inside its `package.json` file:



This is a Metal.js based dialog that has been rendered using SOY templates. Please, click the OK button to dismiss it.

Figure 136.5: Clicking the button returns displays a dialog window.

```
"scripts": {
    "build": "metalsoy && babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-INF/resources && liferay-
npm-bundler"
},
```

Where Is This Sample?

This sample is built with the following build tool:

- Gradle

136.6 React npm Portlet

The React npm Portlet sample provides a portlet that uses the React framework to render its output.

This portlet showcases the how efficiently React can render components based on user interaction.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

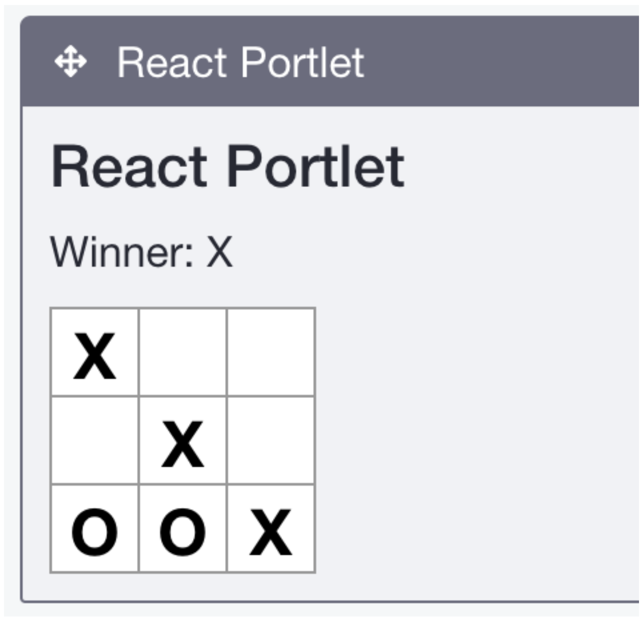


Figure 136.6: You can play the game Tic-tac-toe with this sample portlet.

How does this sample leverage the API(s) and/or code component?

This sample uses the npm registry to download project dependencies and uses the liferay-npm-bundler tool to bundle the project dependencies inside the OSGi bundle JAR file.

To accomplish the bundling, the project's build process relies on a `build` script inside its `package.json` file:

```
"scripts": {
    "build": "babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-INF/resources && liferay-npm-
bundler"
},
```

Where Is This Sample?

This sample is built with the following build tool:

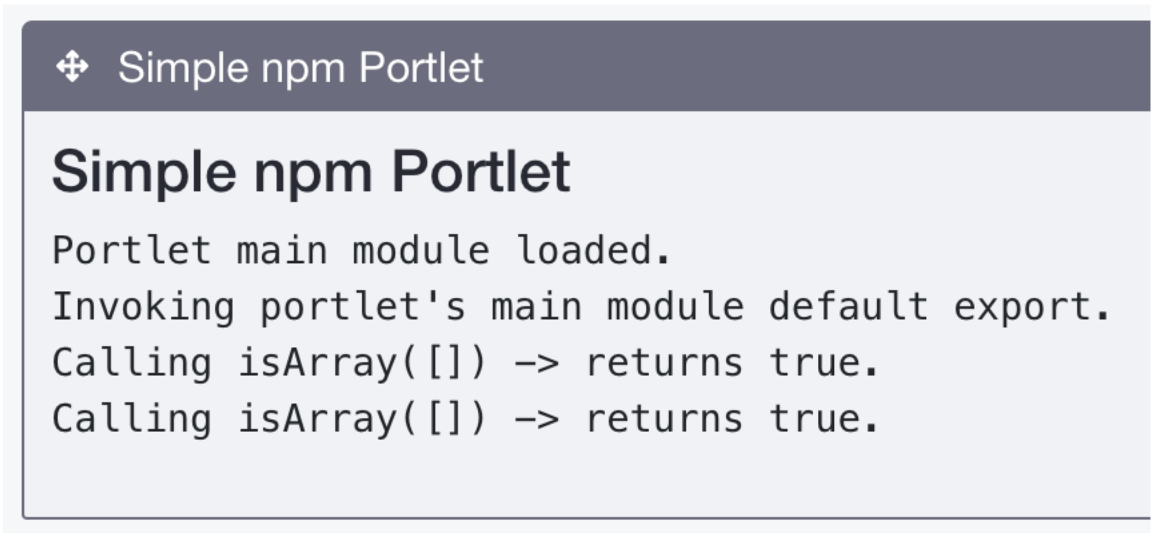
- Gradle

136.7 Simple npm Portlet

The Simple npm Portlet sample provides a portlet that uses the isarray npm package when rendering its output.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.



Simple npm Portlet

```
Portlet main module loaded.
Invoking portlet's main module default export.
Calling isArray([]) -> returns true.
Calling isArray([]) -> returns true.
```

Figure 136.7: The portlet's status and actions are displayed as output.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

How does this sample leverage the API(s) and/or code component?

This sample uses the npm registry to download project dependencies and uses the liferay-npm-bundler tool to bundle the project dependencies inside the OSGi bundle JAR file.

To accomplish the bundling, the project's build process relies on a `build` script inside its `package.json` file:

```
"scripts": {
    "build": "babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-INF/resources && liferay-npm-
bundler"
},
```

Where Is This Sample?

This sample is built with the following build tool:

- Gradle

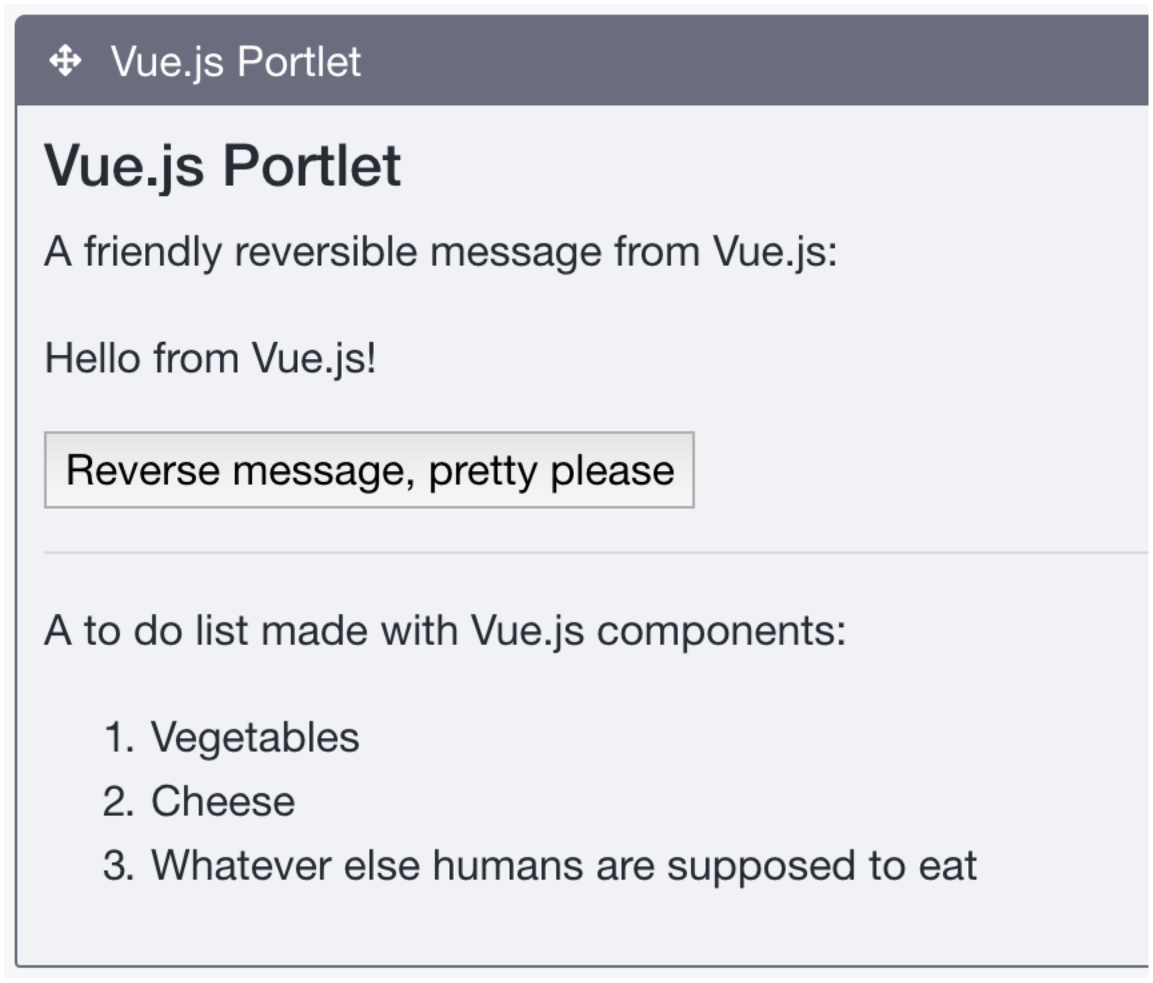
136.8 Vue.js npm Portlet

The Vue.js npm Portlet sample provides a portlet that uses the Vue.js framework to render its output.

This portlet showcases Vue.js's speed and performance when rendering a user interface.

Note: The minifier fails on Liferay DXP 7.0 when JSDoc is present in a portlet. To resolve this, use Grunt uglify to remove the JSDoc comments. This process may take a long time, depending on the number of files that require an update.

Important: This sample works for Liferay DXP 7.0 Fix Pack 44+ and Liferay Portal CE GA7+.


Vue.js Portlet

Vue.js Portlet

A friendly reversible message from Vue.js:

Hello from Vue.js!

A to do list made with Vue.js components:

1. Vegetables
2. Cheese
3. Whatever else humans are supposed to eat

Figure 136.8: Clicking the portlet's button reverses the message.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

How does this sample leverage the API(s) and/or code component?

This sample uses the npm registry to download project dependencies and uses the liferay-npm-bundler tool to bundle the project dependencies inside the OSGi bundle JAR file.

To accomplish the bundling, the project's build process relies on a `build` script inside its `package.json` file:

```
"scripts": {
    "build": "babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-INF/resources && liferay-npm-
bundler"
},
```

Where Is This Sample?

This sample is built with the following build tool:

- Gradle

Chapter 137

Service Builder Samples

This section focuses on Liferay Service Builder sample projects built with various build tools. You can view these samples by visiting the `apps/service-builder` folder corresponding to your preferred build tool:

- Gradle Service Builder sample apps
- Liferay Service Builder Workspace sample apps
- Maven Service Builder sample apps

The following Service Builder samples are documented:

- Service Builder application demonstrating Actionable Dynamic Query
- Service Builder application with JDBC connection
- Service Builder application with JNDI connection

Visit a particular sample page to learn more!

137.1 Service Builder Application Demonstrating Actionable Dynamic Query

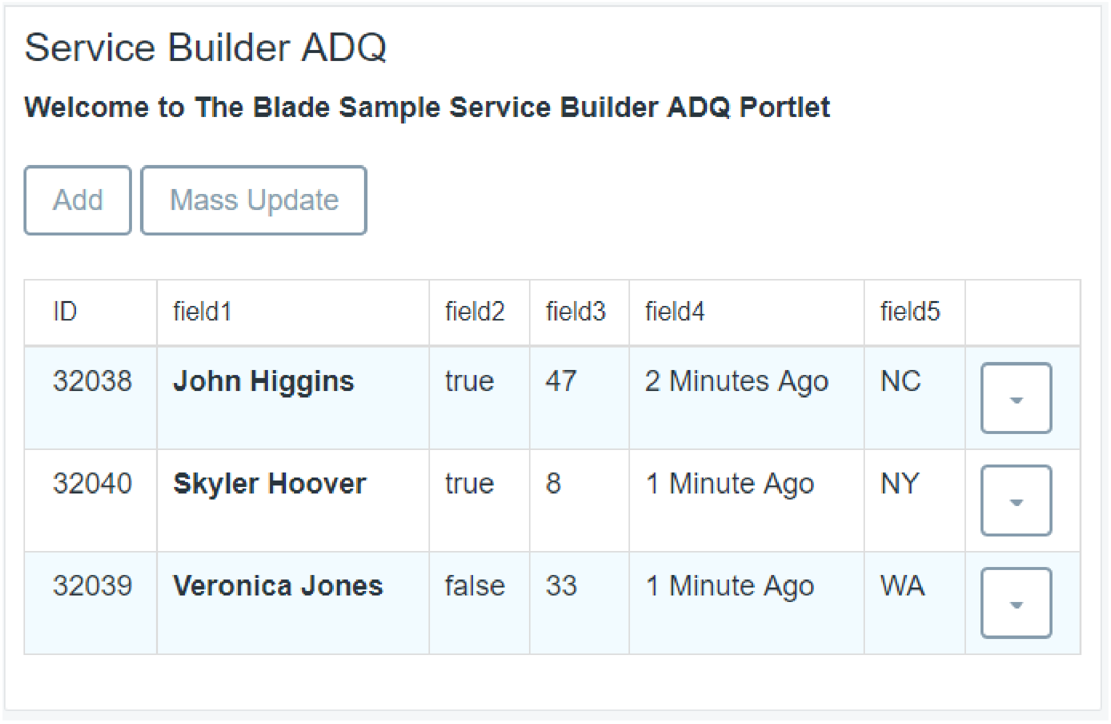
This sample is similar to the `basic` Service Builder sample, which lets you perform CRUD (create, read, update, delete) operations on service builder entities. The distinctive feature of the Service Builder Actionable Dynamic Query (ADQ) sample is that it also lets you perform a mass update on all existing service builder entities.

To see the ADQ Service Builder sample in action, complete the following steps:

1. Add the sample to a page by navigating to *Add* (➕) → *Applications* → *Sample* and dragging it to the page.
2. Select the app's *Add* button and add an entity. Do this several times to create multiple entities.
3. Click the *Mass Update* button and click *Save* to invoke the update.

   After invoking the update, each entity's `field3` value (whose value is less than 100) is incremented.

You've leveraged the actionable dynamic query API in your sample!

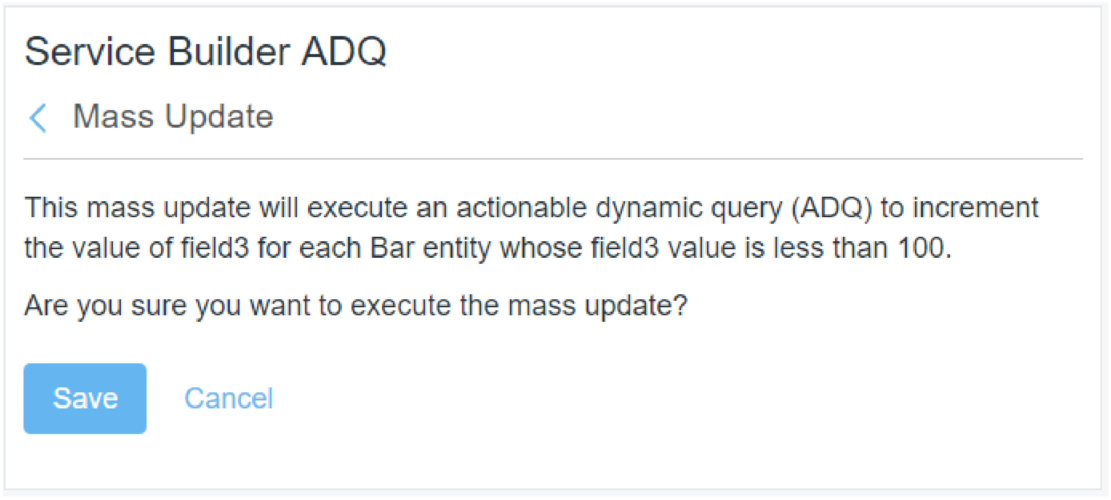


Service Builder ADQ

Welcome to The Blade Sample Service Builder ADQ Portlet

ID	field1	field2	field3	field4	field5	
32038	**John Higgins**	true	47	2 Minutes Ago	NC	
32040	**Skyler Hoover**	true	8	1 Minute Ago	NY	
32039	**Veronica Jones**	false	33	1 Minute Ago	WA	

Figure 137.1: This sample provides options to add entities and perform a mass update.



Service Builder ADQ

This mass update will execute an actionable dynamic query (ADQ) to increment the value of field3 for each Bar entity whose field3 value is less than 100.

Are you sure you want to execute the mass update?

Figure 137.2: Clicking the *Save* button executes the mass update.

What API(s) and/or code components does this sample highlight?

This sample demonstrates Liferay DXP's actionable dynamic query API. Specifically, it demonstrates how to create an ADQ, add criteria to an ADQ, specify an action for the ADQ, and execute the ADQ.

How does this sample leverage the API(s) and/or code component?

An action request is sent to the `JSPPortlet` with a `cmd` request parameter. When the `JSPPortlet`'s `processAction` method processes the request, the value of the `cmd` parameter is parsed and then the portlet's `massUpdate` method is invoked. The `massUpdate` method, in turn, invokes the `massUpdate` method defined in the `adq-service` module's `BarLocalServiceImpl`. This is where the sample leverages the actionable dynamic query API:

```
public void massUpdate() {
    ActionableDynamicQuery adq = getActionableDynamicQuery();

    adq.setAddCriteriaMethod(
        new ActionableDynamicQuery.AddCriteriaMethod() {

            @Override
            public void addCriteria(DynamicQuery dynamicQuery) {
                dynamicQuery.add(RestrictionsFactoryUtil.lt("field3", 100));
            }

        });

    adq.setPerformActionMethod(
        new ActionableDynamicQuery.PerformActionMethod<Bar>() {

            @Override
            public void performAction(Bar bar) {
                int field3 = bar.getField3();

                field3++;
                bar.setField3(field3);

                updateBar(bar);
            }

        });

    try {
        adq.performActions();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

For more information on the actionable dynamic query API, visit its dedicated tutorial.

137.2 Service Builder Application Using External Database via JDBC

This sample demonstrates how to connect a Liferay Service Builder application to an external database via a JDBC connection. Here, an external database means any database other than Liferay DXP's database. For this sample to work correctly, you must prepare such an external database and configure Liferay DXP to use it. Follow the steps below to make the required preparations before deploying the application.

1. Create the external database to which your Service Builder application will connect. For example, create a MariaDB database called `external`. Add a table to this database called `country` with a `BIGINT`

column called Id and a VARCHAR(255) column called Name. Add at least one record to this table. Here are the MariaDB commands to accomplish this:

```
create database external character set utf8;

use external;

create table country(id bigint not null primary key, name varchar(255));

insert into country(id, name) values(1, 'Australia');
```

Make sure that your database commands were successful: Running select * from country; should return the record you added.

2. Create a portal-ext.properties file in your Liferay DXP instance's [LIFERAY_HOME] folder (this folder should be marked by the presence of a .liferay-home file). In your portal-ext.properties file, define the details of your JDBC data source connection:

```
jdbc.ext.driverClassName=org.mariadb.jdbc.Driver
jdbc.ext.password=userpassword
jdbc.ext.url=jdbc:mariadb://localhost/external?useUnicode=true&characterEncoding=UTF-8&useFastDateParsing=false
jdbc.ext.username=yourusername
```

Note that Liferay DXP's primary data source is specified by the jdbc.default prefix. These details are often specified in a portal-setup-wizard.properties file. Here, we've chosen to use the jdbc.ext prefix for our alternate data source.

3. Create a com.liferay.blade.samples.jdbcservicebuilder.service-log4j-ext.xml in your Liferay instance's [LIFERAY_HOME]/osgi/log4 folder. Create this folder if it doesn't yet exist. Add this content to the XML file that you created:

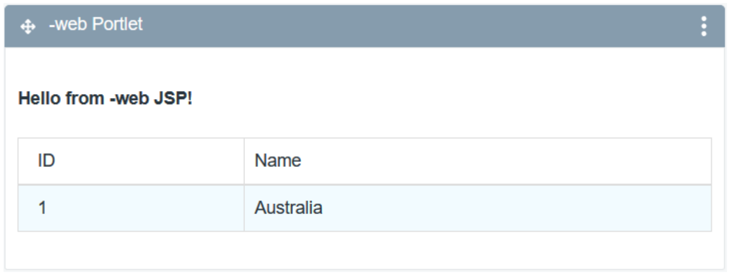
```
<?xml version="1.0"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <category name="com.liferay.blade.samples.jdbcservicebuilder.service.impl">
        <priority value="INFO" />
    </category>
</log4j:configuration>
```

This XML file defines the log level for the classes in the com.liferay.blade.samples.jdbcservicebuilder.service.impl package. The com.liferay.blade.samples.jdbcservicebuilder.service.impl.CountryLocalServiceImpl is the class that will produce log messages when the sample portlet is viewed.

Now your sample is ready for deployment! Make sure to build and deploy each of the three modules that comprise the sample application:

- jdbc-api
- jdbc-service
- jdbc-web

After these modules have been deployed, add the -web portlet to a Liferay DXP page.
A sample table is printed in the portlet's view, representing the info inputted into the database.


-web Portlet

Hello from -web JSP!

ID	Name
1	Australia

Figure 137.3: This sample prints out the values previously inputted into the database.

What API(s) and/or code components does this sample highlight?

This sample demonstrates two ways to access data from an external database defined by a JDBC connection:

- extract data directly from the raw data source by explicitly specifying a SQL query.
- read data using the helper methods that Service Builder generates in your application's persistence layer.

How does this sample leverage the API(s) and/or code component?

Once you've added the `-web` portlet to a page, the `CountryLocalService.useJDBC` method is invoked. This method accesses the database defined by the JDBC connection you specified and logs information about the rows in the `country` table to Liferay DXP's log.

The first way of accessing data from the external database is to extract it directly from the raw data source by explicitly specifying a SQL query. This technique is demonstrated by the `CountryLocalServiceImpl.useJDBC` method. That method obtains the Spring-defined data source that's injected into the `countryPersistence` bean, opens a new connection, and reads data from the data source. This is the technique used by the sample application to write the data to Liferay DXP's log.

The second way of accessing data from the external database is to read data using the helper methods that Service Builder generates in your application's persistence layer. This technique is demonstrated by the `UseJDBC.getCountries` method which first obtains an instance of the `CountryLocalService` OSGi service and then invokes `countryLocalService.getCountries`. The `countryLocalService.getCountries` and `countryLocalService.getCountriesCount` methods are two examples of the persistence layer helper methods that Service Builder generates. This is the technique used by the sample application to actually display the data. The portlet's `view.jsp` uses the `<search-container>` JSP tag to display a list of results. The results are obtained by the `UseJDBC.getCountries` method mentioned above.

137.3 Service Builder Application Using External Database via JNDI

This sample demonstrates how to connect a Liferay Service Builder application to an external database via a JNDI connection. Here, an external database means any database other than Liferay DXP's database. For this sample to work correctly, you must prepare such an external database and configure Liferay DXP to use it. Follow the steps below to make the required preparations before deploying the application.

1. Create the external database to which your Service Builder application will connect. For example, create a MariaDB database called `external`. Add a table to this database called `region` with a `BIGINT` column called `Id` and a `VARCHAR(255)` column called `Name`. Add at least one record to this table. Here are the MariaDB commands to accomplish this:

```
create database external character set utf8;

use external;

create table region(id bigint not null primary key, name varchar(255));

insert into region(id, name) values(1, 'Tasmania');
```

Make sure that your database commands were successful: Running `select * from region;` should return the record you added.

2. Now you need to define a JNDI connection to your database. The way this is done depends on your application server. Here we demonstrate how to specify the JNDI connection for Tomcat. First, edit your `[LIFERAY_HOME]/tomcat-8.0.32/conf/server.xml` file and add this resource element inside of the `<GlobalNamingResources>` element:

```
<Resource
    name="jdbc/externalDataSource"
    auth="Container"
    type="javax.sql.DataSource"
    factory="org.apache.tomcat.jdbc.pool.DataSourceFactory"
    driverClassName="org.mariadb.jdbc.Driver"
    url="jdbc:mariadb://localhost/external"
    username="yourusername"
    password="yourpassword"
    maxActive="20"
    maxIdle="5"
    maxWait="10000"
/>
```

   Replace the specified username and password with the correct values for your database.

3. Edit your `[LIFERAY_HOME]/tomcat-8.0.32/conf/context.xml` file and add this resource link element inside of the `<Context>` element:

```
<ResourceLink name="jdbc/externalDataSource" global="jdbc/externalDataSource" type="javax.sql.DataSource"/>
```

   Now your data source is defined at Tomcat's scope.

4. Create a `com.liferay.blade.samples.jndiservicebuilder.service-log4j-ext.xml` in your Liferay DXP instance's `[LIFERAY_HOME]/osgi/log4` folder. Create this folder if it doesn't yet exist. Add this content to the XML file that you created:

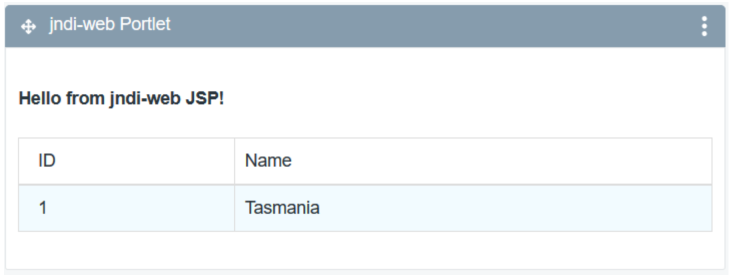
```
<?xml version="1.0"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <category name="com.liferay.blade.samples.jndiservicebuilder.service.impl">
        <priority value="INFO" />
    </category>
</log4j:configuration>
```

   This XML file defines the log level for the classes in the `com.liferay.blade.samples.jndiservicebuilder.service.impl` package. The `com.liferay.blade.samples.jndiservicebuilder.service.impl.RegionLocalServiceImpl` is the class that will produce log messages when the sample portlet is viewed.

Now your sample is ready for deployment! Make sure to build and deploy each of the three modules that comprise the sample application:

- `jndi-api`
- `jndi-service`
- `jndi-web`

After these modules have been deployed, add the `jndi-web` portlet to a Liferay DXP page.



Hello from jndi-web JSP!

ID	Name
1	Tasmania

Figure 137.4: This sample prints out the values previously inputted into the database.

A sample table is printed in the portlet's view, representing the info inputted into the database.

What API(s) and/or code components does this sample highlight?

This sample demonstrates two ways to access data from an external database defined by a JNDI connection:

- extract data directly from the raw data source by explicitly specifying a SQL query.
- read data using the helper methods that Service Builder generates in your application's persistence layer.

How does this sample leverage the API(s) and/or code component?

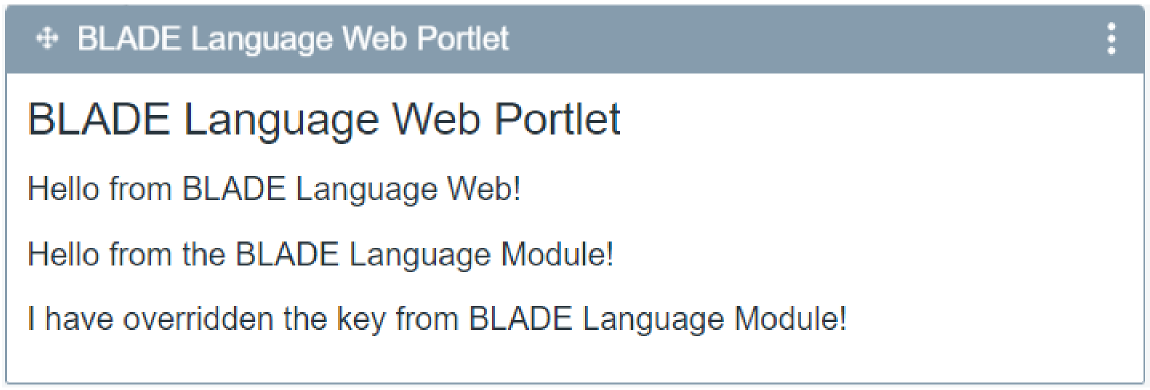
Once you've added the `jndi-web` portlet to a page, the `RegionLocalServiceUtil.useJNDI` method is invoked. This method accesses the database defined by the JNDI connection you specified and logs information about the rows in the `region` table to Liferay DXP's log.

The first way of accessing data from the external database is to extract data directly from the raw data source by explicitly specifying a SQL query. This technique is demonstrated by the `RegionLocalServiceImpl.useJNDI` method. That method obtains the Spring-defined data source that's injected into the `regionPersistence` bean, opens a new connection, and reads data from the data source. This is the technique used by the sample application to write the data to Liferay DXP's log.

The second way of accessing data from the external database is to read data using the helper methods that Service Builder generates in your application's persistence layer. This technique is demonstrated by the `UseJNDI.getRegions` method which first obtains an instance of the `RegionLocalService` OSGi service and then invokes `regionLocalService.getRegions`. The `regionLocalService.getRegions` and `regionLocalService.getRegionsCount` methods are two examples of the persistence layer helper methods that Service Builder generates. This is the technique used by the sample application to actually display the data. The portlet's `view.jsp` uses the `<search-container>` JSP tag to display a list of results. The results are obtained by the `UseJNDI.getRegions` method mentioned above.

137.4 Shared Language Keys

The Shared Language Keys sample provides a JSP portlet that displays language keys.


BLADE Language Web Portlet

BLADE Language Web Portlet

Hello from BLADE Language Web!

Hello from the BLADE Language Module!

I have overridden the key from BLADE Language Module!

Figure 137.5: The sample JSP portlet displays three language keys.

The language keys displayed in the portlet come from two different modules.

What API(s) and/or code components does this sample highlight?

This sample is broken into two modules:

- `language`
- `language-web`

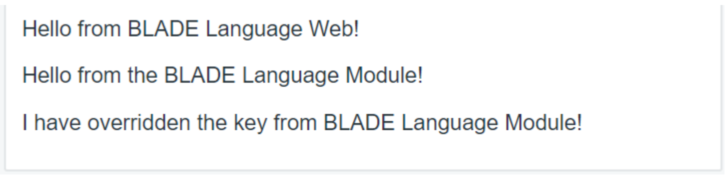
The `language-web` module provides a JSP portlet with unique language keys that it displays. The `language` module provides a resource module which only holds language keys. Its sole purpose is to share language keys with the JSP portlet provided in `language-web`. This sample conveys Liferay's recommended approach to sharing language keys through OSGi services.

How does this sample leverage the API(s) and/or code component?

You must deploy both `language-web` and `language` modules to simulate this sample's targeted demonstration.

First, note the language keys provided by each module:

- `language-web`
  - `blade_language_web_LanguageWebPortlet.caption=Hello from BLADE Language Web!`
  - `blade_language_web_override_LanguageWebPortlet.caption=I have overridden the key from BLADE Language Module!`
- `language`
  - `blade_language_LanguageWebPortlet.caption=Hello from the BLADE Language Module!`
  - `blade_language_web_override_LanguageWebPortlet.caption=Hello from the BLADE Language Module but you won't see me!`


Hello from BLADE Language Web!

Hello from the BLADE Language Module!

I have overridden the key from BLADE Language Module!

Figure 137.6: The Language Web portlet displays three phrases, two of which are shared from a different module.

When you place the sample BLADE Language Web portlet on a Liferay DXP page, you're presented with three language keys.

The first message is provided by the `language-web` module. The second message is from the `language` module. The third message is provided by both modules; as you can see, the `language-web`'s message is used, overriding the `language` module's identically named language key.

This sample shows what takes precedence when displaying language keys. The order for this example goes

1. `language-web` module language keys
2. `language` module language keys
3. Liferay DXP language keys

So how does sharing language keys work?

By default, the `ResourceBundleLoaderAnalyzerPlugin` expands modules with `/content/Language.properties` files to add provided capabilities:

- `bundle.symbolic.name`
- `resource.bundle.base.name`

Then the deployed `LanguageExtender` scans modules with those capabilities to automatically register an associated `ResourceBundleLoader`.

You can leverage this functionality to use keys from common language modules by republishing an aggregate `ResourceBundleLoader`. This can be done two ways:

1. Via Components

   You can get a reference to the registered service in your components as detailed in the Overriding a Module's Language Keys tutorial. The main disadvantage of this approach is that it forces you to provide a specific implementation of the `ResourceBundleLoader`, making it harder to modularize in the future.

2. Via Provide Capability

   The same `LanguageExtender` that registers the services supports an extended syntax that lets you register an aggregate of a collection of bundles:

   ```
   -liferay-aggregate-resource-bundles: \
       blade.language
   ```

   This approach has the advantage of easier extensibility. When language keys change, only the common language modules must be built and redeployed for the modules referencing them to recognize their updates.

For more information on sharing language keys, visit the Internationalization tutorials.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

137.5 Simulation Panel App

The Simulation Panel App provides new functionality in @product's Simulation Menu. When deploying this sample with no customizations, the *Simulation Sample* feature is provided in the Simulation Menu with four options.

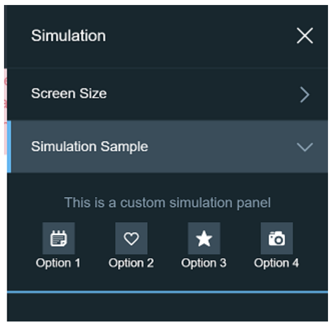


Figure 137.7: A simulation panel app adds new functionality to the Simulation Menu.

What API(s) and/or code components does this sample highlight?

This sample leverages the PanelApp API.

How does this sample leverage the API(s) and/or code component?

This sample leverages the `PanelApp` interface as an OSGi service via the `@Component` annotation:

```
@Component(
    immediate = true,
    property = {
        "panel.app.order:Integer=500",
        "panel.category.key=" + SimulationPanelCategory.SIMULATION
    },
    service = PanelApp.class
)
```

There are also two properties provided via the `@Component` annotation:

- `panel.app.order`: the order in which the panel app is displayed among other panel apps in the chosen category. Entries are ordered from top to bottom. For example, an entry with order 1 will be listed above an entry with order 2. If the order is not specified, it's chosen at random based on which service was registered first in the OSGi container.
- `panel.category.key`: the host panel category for your panel app, which should be the Simulation Menu category.

The simulation panel app extends the BaseJSPPanelApp, which provides a skeletal implementation of the PanelApp interface with JSP support. JSPs, however, are not the only way to provide frontend functionality to your panel categories/apps. You can create your own class implementing `PanelApp` to use other technologies, such as FreeMarker.

To learn more about Liferay Portal's product navigation using panel categories and panel apps, see the Customizing the Product Menu tutorial. For more information on extending the Simulation Menu, see the Extending the Simulation Menu tutorial.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

137.6 Spring MVC Portlet

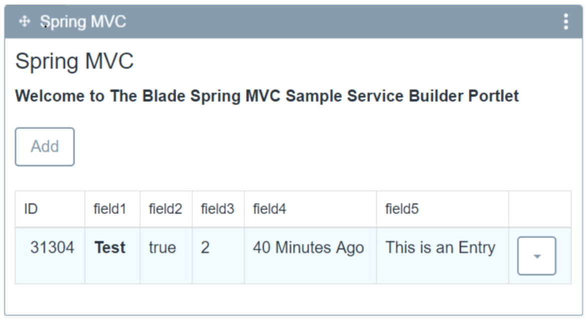
The Spring MVC portlet provides a way to add various different fields into the database and display them in a table. This project is a Spring MVC based portlet WAR that implements the same functionality as the `apps/service-builder/basic-web` sample project. It manages JSP pages for display, uses a Spring-annotated portlet class, and invokes the `apps/service-builder/basic-api` module to call services.

Note: If you're planning to package this sample using Maven, you must complete a few additional steps to avoid build errors. This sample relies on the `service-builder/basic-api` module. Since the `basic-api` bundle is not available on Liferay's CDN repo or Maven Central, this sample can not reference it, resulting in build failures.

To satisfy this dependency, you must install the bundle dependency to your local `~/.m2 repo`, along with the parent BND plugin and root Maven project. Here are the steps to accomplish this:

1. Run `mvn clean install` on `maven/apps/service-builder/basic-api`.
2. Run `mvn clean install` on `maven/parent.bnd.bundle.plugin`.
3. Run `mvn clean install -N` in the root `liferay-blade-samples/maven` folder.

Now you can build this sample successfully.



Spring MVC

Welcome to The Blade Spring MVC Sample Service Builder Portlet

ID	field1	field2	field3	field4	field5
31304	**Test**	true	2	40 Minutes Ago	This is an Entry

Figure 137.8: Click *Add* and fill out the sample fields to generate a custom entry in the portlet's table.

Unlike the `service-builder/basic-web` module, Spring MVC portlets must be delivered as portlet WAR projects. This project builds to a WAR file but leverages all of the Liferay Workspace tools and Gradle to build the WAR. You must build and deploy the `service-builder/basic-api` and `service-builder/basic-service` modules for this sample to work properly. For more information on using Spring MVC portlets in Liferay DXP, visit the Spring MVC tutorial.

What API(s) and/or code components does this sample highlight?

This sample demonstrates a Liferay DXP portlet built using the Spring Web MVC framework.

How does this sample leverage the API(s) and/or code component?

You can easily modify this sample by customizing its `SpringMVCPortletViewController` Java class or any of its JSPs stored in the `src/main/webapp/WEB-INF/jsp` folder. For more information on customizing this sample, see the Javadoc listed in this sample's `SpringMVCPortletViewController` Java class.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

Chapter 138

Extensions

This section focuses on Liferay sample extensions. You can view these sample extensions by visiting the `extensions` folder corresponding to your preferred build tool:

- Gradle sample extensions
- Liferay Workspace sample extensions
- Maven sample extensions

The following samples are documented:

- Control Menu Entry
- Document Action
- Gogo Shell Command
- Indexer Post Processor
- Model Listener
- Screen Name Validator

Visit a particular sample page to learn more!

138.1 Control Menu Entry

The Control Menu Entry sample provides a customizable button that is added to Liferay Portal's default Control Menu. When deploying this sample with no customizations, an additional button is added to the User (right side) portion of the Control Menu.

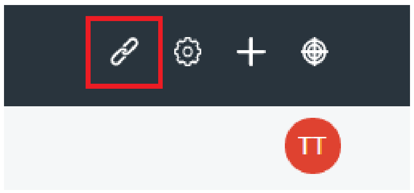


Figure 138.1: The User area of the Control Menu is provided an additional link button when the Control Menu Entry sample is deployed to Liferay DXP.

The button navigates the user to Liferay's website: https://www.liferay.com.

What API(s) and/or code components does this sample highlight?

This sample leverages the ProductNavigationControlMenuEntry API.

How does this sample leverage the API(s) and/or code component?

This sample first leverages the `ProductNavigationControlMenuEntry` interface as an OSGi service via the `@Component` annotation:

```
@Component(
    immediate = true,
    property = {
        "product.navigation.control.menu.category.key=" + ProductNavigationControlMenuCategoryKeys.USER,
        "product.navigation.control.menu.entry.order:Integer=1"
    },
    service = ProductNavigationControlMenuEntry.class
)
```

There are also two properties provided via the `@Component` annotation:

- `product.navigation.control.menu.category.key`: the category in which your entry should reside. The default Control Menu provides three categories: *SITES* (left portion), *TOOLS* (middle portion), and *USER* (right portion).
- `product.navigation.control.menu.entry.order:Integer`: the order in which your entry will be displayed in the category. Entries are ordered from left to right. For example, an entry with order 1 will be listed to the left of an entry with order 2. If the order is not specified, it's chosen at random based on which service was registered first in the OSGi container.

This sample also implements the `ProductNavigationControlMenuEntry` interface. The following methods are implemented:

- `getIcon(HttpServletRequest)`
- `getLabel(Locale)`
- `getURL(HttpServletRequest)`
- `isShow(HttpServletRequest)`

Refer to this sample's `BladeProductNavigationControlMenuEntry` class for Javadocs describing these methods. For more information on how to customize Liferay Portal's Control Menu, visit the Customizing the Control Menu tutorial.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

138.2 Document Action

The Document Action sample shows how to add a context menu option to an entry in the Documents and Media portlet. When deploying this sample with no customizations, an additional menu option is available in the Documents and Media Admin portlet and the Documents and Media portlet. This sample creates a *Blade Basic Info* option that displays basic information about the entry (e.g., file name, type, version, etc.). For example, the Admin portlet provides the new option as illustrated in the images below:

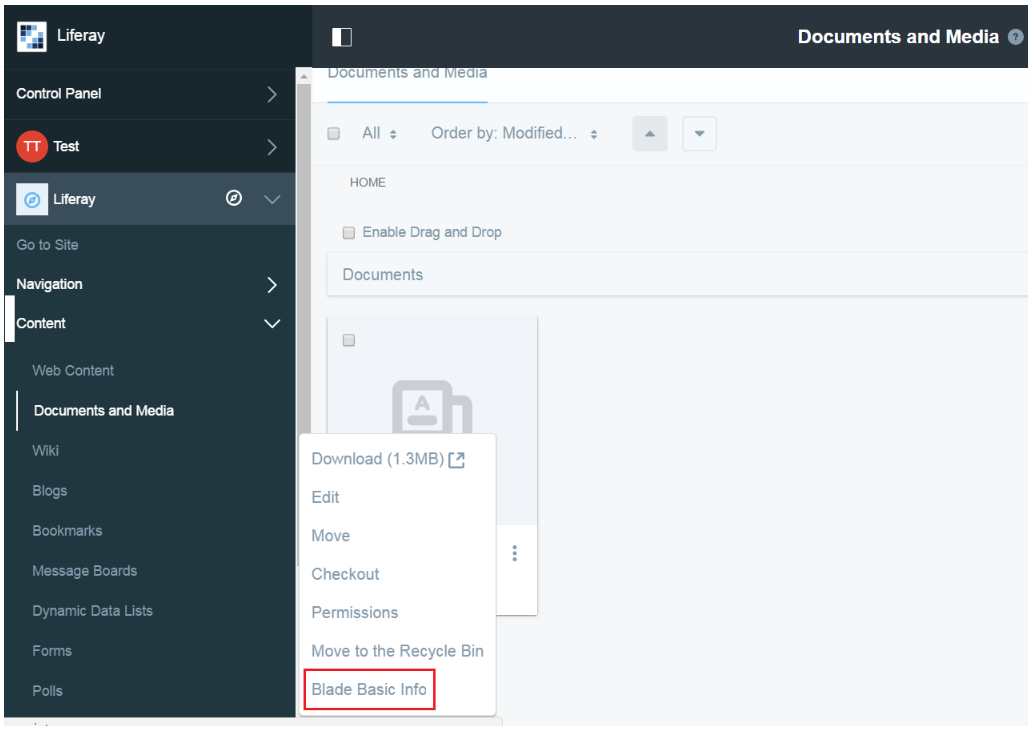


Figure 138.2: The new *Blade Basic Info* option is available from the entry's Options menu.

Likewise, the Documents and Media portlet provides the same option after selecting *Show Actions* from the portlet's Configuration menu.

What API(s) and/or code components does this sample highlight?

This sample leverages the PortletConfigurationIcon API.

How does this sample leverage the API(s) and/or code component?

There are four Java classes used in this sample:

- `BladeActionConfigurationIcon`: Adds the new context menu option to the Document Detail screen options (⋮) (top right corner) of the Documents and Media Admin portlet. See the Configuring Your Admin App's Actions Menu tutorial for more details.
- `BladeActionDisplayContext`: Adds the Display Context for the document action. More about Display Contexts are described later.
- `BladeActionDisplayContextFactory`: Adds the Display Context factory for the document action.

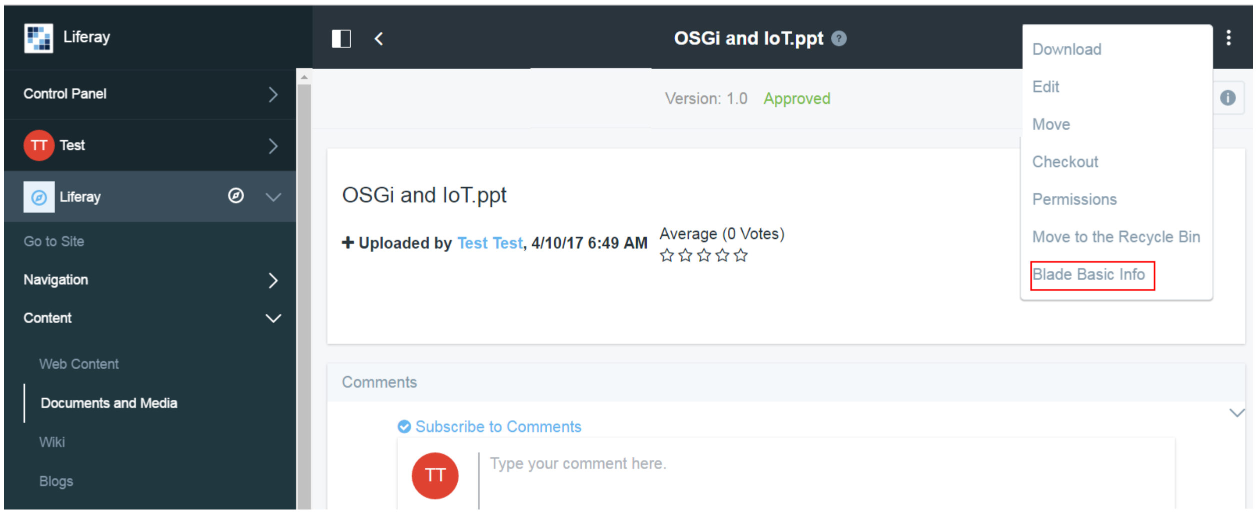


Figure 138.3: The new option is also available from the portlet's Document Details.

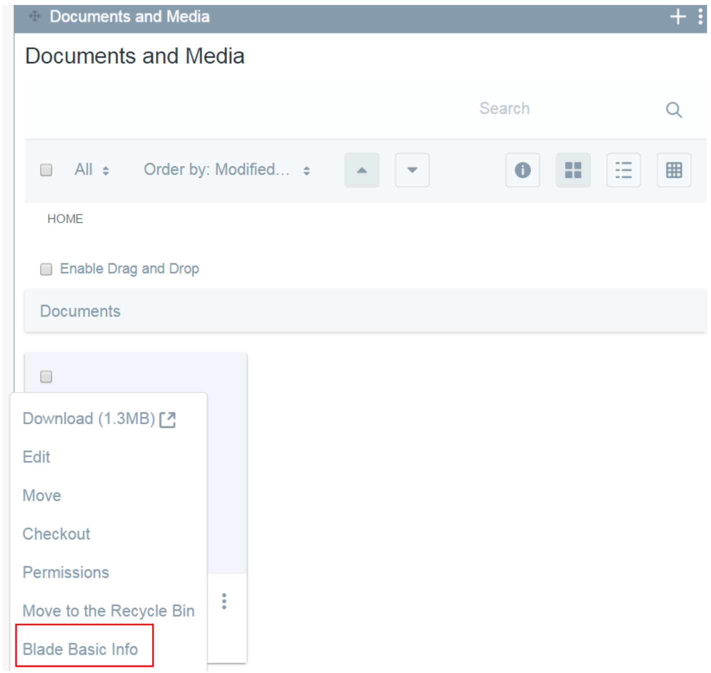


Figure 138.4: You can access the new *Blade Basic Info* option from the Documents and Media portlet added to a page.

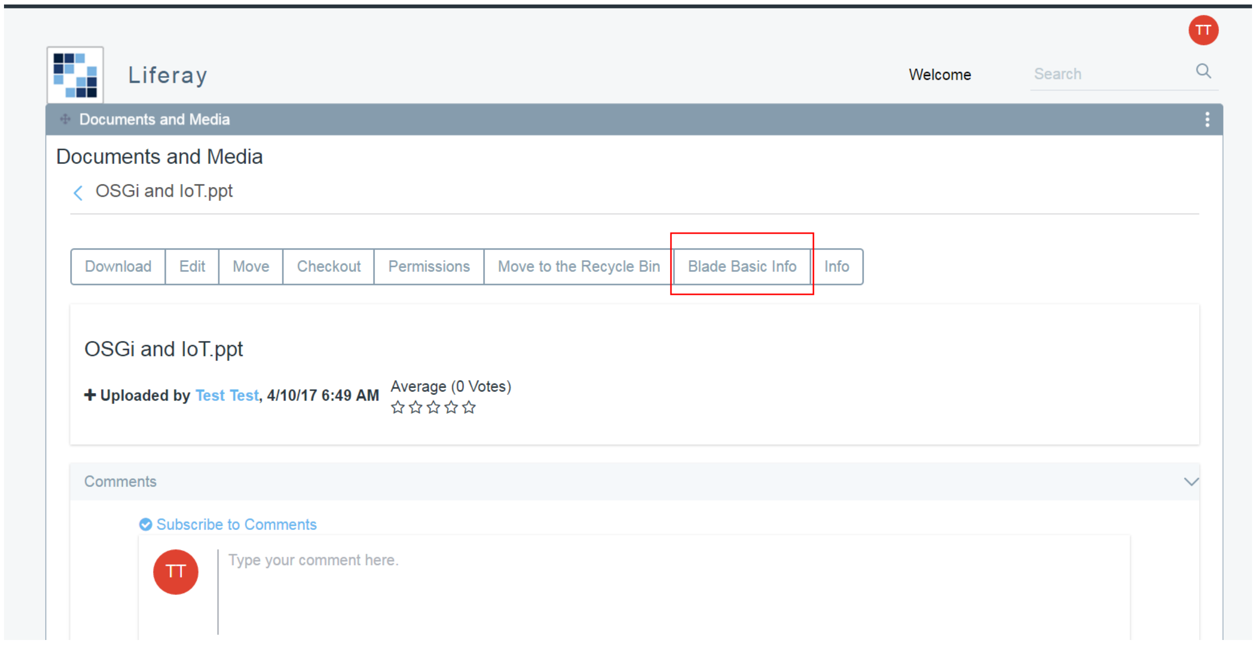


Figure 138.5: The Documents And Media portlet provides the option from its Document Detail too.

- `BladeDocumentActionPortlet`: Provides the portlet class, which extends the GenericPortlet. This class generates what is shown when the context menu option is selected.

A Display Context is a Java class that controls access to a portlet screen's UI elements. For example, the Document Library would use Display Contexts to provide its screens all their UI elements. It would use one Display Context for its document edit screen, another for its document view screen, etc. A portlet ideally uses a different Display Context for each of its screens.

A screen's JSP calls on the Display Context (DC) to get elements to render and to decide whether to render certain types of elements. Some of the DC methods return a collection of UI elements (e.g., a menu object of menu items), while other DC methods return booleans that determine whether to show particular element types. The DC decides which objects to display, while the JSP organizes the rendered objects and implements the screen's look and feel. You don't have to decide which elements to display in your JSP; simply call the DC methods to populate UI components with objects to render.

To customize or extend a portlet screen that uses a DC, you can extend the DC and override the methods that control access to the elements that interest you. For example, you can turn off displaying certain types of elements (e.g., actions) by overriding the DC method that makes that decision. You can add new custom elements (e.g., new actions) or remove existing elements (e.g., a delete action) from a collection of elements a DC method returns. The beauty of customizing via a DC is that you don't have to modify the JSP. You only modify the particular methods that are related to the UI customization goals. And JSP updates won't break the DC customizations. Replacing a JSP, on the other hand, can lead to missing an important JSP modification that a new Liferay version introduces.

As you create custom portlets, you may want to implement DCs. You can benefit from the separation of concerns that DCs provide and customers can extend your portlet DCs to specify which UI elements to display. And they don't need to worry about missing out on the updates you make to the JSPs.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

138.3 Gogo Shell Command

The Gogo Shell Command sample demonstrates adding a custom command to Liferay DXP's Gogo shell environment. All Liferay DXP installations have a Gogo shell environment, which lets system administrators interact with Liferay DXP's module framework on a local server machine.

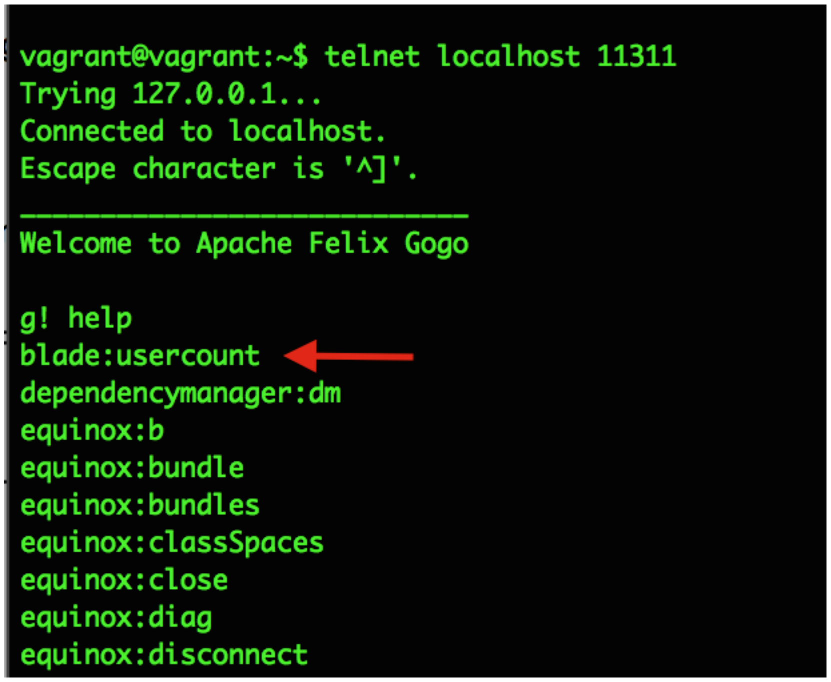
This example adds a new custom Gogo shell command called `usercount` under the `blade` scope. It prints out the number of registered users on your Liferay DXP installation.

To test this sample, follow the instructions below:

1. Start a Liferay DXP installation.

2. Using a command line tool, connect to your local Gogo shell. For example, you can do this by executing `telnet localhost 11311`.

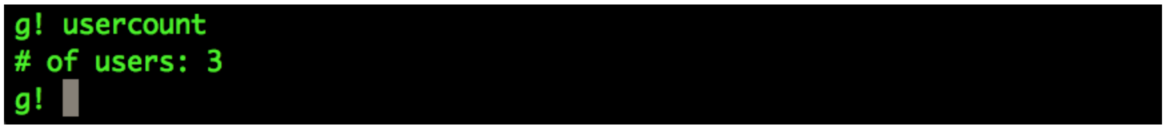
3. Run `help` to view all the available commands. The sample Gogo shell command is listed.



```
vagrant@vagrant:~$ telnet localhost 11311
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
____________________________
Welcome to Apache Felix Gogo

g! help
blade:usercount
dependencymanager:dm
equinox:b
equinox:bundle
equinox:bundles
equinox:classSpaces
equinox:close
equinox:diag
equinox:disconnect
```

Figure 138.6: The sample Gogo shell command is listed with all the available commands.

4. Execute `usercount` to execute the new custom command. The number of users on your running Liferay Portal installation is printed.



```
g! usercount
# of users: 3
g!
```

Figure 138.7: The outcome of executing the usercount command.

What API(s) and/or code components does this sample highlight?

This sample demonstrates creating a new Gogo shell command by leveraging `osgi.command.*` properties in a Java class.

How does this sample leverage the API(s) and/or code component?

To add this new Gogo shell command, you must implement the logic in a Java class with the following two properties:

- `osgi.command.function`: the command's name, which must match the method name in the registered service implementation.
- `osgi.command.scope`: the general scope or namespace for the command.

These properties are set in your class's `@Component` annotation like this:

```
@Component(
    property = {"osgi.command.function=usercount", "osgi.command.scope=blade"},
    service = Object.class
)
```

The logic for the usercount command is specified in the method with the same name:

```
public void usercount() {
    System.out.println(
        "# of users: " + getUserLocalService().getUsersCount());
}
```

This method uses *Declarative Services* to get a reference for the `UserLocalService` to invoke the `getUsersCount` method. This lets you find the number of users currently in the system.

For more information on using the Gogo shell, see the Using the Felix Gogo Shell tutorial.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

138.4 Indexer Post Processor

The Indexer Post Processor sample demonstrates using the `IndexerPostProcessor` interface, which is provided to customize search queries and documents before they're sent to the search engine, and/or customize result summaries when they're returned to end users. This basic demonstration prints a message in the log when one of the `*IndexerPostProcessor` methods is called.

To see this sample's messages in Liferay DXP's log, you must add a logging category to the portal. Navigate to *Control Panel* → *Configuration* → *Server Administration* and click on *Log Levels* → *Add Category*. Then fill out the form:

- *Logger Name*: `com.liferay.blade.samples.indexerpostprocessor`
- *Log Level*: `INFO`

Once you save the new logging category, you can witness the sample indexer post processor in action. For example, you can test the sample's `BlogsIndexerPostProcessor` implementation by creating a blog entry. When you publish the blog, the following message is logged in the console:

```
18:27:30,737 INFO  [http-nio-8080-exec-8][BlogsIndexerPostProcessor:76] postProcessDocument
```

What API(s) and/or code components does this sample highlight?

This sample leverages the IndexerPostProcessor API.

How does this sample leverage the API(s) and/or code component?

This sample contains four implementations of the `IndexerPostProcessor` interface:

- `BlogsIndexerPostProcessor`
- `MultipleEntityIndexerPostProcessor`
- `MultipleIndexerPostProcessor`
- `UserEntityIndexerPostProcessor`

All these classes leverage the interface as an OSGi service via the `@Component` annotation. For example, the `@Component` annotation of the `UserEntityIndexerPostProcessor` looks like this:

```
@Component(
    immediate = true,
    property = {
        "indexer.class.name=com.liferay.portal.kernel.model.User",
        "indexer.class.name=com.liferay.portal.kernel.model.UserGroup"
    },
    service = IndexerPostProcessor.class
)
```

There's one property type provided via the `@Component` annotation:

- `indexer.class.name`: the fully qualified class name of the indexed entity or an `Indexer` class itself.

This sample's implementations of the `IndexerPostProcessor` interface override the following methods:

- `postProcessContextBooleanFilter`
- `postProcessContextQuery`
- `postProcessDocument`

- `postProcessFullQuery`
- `postProcessSearchQuery(BooleanQuery, BooleanFilter)`
- `postProcessSearchQuery(BooleanQuery, SearchContext)`
- `postProcessSummary`

For more information on Liferay's Search API, refer to the Introduction to Liferay Search tutorial.

Where Is This Sample?

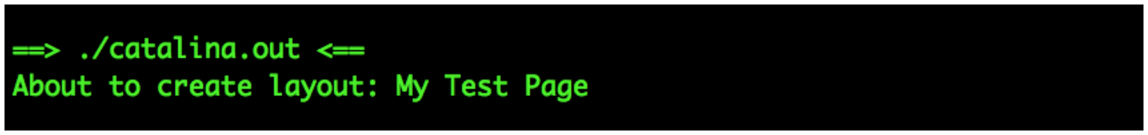
There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

138.5 Model Listener

The Model Listener sample demonstrates adding a custom model listener to a Liferay Portal out-of-the-box entity. When deploying this sample with no customizations, a custom model listener is added to the portal's layouts, listening for `onBeforeCreate` events. This means that any page creation will trigger this listener, which will execute before the new page is created.

For example, if a new page is added with the name *My Test Page*, the following message is printed to the console:



```
==> ./catalina.out <==
About to create layout: My Test Page
```

Figure 138.8: The sample model listener's message in the console.

You can also verify that the model listener sample was executed by navigating to the new page's *Configure Page* → *SEO* option. The HTML Title field looks like this:

What API(s) and/or code components does this sample highlight?

This sample leverages the ModelListener API.

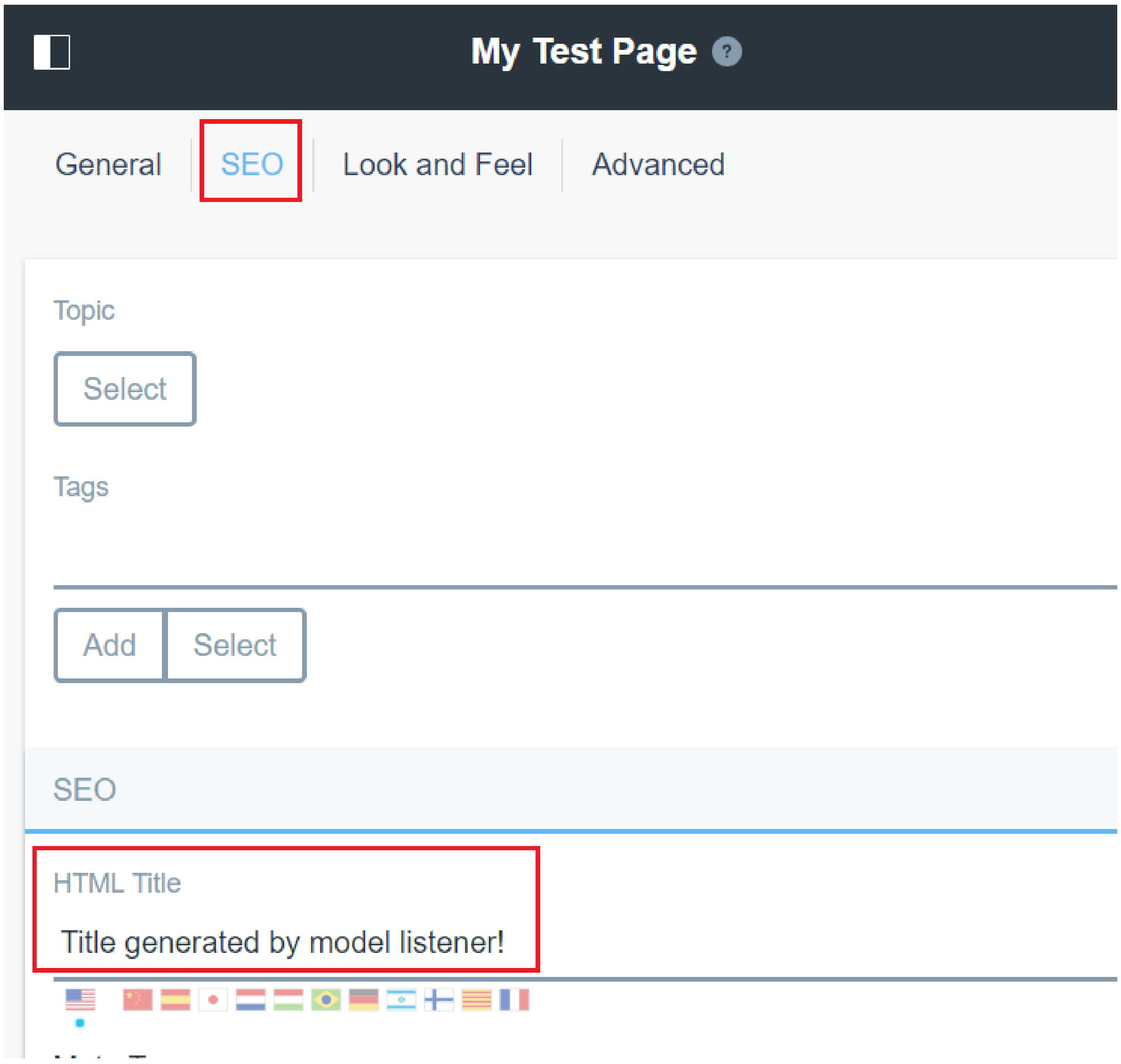
How does this sample leverage the API(s) and/or code component?

Model Listeners are used to listen for persistence events on models and take actions as a result of those events. Actions can be executed on an entity's database table before or after a `create`, `remove`, `update`, `addAssociation`, or `removeAssociation` event. It's possible to have more than one model listener on a single model too; the execution order is not guaranteed.

There are two steps to create a new model listener:

- Implement a Model Listener class
- Register the new service in Liferay's OSGi runtime

This sample adds the model listener logic in a new Java class named `CustomLayoutListener` that extends BaseModelListener.



HTML Title

Title generated by model listener!

Figure 138.9: The page's HTML title updated by the model listener sample.

```
public class CustomLayoutListener extends BaseModelListener<Layout> {

    @Override
    public void onBeforeCreate(Layout model) throws ModelListenerException {
        System.out.println(
            "About to create layout: " + model.getNameCurrentValue());

        model.setTitle("Title generated by model listener!");
    }

}
```

Important things to note in this code snippet are

- The entity to be targeted by this model listener is specified as the parameterized type (e.g., Layout).
- The overridden methods dictate the type of event(s) that are listened for (e.g., `onBeforeCreate`); they also trigger the logic execution.

The final step is registering the service in Liferay's OSGi runtime, which is accomplished by the following annotation (if using Declarative Services):

```
@Component(immediate = true, service = ModelListener.class)
```

For more information on model listeners, see the Creating Model Listeners tutorial.

Where Is This Sample?

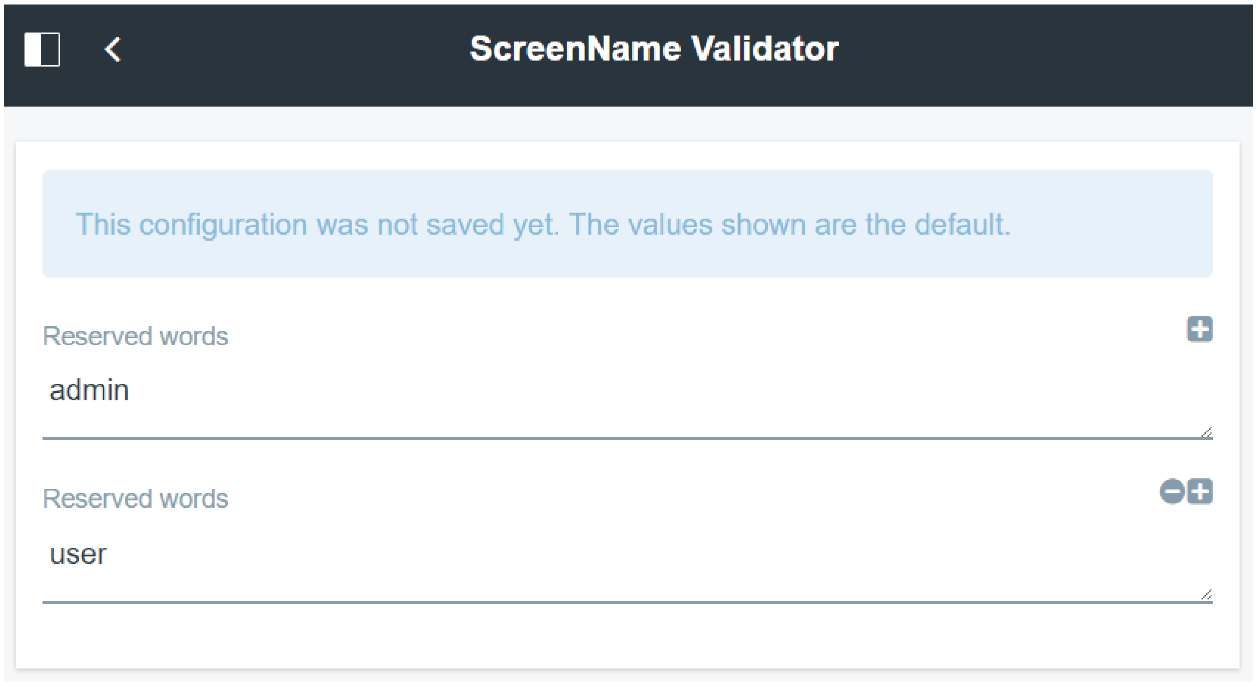
There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

138.6 Screen Name Validator

The Screen Name Validator sample provides a way to validate a user's inputted screen name. During validation, the screen name is tested client-side and server-side.

This sample checks if a user's screen name contains reserved words that are configured in the *Control Panel → Configuration → System Settings → Foundation → ScreenName Validator* menu. The default values for the screen name validator's reserved words are *admin* and *user*.


ScreenName Validator

This configuration was not saved yet. The values shown are the default.

Reserved words: admin

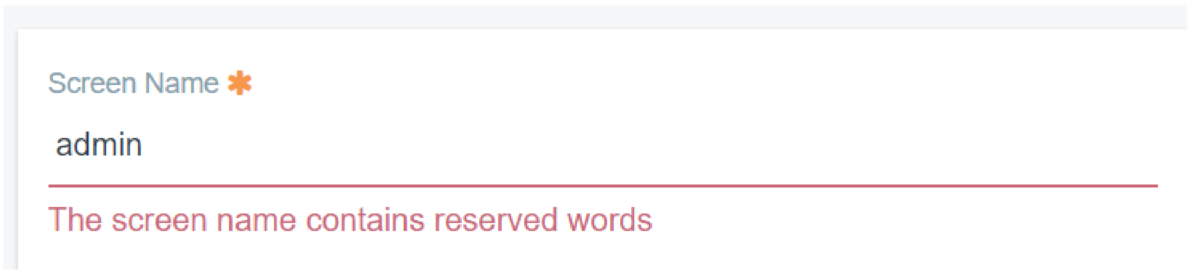
Reserved words: user

Figure 138.10: Enter reserved words for the screen name validator.

You can test this sample by following the following steps:

1. Deploy the Screen Name Validator to your portal installation.
2. Navigate to the *Control Panel → Users → Users and Organizations* menu.

3. Create a new user by selecting the *Add User* (+) button.
4. Adding a screen name that contains the word *admin* or *user*.



admin

The screen name contains reserved words

Figure 138.11: The error message displays when inputting a reserved word for the screen name.

What API(s) and/or code components does this sample highlight?

This sample leverages the ScreenNameValidator API.

How does this sample leverage the API(s) and/or code component?

To customize this sample, modify its `com.liferay.blade.samples.screenname.validator.internal.CustomScreenNameValidator` class.

You can also customize this sample's configuration by adding more properties in its `com.liferay.blade.samples.screenname` class.

For more information on customizing the Validation sample to fit your needs, see the Javadoc provided in this sample's Java classes.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

138.7 Servlet

The Servlet sample provides an OSGi Whiteboard Servlet in Liferay DXP. When deploying this sample and configuring the servlet, a *Hello World* message is displayed when accessing the servlet page URL. Log info is also outputted to your console.

To configure the servlet in Liferay DXP, complete the following steps:

1. Navigate to the *Control Panel* → *Configuration* → *Server Administration* → *Log Levels*.

2. Select *Add Category*.

3. Insert *com.liferay.blade.samples.servlet.BladeServlet* for the Logger Name and *INFO* for the Log Level.

4. Navigate to the http://localhost:8080/o/blade/servlet URL.

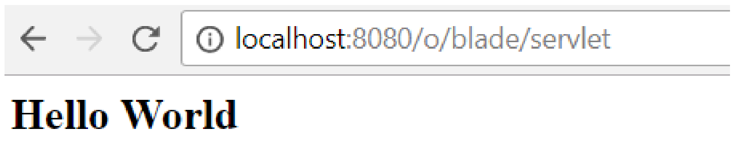
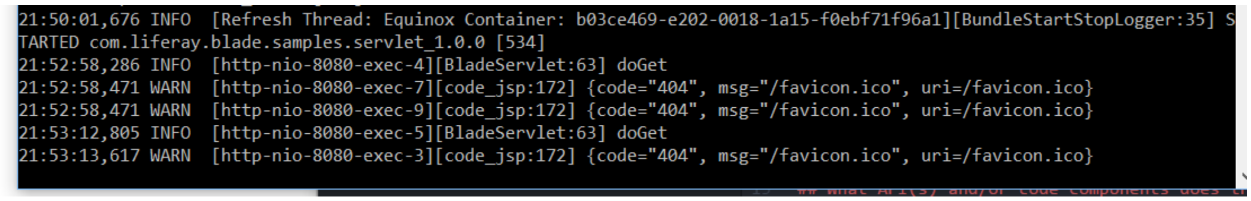


Figure 138.12: The servlet displays *Hello World* from the configured servlet page URL.



```
21:50:01,676 INFO  [Refresh Thread: Equinox Container: b03ce469-e202-0018-1a15-f0ebf71f96a1][BundleStartStopLogger:35] S
TARTED com.liferay.blade.samples.servlet_1.0.0 [534]
21:52:58,286 INFO  [http-nio-8080-exec-4][BladeServlet:63] doGet
21:52:58,471 WARN  [http-nio-8080-exec-7][code_jsp:172] {code="404", msg="/favicon.ico", uri=/favicon.ico}
21:52:58,471 WARN  [http-nio-8080-exec-9][code_jsp:172] {code="404", msg="/favicon.ico", uri=/favicon.ico}
21:53:12,805 INFO  [http-nio-8080-exec-5][BladeServlet:63] doGet
21:53:13,617 WARN  [http-nio-8080-exec-3][code_jsp:172] {code="404", msg="/favicon.ico", uri=/favicon.ico}
```

Figure 138.13: The servlet also logs info in the console.

What API(s) and/or code components does this sample highlight?

This sample leverages the HttpServlet API.

How does this sample leverage the API(s) and/or code component?

To customize this sample, modify its `com.liferay.blade.samples.servlet.BladeServlet` class. This class extends the `HttpServlet` class. Creating your own servlet for Liferay DXP is useful when you need to implement servlet actions. For example, if you wanted to implement the CMIS server by yourself with Apache Chemistry, you would need to implement your own servlet, managing requests at a low level.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

CHAPTER 139

OVERRIDES

This section focuses on Liferay sample overrides. You can view these sample overrides by visiting the `overrides` folder corresponding to your preferred build tool:

- Gradle sample overrides
- Liferay Workspace sample overrides
- Maven sample overrides

The following samples are documented:

- Core JSP Override
- Module JSP Override
- Resource Bundle Override

Visit a particular sample page to learn more!

139.1 Core JSP Override

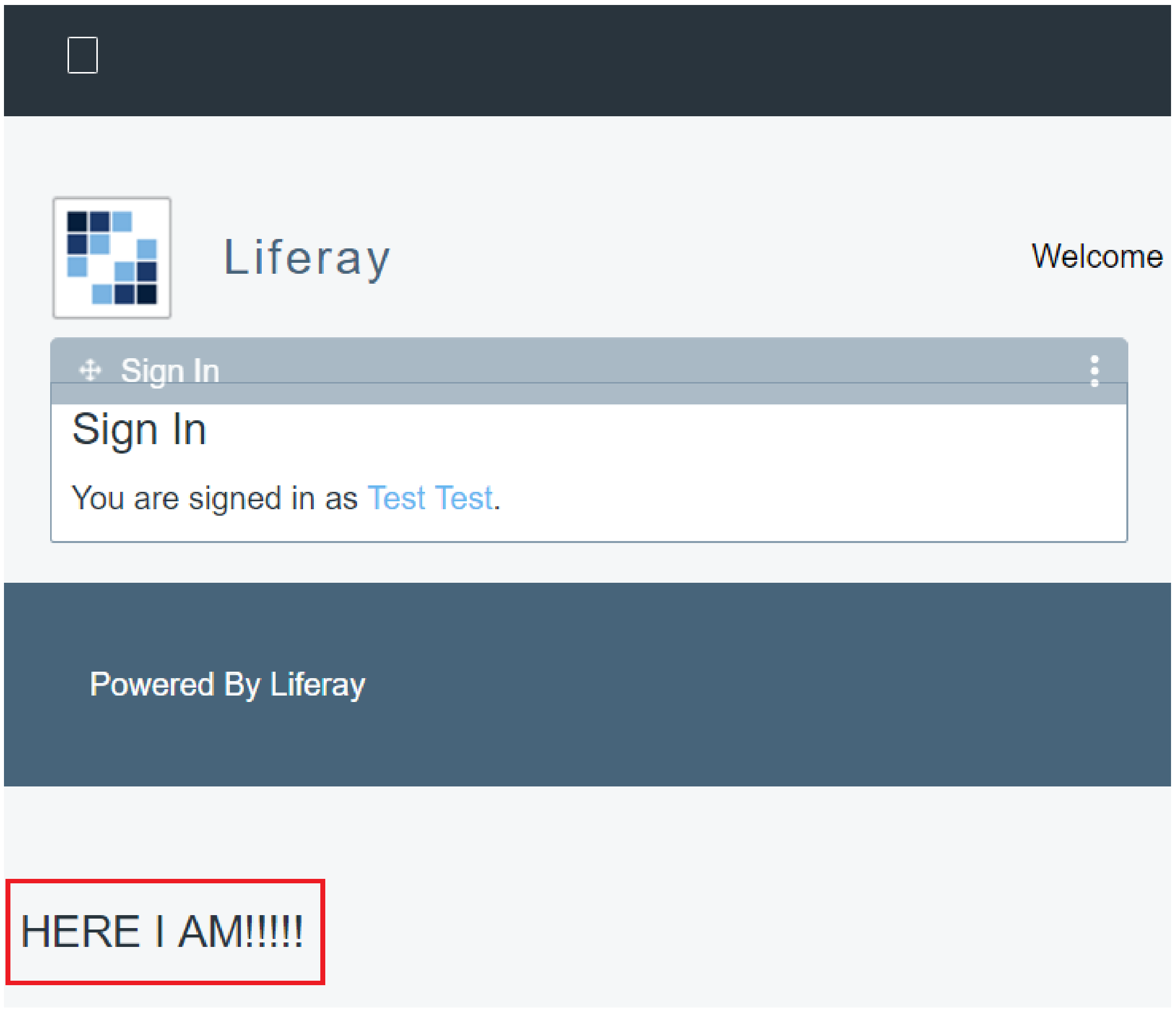
The Core JSP Override sample lets you override core/kernel JSPs by adding them to the module's `META-INF/jsps` folder. This module overrides the Liferay DXP's `bottom.jsp` file by inserting the `bottom-ext.jsp` file in the `META-INF/jsps/html/common/themes` folder. When deploying this sample with no customizations, sample text is added to the bottom of Liferay's default theme.

For more information on how to customize Liferay's Core using JSP overrides, visit the Overriding Core JSPs tutorial.

What API(s) and/or code components does this sample highlight?

This sample leverages the CustomJspBag API.

Important: Using core JSP overrides should be a last resort option only when there is no other way to customize functionality in your Liferay installation. It's up to the maintainer of this JSP override to properly maintain and adapt to changes in the underlying JSP implementation.



Sign In

You are signed in as Test Test.

HERE I AM!!!!!

Figure 139.1: Deploying a core JSP override overrides core functionality, like Liferay DXP's default theme.

How does this sample leverage the API(s) and/or code component?

You can easily modify this sample by customizing its `com.liferay.blade.samples.corejsphook.BladeCustomJspBag` Java class or adding additional JSPs in the configured JSP folder. You can modify the custom JSP folder's path by editing the `BladeCustomJspBag.getCustomJspDir()` method to return a different folder path.

For more information on customizing the Core JSP Override sample to fit your needs, see the Javadoc listed in this sample's `BladeCustomJspBag` class.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

139.2 Module JSP Override

The Module JSP Override sample conveys Liferay's recommended approach to override an application's JSP by leveraging OSGi fragment modules. This example overrides the default `login.jsp` file in the `com.liferay.login.web` bundle by adding the red text *changed* to the Sign In form.

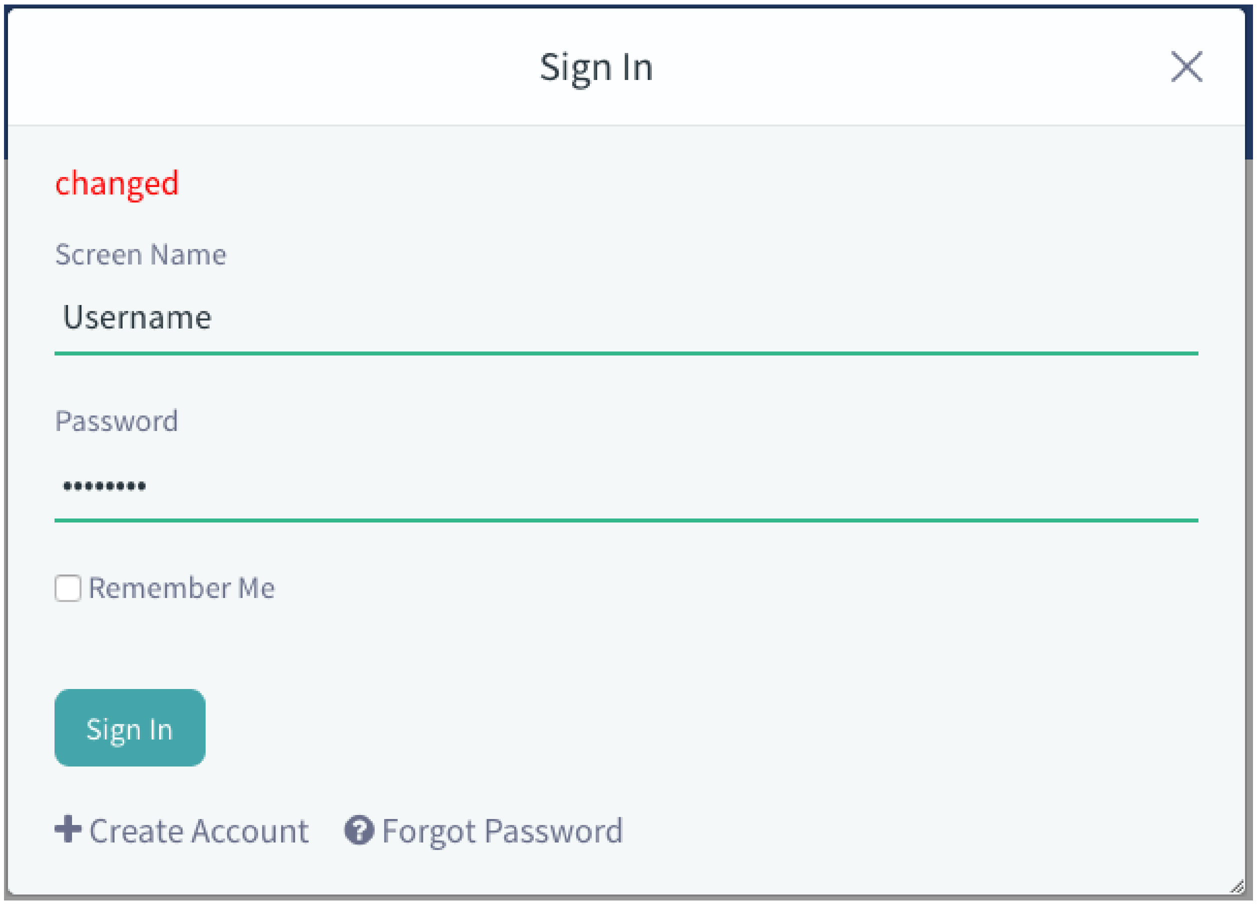


Figure 139.2: The customized Sign In form with the new *changed* text.

What API(s) and/or code components does this sample highlight?

This sample demonstrates how to create a fragment host module and configure it to override an existing module's JSP.

How does this sample leverage the API(s) and/or code component?

You can create your own JSP override by

- Declaring the fragment host.
- Providing the JSP that will override the original one.

To properly declare the fragment host in the `bnd.bnd` file, you must specify the host module's (where the original JSP is located) Bundle Symbolic Name and the host module's exact version to which the fragment belongs. In this example, this is configured like this:

```
Fragment-Host: com.liferay.login.web;bundle-version="1.0.0"
```

Then you must provide the new JSP intended to override the original one. Be sure to mimic the host module's folder structure when overriding its JAR. For this example, since the original JSP is in the folder `/META-INF/resources/login.jsp`, the new JSP file resides in the folder `src/main/resources/META-INF/resources/login.jsp`.

If needed, you can also target the original JSP following one of the two possible naming conventions: `original` or `portal`. This pattern looks like

```
<liferay-util:include
    page="/login.original.jsp"
    servletContext="<%= application %>"
/>
```

or

```
<liferay-util:include
    page="/login.portal.jsp"
    servletContext="<%= application %>"
/>
```

This approach can be used to override any application JSP (i.e., JSPs residing in a module). You can also add new JSPs to an existing module with this technique. If you need to override a core JSP, see the `core-jsp-override` sample.

For more information on using fragment bundles to override application JSPs, see the Overriding App JSPs tutorial.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

139.3 Resource Bundle Override

This example overrides the default `add-blog-entry` language key (English and Spanish) for Liferay DXP's default Blogs application. After deploying this sample hook to Liferay DXP, the Blogs application's *Add Blog Entry* button is modified to display *Overriden Add Blog Entry*. If you change Liferay DXP's default language to Spanish, the modified language key is translated to display in that language. For example, the text changes to *Añadir entrada sobreescrita*.

For reference, the Blogs application's language keys are stored in the liferay-portal Github repo's `modules/apps/collaboration/blogs/blogs-web/src/main/resources/content` folder.

What API(s) and/or code components does this sample highlight?

This sample leverages the ResourceBundleLoader API.

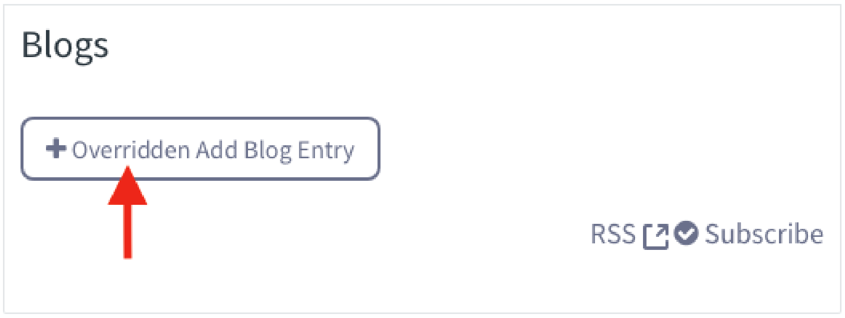


Figure 139.3: The customized Blogs application displays the new `add-blog-entry` language key in English.

How does this sample leverage the API(s) and/or code component?

This sample conveys the recommended approach to override an application's language keys file for any module that is deployed to Liferay DXP's OSGi runtime (not applicable to Liferay DXP's core language keys).

The steps to override applications' language keys are

- Implement a resource bundle loader.
- Register the service.
- Provide the new language keys that will override the original ones.

The resource bundle loader is a class that should implement the interface `com.liferay.portal.kernel.util.ResourceBundle`
Specifically, you must implement the `loadResourceBundle` method, which returns the loaded resource bundle:

```
@Override
public ResourceBundle loadResourceBundle(String languageId) {
    return _resourceBundleLoader.loadResourceBundle(languageId);
}
```

Then you must set the resource bundle loader to load the resource bundles as an `AggregateResourceBundleLoader`.

```
@Reference(target = "(&(bundle.symbolic.name=com.liferay.blogs.web)(!(component.name=com.liferay.blade.samples.hook.resourcebundle.ResourceBundleLoaderCompo
public void setResourceBundleLoader(
    ResourceBundleLoader resourceBundleLoader) {

    _resourceBundleLoader = new AggregateResourceBundleLoader(
        new CacheResourceBundleLoader(
            new ClassResourceBundleLoader(
                "content.Language",
                ResourceBundleLoaderComponent.class.getClassLoader())),
    resourceBundleLoader);
}
```

The `@Reference` annotation targets the original Blogs module by specifying its symbolic name `com.liferay.blogs.web`. This sample's own component name (i.e., `com.liferay.blade.samples.hook.resourcebundle.ResourceB`
is not targeted to use this resource bundle.

Also note the required parameters to set the resource bundle loader:

- The base language file name (e.g., `content.Language`).
- The classloader for your resource bundle loader.
- The resource bundle loader from the method's parameter.

The class should also register the resource bundle loader in the OSGi runtime. This is done by setting the following three properties:

- `bundle.symbolic.name`: The symbolic name of the target module (i.e., the module's keys you're overriding).
- `resource.bundle.base.name`: The resource bundle base name that points to your language files.
- `servlet.context.name`: The servlet context name of the target module.

These properties are set in your class's `@Component` annotation like this:

```
@Component(
    immediate = true,
    property = {
        "bundle.symbolic.name=com.liferay.blogs.web",
        "resource.bundle.base.name=content.Language",
        "servlet.context.name=blogs-web"
    }
)
```

Lastly, the new `language.properties` files should be added to the folder `src/content` for each locale's keys you want to override. Since this example's goal is to only override the English and Spanish keys, the `Language_en.properties` and `Language_es.properties` are added.

This approach can be used to override any application's language keys (i.e., `language.properties` files that are inside a module deployed to Liferay DXP's OSGi runtime). If you need to override Liferay DXP's core language keys, see the Modifying Global Language Keys tutorial.

For more information on using a resource bundle to override an application's language keys, see the Overriding a Module's Language Keys tutorial.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

Chapter 140

Themes

This section focuses on Liferay sample themes. You can view these sample themes by visiting the `themes` folder corresponding to your preferred build tool:

- Gradle sample themes
- Liferay Workspace sample themes
- Maven sample themes

The following samples are documented:

- Simple Theme
- Template Context Contributor
- Theme Contributor

Visit a particular sample page to learn more!

140.1 Simple Theme

The Simple Theme sample provides the base files for a theme, using the Theme Builder Gradle plugin. When deploying this sample with no customizations, a theme based off of the `_styled` base theme is created.

For more information on themes, visit the Introduction to Themes tutorial.

What API(s) and/or code components does this sample highlight?

This sample demonstrates a way to create a simple theme in Liferay DXP.

How does this sample leverage the API(s) and/or code component?

To modify this sample, add the `images`, `js`, or `templates` folder, along with your modified files, to the `src/main/webapp` folder. The sample already provides the `src/main/resources/resources-importer`, `src/main/webapp/WEB-INF`, and `src/main/webapp/css` folders for you. Add your style modifications to the provided `css/_custom.scss` file. For a complete explanation of a theme's files, see the Theme Reference Guide.

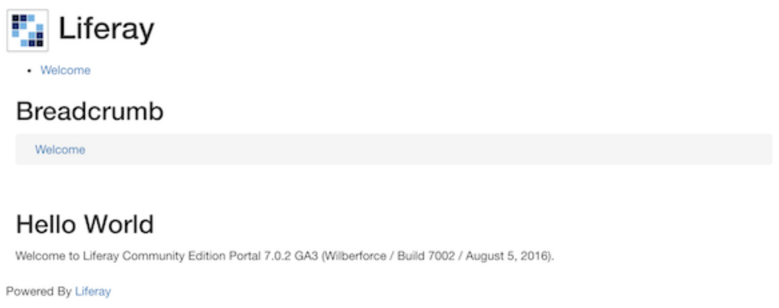

Liferay
Welcome
Breadcrumb
Welcome
Hello World
Welcome to Liferay Community Edition Portal 7.0.2 GA3 (Wilberforce / Build 7002 / August 5, 2016).
Powered By Liferay

Figure 140.1: A theme based off of the Styled base theme is created when the Theme Blade sample is deployed to Liferay Portal.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

140.2 Template Context Contributor

The Template Context Contributor sample injects a new variable into Liferay DXP's theme context. When deploying this sample with no customizations, you can use the `${sample_text}` variable from any theme.

What API(s) and/or code components does this sample highlight?

Many developers prefer using templating frameworks like FreeMarker and Velocity, but don't have access to the common objects offered to those working with JSPs. Context contributors allow non-JSP developers an easy way to inject variables into their Liferay templates.

This sample leverages the TemplateContextContributor API.

How does this sample leverage the API(s) and/or code component?

You can easily modify this sample by customizing its `BladeTemplateContextContributor.java` Java class. For example, the default context contributor sample provides the `${sample_text}` variable by injecting it into Liferay's `contextObjects`, which is a map provided by default to offer common variables to non-JSP template developers. You can easily inject your own variables into the `contextObjects` map usable by any theme deployed to Liferay DXP.

Are you working with templates that aren't themes (e.g., ADTs, DDM templates, etc.)? You can change the context in which your variables are injected by modifying the `property` attribute in the `@Component` annotation. If you want your variable available for all templates, change it to

```
property = {"type=" + TemplateContextContributor.TYPE_GLOBAL}
```

For more information on customizing the Template Context Contributor sample to fit your needs, see the Javadoc listed in this sample's `com.liferay.blade.samples.theme.contributorBladeTemplateContextContributor` class. For more information on context contributors and how to create them in Liferay DXP, visit the Context Contributors tutorial.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

140.3 Theme Contributor

The Theme Contributor sample contributes updates to the UI of the theme body, Control Menu, Product Menu, and Simulation Panel. When deploying this sample with no customizations, the colors of the theme and aforementioned menus are updated.

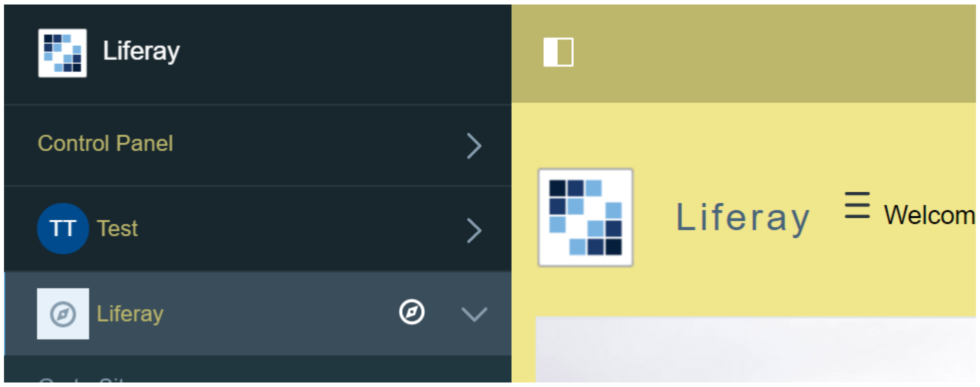
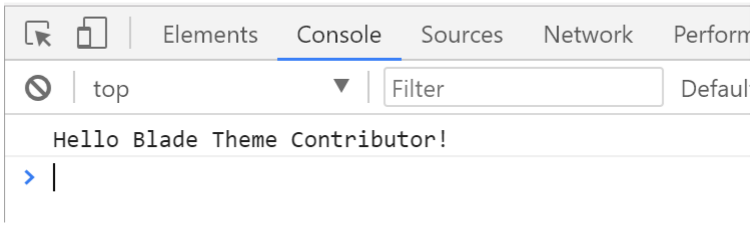


Figure 140.2: Your Liferay DXP pages and menu fonts now have a yellow tint.

Also, there's a simple JavaScript update that is provided, which logs a message to the browser's console window that states *Hello Blade Theme Contributor!*.



```
Hello Blade Theme Contributor!
```

Figure 140.3: The message is printed to your browser's console window using JavaScript.

What API(s) and/or code components does this sample highlight?

This sample demonstrates a way to contribute updates to a Liferay DXP theme. Theme Contributors let you package UI resources (e.g., CSS and JS) independent of a theme to include on a Liferay DXP page.

How does this sample leverage the API(s) and/or code component?

To modify this sample, replace the corresponding JS or SCSS file with the JavaScript or styles that you want, or add your own JS or SCSS files. For example, this sample provides an update to the Control Menu's `background-color` in its `src/main/resources/META-INF/resources/css/blade.theme.contributor/_control_menu.scss` file:

```
body {
	.control-menu {
		background-color: darkkhaki;
	}
}
```

All of the SCSS files used in this sample are imported into the main `blade.theme.contributor.scss` file:

```
@import "bourbon";
@import "mixins";

@import "blade.theme.contributor/body";
@import "blade.theme.contributor/control_menu";
@import "blade.theme.contributor/product_menu";
@import "blade.theme.contributor/simulation_panel";
```

If you add your own SCSS files, you must add them to the list of imports in the `blade.theme.contributor.scss` file.

Likewise, the sample `blade.theme.contributor.js` logs a message to your browser's console window using the following JS logic:

```
console.log('Hello Blade Theme Contributor!');
```

For more information on Theme Contributors, visit the Theme Contributors tutorial.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

140.4 Third Party Packages Portal Exports

The `com.liferay.portal.bootstrap` module exports many third party Java packages that can cause problems if used improperly. If your WAR's Gradle file, for example, uses the `compile` scope for a dependency that Liferay's OSGi runtime already provides, the dependency JAR is included in the WAR's `WEB-INF/lib` and deployed in the resulting WAB, and two versions of dependency classes wind up on the classpath. This can cause weird errors that are hard to debug.

To find a list of the packages exported by `com.liferay.portal.bootstrap`, go to the source file `modules/core/portal-bootstrap/system.packages.extra.bnd`. If you don't have access to the source code, the same list (in a less user-friendly format) is in the `META-INF/system.packages.extra.mf` file in `[LIFERAY_HOME]/osgi/core/com.liferay.portal.bootstrap.jar`. These packages are installed and available in Liferay's OSGi runtime. If your module or WAR uses one of them, specify the corresponding dependency as being "provided" (provided by Liferay DXP). Here's how to specify a provided dependency:

Maven: `<scope>provided</scope>`

Gradle: `providedCompile`

Now you can safely leverage third party packages Liferay DXP provides!

Related topics

Resolving a Plugin's Dependencies
Configuring Dependencies

140.5 Resolving Common Output Errors Reported by the resolve Task

Liferay Workspace provides the `resolve` Gradle task to validate modules. This is very useful for finding issues and reporting them as output before deployment. For more information on running this task from Liferay Workspace, see the Validating Modules Against the Target Platform tutorial section. For general help with OSGi related issues, visit the Troubleshooting FAQ tutorial section.

For help interpreting the `resolve` task's output, see the list below for common output errors, what they mean, and how to fix them.

Missing Import Error

When your module refers to an unavailable import, the container throws this error. For example, suppose you have a module `test-service` that depends on the `com.google.common.base` package. If the container can't find that package, it throws this error:

```
Resolution exception in project 'modules:test-service': Unresolved requirements in root project 'modules:test-service':
    Mandatory:
        [osgi.wiring.package ] com.google.common.base; version=[23.0.0,24.0.0)
        [osgi.identity       ] test.service
```

This kind of error can also occur when separate modules require different versions of another module. If you have *module A* requiring *module Test version 1* and *module B* requiring *module Test version 4*, without running the resolver, both modules A and B would compile successfully. When they were deployed, however, one would fail in the OSGi runtime because both dependencies cannot be satisfied. These types of scenarios are difficult to diagnose, but with the `resolve` task, can be found with ease.

To fix missing import errors, you may need to adjust the export and/or import configuration of your modules. Also, see the Resolving Third Party Library Package Dependencies tutorial for more information on resolving import errors. Sometimes, this kind of error can be solved by editing the `resolve` task's list of capabilities. See the Depending on Third Party Libraries Not Included in Liferay DXP section to learn how to do this.

Missing Service Reference

If your module references a non-existent service, an error is thrown. This is helpful because service reference issues are hard to diagnose during deployment without using the Gogo Shell.

For example, if your module `test-portlet` references a service (e.g., `test.api.TestApi`) it does not have access to, the following error is thrown:

```
Resolution exception in project 'modules:test-portlet': Unresolved requirements in project 'modules:test-portlet':
    Mandatory:
        [osgi.identity ] test.portlet
        [osgi.service  ] objectClass=test.api.TestApi
```

To fix this, you must make the service available to your module. If you're expecting the service to be provided by your target platform, check to make sure it's being provided. If it's a service provided by a custom module, check that service provider module and ensure it's correctly providing that service to your module. To check the target platform for available services, follow the steps below:

1. Start your target platform instance.

2. Start the Gogo shell from a local telnet session (e.g., `telnet localhost 11311`).

3. List all services containing a keyword by running `services | grep "SERVICE_NAME"`. It's easiest to do this rather than listing all services since there are usually too many to sift through.

4. You can also list services provided by a component. Run `lb -s` to list all provided bundles by their bundle symbolic name (BSN). Find the BSN for the desired component and then run `scr:info <BSN>`.

If you're unable to track down your missing service, it may be provided by a customized Liferay DXP core feature or an external Liferay DXP feature. If this is the case, it isn't included in the target platform's default capabilities. You can make the custom service capability available to reference by generating a new custom distro JAR.

Missing Fragment Host

Referring to a non-existent fragment host throws an error. For example, if your `test.login` fragment is configured to modify a fragment host named `com.liferay.login.web` that cannot be referenced, the following error is thrown:

```
Resolution exception in project 'modules:test.login': Unresolved requirements in project 'modules:test-login':
    Mandatory:
        [osgi.identity    ] test.login
        [osgi.wiring.host ] com.liferay.login.web; version=1.0.10
```

Configuring a fragment host in your module is typically done with the `Fragment-Host` header in the `bnd.bnd` file:

```
Fragment-Host: com.liferay.login.web;bundle-version="[1.0.0,1.0.1)"
```

To fix this, inspect your target platform to ensure it includes the JAR you're attempting to add a fragment for. Your fragment host header may be referencing an incorrect bundle symbolic name (BSN) or version. The easiest way to check this is by using the Gogo Shell. Follow the steps below to find the bundle symbolic name:

1. Start your target platform instance.

2. Start the Gogo shell from a local telnet session (e.g., `telnet localhost 11311`).

3. List all installed bundles by BSN with the command `lb -s`. You can search through the output to find the BSN. If you already know the BSN and want to check the version, run `lb -s | grep "<BSN>"`.

Once you know the correct BSN/version to reference, update your `Fragment-Host` header to resolve the error.

For more information on fragments, see the JSP Overrides Using OSGi Fragments tutorial.

140.6 CKEditor Plugin Reference Guide

This reference guide provides a list of the default CKEditor plugins bundled with Liferay DXP's AlloyEditor. Each plugin below links to its `plugin.js` file for reference:

- about
- allyhelp
- allyhelpbtn
- ajaxsave
- autocomplete
- basicstyles
- bbcode
- bidi
- blockquote
- clipboard
- colorbutton
- colordialog
- contextmenu
- creole
- dialogadvtab
- div
- elementspath
- enterkey
- entities
- filebrowse
- find
- flash
- floatingspace
- font
- format
- forms
- horizontalrule
- htmlwriter
- image
- iframe
- indent
- itemselector
- justify
- link
- list
- liststyle
- lfrpopup
- magicline
- media
- newpage
- pagebreak
- pastefromword
- pastetext

- preview
- removeformat
- resize
- restore
- selectall
- showblocks
- showborders
- smiley
- sourcearea
- specialchar
- stylescombo
- tab
- table
- tabletools
- templates
- toolbar
- undo
- wikilink
- wysiwygarea

Note: The following CKEditor plugins are not available for inline mode in AlloyEditor at this time; however, you can still use them in the classic CKEditor:

- maximize
- print
- save

To use the Classic CKEditor instead of AlloyEditor, there are a few properties you can use, depending on the portlet. Add the properties that you need to your `portal-ext.properties` file:

```
editor.wysiwyg.default=ckeditor
editor.wysiwyg.portal-impl.portlet.ddm.text_html.ftl=ckeditor
editor.wysiwyg.portal-web.docroot.html.portlet.announcements.edit_entry.jsp=ckeditor
editor.wysiwyg.portal-web.docroot.html.portlet.blogs.edit_entry.jsp=ckeditor
editor.wysiwyg.portal-web.docroot.html.portlet.mail.edit.jsp=ckeditor
editor.wysiwyg.portal-web.docroot.html.portlet.mail.edit_message.jsp=ckeditor
editor.wysiwyg.portal-web.docroot.html.portlet.message_boards.edit_message.html.jsp=ckeditor
editor.wysiwyg.portal-web.docroot.html.taglib.ui.discussion.jsp=ckeditor
editor.wysiwyg.portal-web.docroot.html.taglib.ui.email_notification_settings.jsp=ckeditor
```

140.7 Item Selector Criterion and Return Types

Liferay DXP bundles have apps and app suites containing `ItemSelectorCriterion` classes and `ItemSelectorReturnType` classes developers can use.

Item Selector Criterion Classes

Collaboration App Suite Modules:

- `com.liferay.item.selector.criteria.api`:

    - ImageItemSelectorCriterion: Image file entity type.
    - AudioItemSelectorCriterion: Audio file entity type.
    - FileItemSelectorCriterion: Document Library file entity type.
    - UploadItemSelectorCriterion: Uploadable file entity type.
    - URLItemSelectorCriterion: URL entity type.
    - VideoItemSelectorCriterion: Video file entity type.

- `com.liferay.wiki.api` has wiki criterion.

Web Experience App Suite Modules:

- `com.liferay.site.item.selector.api` has site criterion.
- `com.liferay.layout.item.selector.api` has layout criterion.
- `com.liferay.journal.item.selector.api` has web content criterion.

If there's no criterion class for your entity, you can create your own `ItemSelectorCriterion` class (tutorial coming soon).

Item Selector Return Type Classes

The Liferay Collaboration app suite's `com.liferay.item.selector.criteria.api` module includes the following return types:

- Base64ItemSelectorReturnType: Base64 encoding of the entity as a `String`.
- FileEntryItemSelectorReturnType: File entry information as a JSON object.
- URLItemSelectorReturnType: URL of the entity as a `String`.
- UUIDItemSelectorReturnType: Universally Unique Identifier (UUID) of the entity as a `String`.

If there's no return type class that meets your needs, you can implement your own `ItemSelectorReturnType` class (tutorial coming soon).

140.8 Breaking Changes

This document presents a chronological list of changes that break existing functionality, APIs, or contracts with third party Liferay developers or users. We try our best to minimize these disruptions, but sometimes they are unavoidable.

The breaking changes covered in this article apply to both the commercial and open source versions of Liferay.

Here are some of the types of changes documented in this file:

- Functionality that is removed or replaced
- API incompatibilities: Changes to public Java or JavaScript APIs

- Changes to context variables available to templates
- Changes in CSS classes available to Liferay themes and portlets
- Configuration changes: Changes in configuration files, like `portal.properties`, `system.properties`, etc.
- Execution requirements: Java version, J2EE Version, browser versions, etc.
- Deprecations or end of support: For example, warning that a certain feature or API will be dropped in an upcoming version.
- Recommendations: For example, recommending using a newly introduced API that replaces an old API, in spite of the old API being kept in Liferay Portal for backwards compatibility.

Breaking Changes List

The liferay-ui:logo-selector Tag Requires Parameter Changes

- **Date:** 2013-Dec-05
- **JIRA Ticket:** LPS-42645

What changed? The Logo Selector tag now supports uploading an image, storing it as a temporary file, cropping it, and canceling edits. The tag no longer requires creating a UI to include the image. Consequently, the `editLogoURL` parameter is no longer needed and has been removed. The tag now uses the following parameters to support the new features:

- `currentLogoURL`: the URL to display the image being stored
- `maxFileSize`: the size limit for the logo to be uploaded
- `tempImageFileName`: the unique identifier to store the temporary image on upload

Who is affected? Plugins or templates that are using the `liferay-ui:logo-selector` tag need to update their usage of the tag.

How should I update my code? You should remove the parameter `editLogoURL` and include (if neccessary) the parameters `currentLogoURL`, `maxFileSize`, and/or `tempImageFileName`.

Example

Old way:

```
<portlet:renderURL var="editUserPortraitURL" windowState="<%= LiferayWindowState.POP_UP.toString() %>">
    <portlet:param name="struts_action" value="/users_admin/edit_user_portrait" />
    <portlet:param name="redirect" value="<%= currentURL %>" />
    <portlet:param name="p_u_i_d" value="<%= String.valueOf(selUser.getUserId()) %>" />
    <portlet:param name="portrait_id" value="<%= String.valueOf(selUser.getPortraitId()) %>" />
</portlet:renderURL>

<liferay-ui:logo-selector
    defaultLogoURL="<%= UserConstants.getPortraitURL(themeDisplay.getPathImage(), selUser.isMale(), 0) %>"
    editLogoURL="<%= editUserPortraitURL %>"
    imageId="<%= selUser.getPortraitId() %>"
    logoDisplaySelector=".user-logo"
/>
```

New way:

```
<liferay-ui:logo-selector
    currentLogoURL="<%= selUser.getPortraitURL(themeDisplay) %>"
    defaultLogoURL="<%= UserConstants.getPortraitURL(themeDisplay.getPathImage(), selUser.isMale(), 0) %>"
    imageId="<%= selUser.getPortraitId() %>"
    logoDisplaySelector=".user-logo"
    maxFileSize="<%= PrefsPropsUtil.getLong(PropsKeys.USERS_IMAGE_MAX_SIZE) / 1024 %>"
    tempImageFileName="<%= String.valueOf(selUser.getUserId()) %>"
/>
```

Why was this change made? This change helps keep a unified UI and consistent experience for uploading logos in the portal. The logos can be customized from a single location and used throughout the portal. In addition, the change adds new features such as image cropping and support for canceling image upload.

Merged Configured Email Signature Field into the Body of Email Messages from Message Boards and Wiki

- **Date:** 2014-Feb-28
- **JIRA Ticket:** LPS-44599

What changed? The configuration for email signatures of notifications from Message Boards and Wiki has been removed. An automatic update process is available that appends existing signatures into respective email message bodies for Message Boards and Wiki notifications. The upgrade process only applies to configured signatures in the database. In case you declared signatures in portal properties (e.g., `portal-ext.properties`), you must make the manual changes explained below.

Who is affected? Users and system administrators who have configured email signatures for Message Boards or Wiki notifications are affected. System administrators who have configured portal properties (e.g., `portal-ext.properties`) must make the manual changes described below.

How should I update my code? You should modify your `portal-ext.properties` file to remove the properties `message.boards.email.message.added.signature`, `message.boards.email.message.updated.signature`, `wiki.email.page.added.signature`, and `wiki.email.page.updated.signature`. Then, you should append the contents of the signatures to the bodies you had previously configured in your `portal-ext.properties` file.

Example

Old way:

```
wiki.email.page.updated.body=A wiki page was updated.
wiki.email.page.updated.signature=For any doubts email the system administrator
```

New way:

```
wiki.email.page.updated.body=A wiki page was updated.\n--\nFor any doubts email the system administrator
```

Why was this change made? This change helps simplify the user interface. The signatures can still be set inside the message body. There was no real benefit in keeping the signature and body fields separate.

Removed get and format Methods that Used PortletConfig Parameters

- **Date:** 2014-Mar-07
- **JIRA Ticket:** LPS-44342

What changed? All the methods `get()` and `format()` which had the PortletConfig as a parameter have been removed.

Who is affected? Any invocations from Java classes or JSPs to these methods in `LanguageUtil` and `UnicodeLanguageUtil` are affected.

How should I update my code? Replace invocations to these methods with invocations to methods of the same name that take a `ResourceBundle` parameter, instead of taking a `PortletConfig` parameter.

Example

Old call:

```
LanguageUtil.get(portletConfig, locale, key);
```

New call:

```
LanguageUtil.get(portletConfig.getResourceBundle(locale), key);
```

Why was this change made? The removed methods didn't work properly and would never work properly, since they didn't have all the information they required. Since we expected the methods were rarely used, we thought it better to remove them without deprecation than to leave them as buggy methods in the API.

Web Content Articles Now Require a Structure and Template

- **Date:** 2014-Mar-18
- **JIRA Ticket:** LPS-45107

What changed? Web content is now required to use a structure and template. A default structure and template named *Basic Web Content* was added to the global scope, and can be modified or deleted.

Who is affected? Applications that use the Journal API to create web content without a structure or template are affected.

How should I update my code? You should always use a structure and template when creating web content. You can still use the *Basic Web Content* from the global scope (using the structure key `basic-web-content`), but you should keep in mind that users can modify or delete it.

Why was this change made? This change gives users the flexibility to modify the default structure and template.

Changed the AssetRenderer and Indexer APIs to Include the PortletRequest and PortletResponse Parameters

- **Date:** 2014-May-07
- **JIRA Ticket:** LPS-44639 and LPS-44894

What changed? The `getSummary()` method in the AssetRenderer API and the `doGetSummary()` method in the Indexer API have changed and must include a `PortletRequest` and `PortletResponse` parameter as part of their signatures.

Who is affected? These methods must be updated in all AssetRenderer and Indexer implementations.

How should I update my code? Add a `PortletRequest` and `PortletResponse` parameter to the signatures of these methods.

Example 1

Old signature:

```
protected Summary doGetSummary(Document document, Locale locale, String snippet, PortletURL portletURL)
```

New signature:

```
protected Summary doGetSummary(Document document, Locale locale, String snippet, PortletRequest portletRequest, PortletResponse portletResponse)
```

Example 2

Old signature:

```
public String getSummary(Locale locale)
```

New signature:

```
public String getSummary(PortletRequest portletRequest, PortletResponse portletResponse)
```

Why was this change made? Some content (such as web content) needs the `PortletRequest` and `PortletResponse` parameters in order to be rendered.

Only One Portlet Instance's Settings is Used Per Portlet

- **Date:** 2014-Jun-06
- **JIRA Ticket:** LPS-43134

What changed? Previously, some portlets allowed separate setups per portlet instance, regardless of whether the instances were in the same page or in different pages. For some of the portlet setup fields, however, it didn't make sense to allow different values in different instances. The flexibility of these fields was unnecessary and confused users. As part of this change, these fields have been moved from portlet instance setup to Site Administration.

The upgrade process takes care of making the necessary database changes. In the case of several portlet instances having different configurations, however, only one configuration is preserved.

For example, if you configured three Bookmarks portlets where the mail configuration was the same, upgrade will be the same and you won't have any problem. But if you configured the three portlet instances differently, only one configuration will be chosen. To find out which configuration is chosen, you can check the log generated in the console by the upgrade process.

Since configuring instances of the same portlet type differently is highly discouraged and notoriously problematic, we expect this change will inconvenience only a very low minority of portal users.

Who is affected? Affected users are those who have specified varying configurations for multiple portlet instances of a portlet type, that stores configurations at the layout level.

How should I update my code? The upgrade process chooses one portlet instance's configurations and stores it at the service level. After the upgrade, you should review the portlet's configuration and make any necessary modifications.

Why was this change made? Unifying portlet and service configuration facilitates managing them.

DDM Structure Local Service API No Longer Has the updateXSDFieldMetadata operation

- **Date:** 2014-Jun-11
- **JIRA Ticket:** LPS-47559

What changed? The `updateXSDFieldMetadata()` operation was removed from the DDM Structure Local Service API.

DDM Structure Local API users should reference a structure's internal representation; any call to modify a DDM structure's content should be done through the DDMForm model.

Who is affected? Applications that use the DDM Structure Local Service API might be affected.

How should I update my code? You should always use DDMForm to update the DDM Structure content. You can retrieve it by calling `ddmStructure.getDDMForm()`. Perform any changes to it and then call `DDMStructureLocalServiceUtil.updateDDMStructure(ddmStructure)`.

Why was this change made? This change gives users the flexibility to modify the structure content without concerning themselves with the DDM Structure's internal content representation of data.

The aui:input Tag for Type checkbox No Longer Creates a Hidden Input

- **Date:** 2014-Jun-16
- **JIRA Ticket:** LPS-44228

What changed? Whenever the aui:input tag is used to generate an input of type checkbox, only an input tag will be generated, instead of the checkbox and hidden field it was generating before.

Who is affected? Anyone trying to grab the previously generated fields is affected. The change mostly affects JavaScript code trying to add some additional actions when clicking on the checkboxes.

How should I update my code? In your front-end JavaScript code, follow these steps:

- Remove the `Checkbox` suffix when querying for the node in any of its forms, like `A.one(...)`, `$(...)`, etc.
- Remove any action that tries to set the value of the checkbox on the previously generated hidden field.

Why was this change made? This change makes generated forms more standard and interoperable since it falls back to the checkboxes default behavior. It allows the form to be submitted properly even when JavaScript is disabled.

Using util-taglib No Longer Binds You to Using portal-kernel's javax.servlet.jsp Implementation

- **Date:** 2014-Jun-19
- **JIRA Ticket:** LPS-47682

What changed? Several APIs in `portal-kernel.jar` contained references to the `javax.servlet.jsp` package. This forced `util-taglib`, which depended on many of the package's features, to be bound to the same JSP implementation.

Due to this, the following APIs had breaking changes:

- `LanguageUtil`
- `UnicodeLanguageUtil`
- `VelocityTaglibImpl`
- `ThemeUtil`
- `RuntimePageUtil`
- `PortletDisplayTemplateUtil`
- `DDMXSDUtil`
- `PortletResourceBundles`
- `ResourceActionsUtil`
- `PortalUtil`

Who is affected? This affects anyone calling the classes listed above.

How should I update my code? Code invoking the APIs listed above should be updated to use an `HttpServletRequest` parameter instead of the formerly used `PageContext` parameter.

Why was this change made? As stated previously, the use of the `javax.servlet.jsp` API in `portal-kernel` prevented the use of any other JSP impl within plugins (OSGi or otherwise). This limited what Liferay could change with respect to providing its own JSP implementation within OSGi.

Changes in Exceptions Thrown by User Services

- **Date:** 2014-Jul-03
- **JIRA Ticket:** LPS-47130

What changed? In order to provide more information about the root cause of an exception, several exceptions have been extended with static inner classes, one for each cause. As a result of this effort, some exceptions have been identified that really belong as static inner subclasses of existing exceptions.

Who is affected? Client code which is handling any of the following exceptions:

- `DuplicateUserScreenNameException`
- `DuplicateUserEmailAddressException`

How should I update my code? Replace the old exception with the equivalent inner class exception as follows:

- `DuplicateUserScreenNameException` → `UserScreenNameException.MustNotBeDuplicate`
- `DuplicateUserEmailAddressException` → `UserEmailAddressException.MustNotBeDuplicate`

Why was this change made? This change provides more information to clients of the services API about the root cause of an error. It provides a more helpful error message to the end-user and it allows for easier recovery, when possible.

Removed Trash Logic from DLAppHelperLocalService Methods

- **Date:** 2014-Jul-22
- **JIRA Ticket:** LPS-47508

What changed? The `deleteFileEntry()` and `deleteFolder()` methods in `DLAppHelperLocalService` deleted the corresponding trash entry in the database. This logic has been removed from these methods.

Who is affected? Every caller of the `deleteFileEntry()` and `deleteFolder()` methods is affected.

How should I update my code? There is no direct replacement. Trash operations are now accessible through the `TrashCapability` implementations for each repository. The following code demonstrates using a `TrashCapability` instance to delete a `FileEntry`:

```
Repository repository = getRepository();

TrashCapability trashCapability = repository.getCapability(
    TrashCapability.class);

FileEntry fileEntry = repository.getFileEntry(fileEntryId);

trashCapability.deleteFileEntry(fileEntry);
```

Note that the `deleteFileEntry()` and `deleteFolder()` methods in `TrashCapability` not only remove the trash entry, but also remove the folder or file entry itself, and any associated data, such as assets, previews, etc.

Why was this change made? This change was made to allow different kinds of repositories to support trash operations in a uniform way.

Removed Sync Logic from DLAppHelperLocalService Methods

- **Date:** 2014-Sep-05
- **JIRA Ticket:** LPS-48895

What changed? The `moveFileEntry()` and `moveFolder()` methods in `DLAppHelperLocalService` fired Liferay Sync events. These methods have been removed.

Who is affected? Every caller of the `moveFileEntry()` and `moveFolder()` methods is affected.

How should I update my code? There is no direct replacement. Sync operations are now accessible through the `SyncCapability` implementations for each repository. The following code demonstrates using a `SyncCapability` instance to move a `FileEntry`:

```
Repository repository = getRepository();

SyncCapability syncCapability = repository.getCapability(
    SyncCapability.class);

FileEntry fileEntry = repository.getFileEntry(fileEntryId);

syncCapability.moveFileEntry(fileEntry);
```

Why was this change made? There are repositories that don't support Liferay Sync operations.

Removed the .aui Namespace from Bootstrap

- **Date:** 2014-Sep-26
- **JIRA Ticket:** LPS-50348

What changed? The `.aui` namespace was removed from prefixing all of Bootstrap's CSS.

Who is affected? Theme and plugin developers that targeted their CSS to rely on the namespace are affected.

How should I update my code? Theme developers can still manually add an `aui.css` file in their `_diffs` directory, and add it back in. The `aui` CSS class can also be added to the `$root_css_class` variable.

Why was this change made? Due to changes in the Sass parser, the nesting of third-party libraries was causing some syntax errors which broke other functionality (e.g., RTL conversion). There was also a lot of additional complexity for a relatively minor benefit.

Moved MVCPortlet, ActionCommand and ActionCommandCache from util-bridges.jar to portal-kernel.jar

- **Date:** 2014-Sep-26
- **JIRA Ticket:** LPS-50156

What changed? The classes from package `com.liferay.util.bridges.mvc` in `util-bridges.jar` were moved to a new package `com.liferay.portal.kernel.portlet.bridges.mvc` in `portal-kernel.jar`.

Old classes:

```
com.liferay.util.bridges.mvc.ActionCommand
com.liferay.util.bridges.mvc.BaseActionCommand
```

New classes:

```
com.liferay.portal.kernel.portlet.bridges.mvc.BaseMVCActionCommand
com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand
```

In addition, `com.liferay.util.bridges.mvc.MVCPortlet` is deprecated, but was made to extend `com.liferay.portal.kernel.portlet.bridges.mvc.MVCPortlet`.

The classes in the `com.liferay.portal.kernel.portlet.bridges.mvc` package have been renamed to add the `MVC` prefix. These modifications were made after this breaking change, and can be referenced in LPS-56372.

Who is affected? This will affect any implementations of `ActionCommand`.

How should I update my code? Replace imports of `com.liferay.util.bridges.mvc.ActionCommand` with `com.liferay.portal.kernel.portlet.bridges.mvc.MVCActionCommand` and imports of `com.liferay.util.bridges.mvc.BaseActio` with `com.liferay.portal.kernel.portlet.bridges.mvc.BaseMVCActionCommand`.

Why was this change made? This change was made to avoid duplication of an implementable interface in the system. Duplication can cause `ClassCastExceptions`.

Convert Process Classes Are No Longer Specified via the convert.processes Portal Property, but Are Contributed as OSGi Modules

- **Date:** 2014-Oct-09
- **JIRA Ticket:** LPS-50604

What changed? The implementation class `com.liferay.portal.convert.ConvertProcess` was renamed `com.liferay.portal.convert.BaseConvertProcess`. An interface named `com.liferay.portal.convert.ConvertProcess` was created for it.

The `convert.processes` key was removed from `portal.properties`. Consequentially, `ConvertProcess` implementations must register as OSGi components.

Who is affected? This affects any implementations of the former `ConvertProcess` class, including `ConvertProcess` class implementations in EXT plugins. Until version 6.2, this type of service could only be implemented with an EXT plugin, given that the `ConvertProcess` class resided in `portal-impl`.

How should I update my code? You should replace `extends com.liferay.portal.convert.ConvertProcess` with `extends com.liferay.portal.convert.BaseConvertProcess` and annotate the class with `@Component(service=ConvertProces`

Then turn your EXT plugin into an OSGi bundle and deploy it to the portal. You should see your convert process in the configuration UI.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Migration of the Field Type from the Journal Article API into a Vocabulary

- **Date:** 2014-Oct-13
- **JIRA Ticket:** LPS-50764

What changed? The field *type* from the Journal Article entity has been removed. The Journal API no longer supports this parameter. A new vocabulary called *Web Content Types* is created when migrating from previous versions of Liferay, and the types from the existing articles are kept as categories of this vocabulary.

Who is affected? This affects any caller of the removed methods `JournalArticle.getType()` and `JournalFeed.getType()`, and callers of `ArticleTypeException`'s methods, that attempt to use the former `type` parameter of the `JournalArticle` or `JournalFeed` service.

How should I update my code? If your logic was not affected by the type, you can simply remove the `type` parameter from the Journal API call. If your logic was affected by the type, you should now use the `AssetCategoryService` to obtain the category of the journal articles.

Why was this change made? Web Content Types had to be updated in a properties file and could not be translated easily. Categories provide a much more flexible behavior and a better UI. In addition, all the features, such as filters, developed for categories can be used now in asset publishers and faceted search.

Removed the getClassNamePortletId(String) Method from PortalUtil Class

- **Date:** 2014-Nov-11
- **JIRA Ticket:** LPS-50604

What changed? The `getClassNamePortletId(String)` method from the `PortalUtil` class has been removed.

Who is affected? This affects any plugin using the method.

How should I update my code? If you are using the method, you should implement it yourself in a private utility class.

Why was this change made? This change was needed in order to modularize the portal. Also, the method is no longer being used inside Liferay Portal.

Removed the Header Web Content and Footer Web Content Preferences from the RSS Portlet

- **Date:** 2014-Nov-12
- **JIRA Ticket:** LPS-46984

What changed? The *Header Web Content* and *Footer Web Content* preferences from the RSS portlet have been removed. The portlet now supports Application Display Templates (ADT), which provide templating capabilities that can apply web content to the portlet's header and footer.

Who is affected? This affects RSS portlets that are displayed on pages and that use these preferences. These preferences are no longer used in the RSS portlet.

How should I update my code? Even though these preferences have been removed, an ADT can be created to produce the same result. Liferay will publish this ADT so that it can be used in the RSS portlet.

Why was this change made? The support for ADTs in the RSS portlet not only covers this use case, but also covers many other use cases, providing a much simpler way to create custom preferences.

Removed the createFlyouts Method from liferay/util.js and Related Resources

- **Date:** 2014-Dec-18
- **JIRA Ticket:** LPS-52275

What changed? The `Liferay.Util.createFlyouts` method has been completely removed from core files.

Who is affected? This only affects third party developers who are explicitly calling `Liferay.Util.createFlyouts` for the creation of flyout menus. It will not affect any menus in core files.

How should I update my code? If you are using the method, you can achieve the same behavior with CSS.

Why was this change made? This method was removed due to there being no working use cases in Portal, and its overall lack of functionality.

Removed Support for Flat Thread View in Discussion Comments

- **Date:** 2014-Dec-30
- **JIRA Ticket:** LPS-51876

What changed? Discussion comments are now displayed using the *Combination* thread view, and the number of levels displayed in the tree is limited.

Who is affected? This affects installations that specify portal property setting `discussion.thread.view=flat`, which was the default setting.

How should I update my code? There is no need to update anything since the portal property has been removed and the `combination` thread view is now hard-coded.

Why was this change made? Flat view comments were originally implemented as an option to tree view comments, which were having performance issues with comment pagination.

Portal now uses a new pagination implementation that performs well. It allows comments to display in a hierarchical view, making it easier to see reply history. Therefore, the `flat` thread view is no longer needed.

Removed Asset Tag Properties

- **Date:** 2015-Jan-13
- **JIRA Ticket:** LPS-52588

What changed? The *Asset Tag Properties* have been removed. The service no longer exists and the Asset Tag Service API no longer has this parameter. The behavior associated with tag properties in the Asset Publisher and XSL portlets has also been removed.

Who is affected? This affects any plugin that uses the Asset Tag Properties service.

How should I update my code? If you are using this functionality, you can achieve the same behavior with *Asset Category Properties*. If you are using the Asset Tag Service, remove the `String[]` tag properties parameter from your calls to the service's methods.

Why was this change made? The Asset Tag Properties were deprecated for the 6.2 version of Liferay Portal.

Removed the asset.publisher.asset.entry.query.processors Property

- **Date:** 2015-Jan-22
- **JIRA Ticket:** LPS-52966

What changed? The `asset.publisher.asset.entry.query.processors` property has been removed from `portal.properties`.

Who is affected? This affects any hook that uses the `asset.publisher.asset.entry.query.processors` property.

How should I update my code? If you are using this property to register Asset Entry Query Processors, your Asset Entry Query Processor must implement the `com.liferay.portlet.assetpublisher.util.AssetEntryQueryProcessor` interface and must specify the `@Component(service=AssetEntryQueryProcessor.class)` annotation.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal.

Replaced the ReservedUserScreenNameException with UserScreenNameException.MustNotBeReserved in UserLocalService

- **Date:** 2015-Jan-29
- **JIRA Ticket:** LPS-53113

What changed? Previous to Liferay 7, several methods of `UserLocalService` could throw a `ReservedUserScreenNameException` when a user set a screen name that was not allowed. That exception has been deprecated and replaced with `UserScreenNameException.MustNotBeReserved`.

Who is affected? This affects developers who have written code that catches the `ReservedUserScreenNameException` while calling the affected methods.

How should I update my code? You should replace catching exception `ReservedUserScreenNameException` with catching exception `UserScreenNameException.MustNotBeReserved`.

Why was this change made? A new pattern has been defined for exceptions that provides higher expressivity in their names and also more information regarding why the exception was thrown.

The new exception `UserScreenNameException.MustNotBeReserved` has all the necessary information about why the exception was thrown and its context. In particular, it contains the user ID, the problematic screen name, and the list of reserved screen names.

Replaced the ReservedUserEmailAddressException with UserEmailAddressException Inner Classes in User Services

- **Date:** 2015-Feb-03
- **JIRA Ticket:** LPS-53279

What changed? Previous to Liferay 7, several methods of `UserLocalService` and `UserService` could throw a `ReservedUserEmailAddressException` when a user set an email address that was not allowed. That exception has been deprecated and replaced with `UserEmailAddressException.MustNotUseCompanyMx`, `UserEmailAddressException.MustNotBePOP3User`, and `UserEmailAddressException.MustNotBeReserved`.

Who is affected? This affects developers who have written code that catches the `ReservedUserEmailAddressException` while calling the affected methods.

How should I update my code? Depending on the method you're calling and the context in which you're calling it, you should replace catching exception `ReservedUserEmailAddressException` with catching exception `UserEmailAddressException.MustNotUseCompanyMx`, `UserEmailAddressException.MustNotBePOP3User`, or `UserEmailAddressException.MustNotBeReserved`.

Why was this change made? A new pattern has been defined for exceptions. This pattern requires using higher expressivity in exception names and requires that each exception provide more information regarding why it was thrown.

Each new exception provides its context and has all the necessary information about why the exception was thrown. For example, the `UserEmailAddressException.MustNotBeReserved` exception contains the problematic email address and the list of reserved email addresses.

Replaced ReservedUserIdException with UserIdException Inner Classes

- **Date:** 2015-Feb-10
- **JIRA Ticket:** LPS-53487

What changed? The `ReservedUserIdException` has been deprecated and replaced with `UserIdException.MustNotBeReserved`.

Who is affected? This affects developers who have written code that catches the `ReservedUserIdException` while calling the affected methods.

How should I update my code? You should replace catching exception `ReservedUserIdException` with catching exception `UserIdException.MustNotBeReserved`.

Why was this change made? A new pattern has been defined for exceptions that provides higher expressivity in their names and also more information regarding why the exception was thrown.

The new exception `UserIdException.MustNotBeReserved` provides its context and has all the necessary information about why the exception was thrown. In particular, it contains the problematic user ID and the list of reserved user IDs.

Moved the AssetPublisherUtil Class and Removed It from the Public API

- **Date:** 2015-Feb-11
- **JIRA Ticket:** LPS-52744

What changed? The class `AssetPublisherUtil` from the `portal-kernel` module has been moved to the module `AssetPublisher` and it is no longer a part of the public API.

Who is affected? This affects developers who have written code that uses the `AssetPublisherUtil` class.

How should I update my code? This `AssetPublisherUtil` class should no longer be used from other modules since it contains utility methods for the Asset Publisher portlet. If needed, you can define a dependency with the Asset Publisher module and use the new class.

Why was this change made? This change has been made as part of the modularization efforts to decouple the different parts of the portal.

Removed Operations That Used the Fields Class from the StorageAdapter Interface

- **Date:** 2015-Feb-11
- **JIRA Ticket:** LPS-53021

What changed? All operations that used the `Fields` class have been removed from the `StorageAdapter` interface.

Who is affected? This affects developers who have written code that directly calls these operations.

How should I update my code? You should update your code to use the `DDMFormValues` class instead of the `Fields` class.

Why was this change made? This change has been made due to the deprecation of the `Fields` class.

Created a New getType Method That is Implemented in DLProcessor

- **Date:** 2015-Feb-17
- **JIRA Ticket:** LPS-53574

What changed? The `DLProcessor` interface has a new method `getType()`.

Who is affected? This affects developers who have created a `DLProcessor`.

How should I update my code? You should implement the new method and return the type of processor. You can check the class `DLProcessorConstants` to see processor types.

Why was this change made? Previous to Liferay 7, developers were forced to extend one of the existing `DLProcessor` classes and developers using the extended class had to check the instance of that class to determine its processor type.

With this change, developers no longer need to extend any particular class to create their own `DLProcessor` and their processor's type can be clearly specified by a constant from the class `DLProcessorConstants`.

Changed the Usage of the liferay-ui:restore-entry Tag

- **Date:** 2015-Mar-01
- **JIRA Ticket:** LPS-54106

What changed? The usage of the taglib tag `liferay-ui:restore-entry` serves a different purpose now. It renders the UI to restore elements from the Recycle Bin.

Who is affected? This affects developers using the tag `liferay-ui:restore-entry`.

How should I update my code? You should replace your calls to the tag with code like the listing below:

```
<aui:script use="liferay-restore-entry">
    new Liferay.RestoreEntry(
    {
            checkEntryURL: '<%= checkEntryURL.toString() %>',
            duplicateEntryURL: '<%= duplicateEntryURL.toString() %>',
            namespace: '<portlet:namespace />'
        }
    );
</aui:script>
```

In the above code, the `checkEntryURL` should be an `ActionURL` of your portlet, which checks whether the current entry can be restored from the Recycle Bin. The `duplicateEntryURL` should be a `RenderURL` of your portlet, that renders the UI to restore the entry, resolving any existing conflicts. In order to generate that URL, you can use the tag `liferay-ui:restore-entry`, which has been refactored for this usage.

Why was this change made? This change allows the Trash portlet to be an independent module. Its actions and views are no longer used by the tag; they are now the responsability of each plugin.

Added Required Parameter resourceClassNameId for DDM Template Search Operations

- **Date:** 2015-Mar-03
- **JIRA Ticket:** LPS-52990

What changed? The DDM template `search` and `searchCount` operations have a new parameter called `resourceClassNameId`.

Who is affected? This affects developers who have direct calls to the `DDMTemplateService` or `DDMTemplateLocalService`.

How should I update my code? You should add the `resourceClassNameId` parameter to your calls. This parameter represents the resource that owns the permission for the DDM template. For example, if the template is a WCM template, the `resourceClassNameId` points to the `JournalArticle`'s `classNameId`. If the template is a DDL template, the `resourceClassNameId` points to the `DDLRecordSet`'s `classNameId`. If the template is an ADT template, the `resourceClassNameId` points to the `PortletDisplayTemplate`'s `classNameId`.

Why was this change made? This change was made in order to implement model resource permissions for DDM templates, such as `VIEW`, `DELETE`, `PERMISSIONS`, and `UPDATE`.

Replaced the Breadcrumb Portlet's Display Styles with ADTs

- **Date:** 2015-Mar-12
- **JIRA Ticket:** LPS-53577

What changed? The custom display styles of the breadcrumb tag added using JSPs no longer work. They have been replaced by Application Display Templates (ADT).

Who is affected? This affects developers that use the following properties:

```
breadcrumb.display.style.default=horizontal

breadcrumb.display.style.options=horizontal,vertical
```

How should I update my code? To style the Breadcrumb portlet, you should use ADTs instead of using custom styles in your JSPs. ADTs can be created from the UI of the portal by navigating to *Site Settings → Application Display Templates*. ADTs can also be created programatically.

Why was this change made? ADTs allow you to change an application's look and feel without changing its JSP code.

Changed Usage of the liferay-ui:ddm-template-selector Tag

- **Date:** 2015-Mar-16
- **JIRA Ticket:** LPS-53790

What changed? The attribute `classNameId` of the `liferay-ui:ddm-template-selector` taglib tag has been renamed `className`.

Who is affected? This affects developers using the `liferay-ui:ddm-template-selector` tag.

How should I update my code? In your `liferay-ui:ddm-template-selector` tags, rename the `classNameId` attribute to `className`.

Why was this change made? Application Display Templates were being referenced by their UUID, which was usually not known by the developer. Referencing all DDM templates by their class name simplifies using this tag.

Changed the Usage of Asset Preview

- **Date:** 2015-Mar-16
- **JIRA Ticket:** LPS-53972

What changed? Instead of directly including the JSP referenced by the `AssetRenderer`'s `getPreviewPath` method to preview an asset, you now use a taglib tag.

Who is affected? This affects developers who have written code that directly calls an `AssetRenderer`'s `getPreviewPath` method to preview an asset.

How should I update my code? JSP code that previews an asset by calling an `AssetRenderer`'s `getPreviewPath` method, such as in the example code below, must be replaced:

```
<liferay-util:include
    page="<%= assetRenderer.getPreviewPath(liferayPortletRequest, liferayPortletResponse) %>"
    portletId="<%= assetRendererFactory.getPortletId() %>"
    servletContext="<%= application %>"
/>
```

To preview an asset, you should instead use the `liferay-ui:asset-display` tag, passing it an instance of the asset entry and an asset renderer preview template. Here's an example of using the tag:

```
<liferay-ui:asset-display
    assetEntry="<%= assetEntry %>"
    template="<%= AssetRenderer.TEMPLATE_PREVIEW %>"
/>
```

Why was this change made? This change simplifies using asset previews.

Added New Methods in the ScreenNameValidator Interface

- **Date:** 2015-Mar-17
- **JIRA Ticket:** LPS-53409

What changed? The `ScreenNameValidator` interface has new methods `getDescription(Locale)` and `getJSValidation()`.

Who is affected? This affects developers who have implemented a custom screen name validator with the `ScreenNameValidator` interface.

How should I update my code? You should implement the new methods introduced in the interface.

- `getDescription(Locale)`: returns a description of what the screen name validator validates.
- `getJSValidation()`: returns the JavaScript input validator on the client side.

Why was this change made? Previous to Liferay 7, validation for user screen name characters was hard-coded in `UserLocalService`. A new portal property named `users.screen.name.special.characters` has been added to provide configurability of special characters allowed in screen names.

In addition, developers can now specify a custom input validator for the screen name on the client side by providing a JavaScript validator in `getJSValidation()`.

Replaced the Language Portlet's Display Styles with ADTs

- **Date:** 2015-Mar-30
- **JIRA Ticket:** LPS-54419

What changed? The custom display styles of the language tag added using JSPs no longer work. They have been replaced by Application Display Templates (ADT).

Who is affected? This affects developers that use the following properties:

```
language.display.style.default=icon

language.display.style.options=icon,long-text
```

How should I update my code? To style the Language portlet, you should use ADTs instead of using custom styles in your JSPs. ADTs can be created from the UI of the portal by navigating to *Site Settings → Application Display Templates*. ADTs can also be created programatically.

Why was this change made? ADTs allow you to change an application's look and feel without changing its JSP code.

Added Required Parameter groupId for Adding Tags, Categories, and Vocabularies

- **Date:** 2015-Mar-31
- **JIRA Ticket:** LPS-54570

What changed? The API for adding tags, categories, and vocabularies now requires passing the `groupId` parameter. Previously, it had to be included in the `ServiceContext` parameter passed to the method.

Who is affected? This affects developers who have direct calls to the following methods:

- `addTag` in `AssetTagService` or `AssetTagLocalService`
- `addCategory` in `AssetCategoryService` or `AssetCategoryLocalService`
- `addVocabulary` in `AssetVocabularyService` or `AssetVocabularyLocalService`
- `updateFolder` in `JournalFolderService` or `JournalFolderLocalService`

How should I update my code? You should add the `groupId` parameter to your calls. This parameter represents the site in which you are creating the tag, category, or vocabulary. It can be obtained from the `themeDisplay` or `serviceContext` using `themeDisplay.getScopeGroupId()` or `serviceContext.getScopeGroupId()`, respectively.

Why was this change made? This change was made in order improve the API. The `groupId` parameter was always required, but it was hidden by the `ServiceContext` object.

Removed the Tags that Start with portlet:icon-

- **Date:** 2015-Mar-31
- **JIRA Ticket:** LPS-54620

What changed? The following tags have been removed:

- `liferay-portlet:icon-close`
- `liferay-portlet:icon-configuration`
- `liferay-portlet:icon-edit`
- `liferay-portlet:icon-edit-defaults`
- `liferay-portlet:icon-edit-guest`
- `liferay-portlet:icon-export-import`
- `liferay-portlet:icon-help`
- `liferay-portlet:icon-maximize`
- `liferay-portlet:icon-minimize`
- `liferay-portlet:icon-portlet-css`
- `liferay-portlet:icon-print`
- `liferay-portlet:icon-refresh`
- `liferay-portlet:icon-staging`

Who is affected? This affects developers who have written code that uses these tags.

How should I update my code? The tag `liferay-ui:icon` can replace the call to the previous tags. All the previous tags have been converted into Java classes that implement the methods that the `icon` tag requires.

See the modules `portlet-configuration-icon-*` in the `modules/apps/web-experience/portlet-configuration` folder.

Why was this change made? These tags were used to generate the configuration icon of portlets. This functionality will now be managed from OSGi modules instead of tags since OSGi modules provide more flexibility and can be included in any app.

Changed the Default Value of the copy-request-parameters Init Parameter for MVC Portlets

- **Date:** 2015-Apr-15
- **JIRA Ticket:** LPS-54798

What changed? The `copy-request-parameters` init parameter's default value is now set to `true` in all portlets that extend `MVCPortlet`.

Who is affected? This affects developers that have created portlets that extend `MVCPortlet`.

How should I update my code? To continue using the property the same way you did before this change was implemented, you'll need to change the default property. To change the property, set the init parameter to `false` in your class extending `MVCPortlet`:

```
javax.portlet.init-param.copy-request-parameters=false
```

Why was this change made? This change was made to allow for backwards compatibility.

Removed Portal Properties Used to Display Sections in Form Navigators

- **Date:** 2015-Apr-16
- **JIRA Ticket:** LPS-54903

What changed? The following portal properties (and the equivalent `PropsKeys` and `PropsValues`) that were used to decide what sections would be displayed in the `form-navigator` have been removed:

- `company.settings.form.configuration`
- `company.settings.form.identification`
- `company.settings.form.miscellaneous`
- `company.settings.form.social`
- `journal.article.form.add`
- `journal.article.form.update`
- `journal.article.form.default.values`
- `layout.form.add`
- `layout.form.update`
- `layout.set.form.update`
- `organizations.form.add.identification`
- `organizations.form.add.main`
- `organizations.form.add.miscellaneous`
- `organizations.form.update.identification`
- `organizations.form.update.main`
- `organizations.form.update.miscellaneous`
- `sites.form.add.advanced`
- `sites.form.add.main`
- `sites.form.add.miscellaneous`
- `sites.form.add.seo`
- `sites.form.update.advanced`
- `sites.form.update.main`
- `sites.form.update.miscellaneous`

- `sites.form.update.seo`
- `users.form.add.identification`
- `users.form.add.main`
- `users.form.add.miscellaneous`
- `users.form.my.account.identification`
- `users.form.my.account.main`
- `users.form.my.account.miscellaneous`
- `users.form.update.identification`
- `users.form.update.main`
- `users.form.update.miscellaneous`

The sections and categories of form navigators are now OSGi components.

Who is affected? This affects administrators who may have added, removed, or reordered sections using those portal properties. Developers using the constants defined in `PropsKeys` or `PropsValues` for those portal properties will also be affected.

How should I update my code? Since those properties no longer exist, you cannot rely on them. References to the constants of `PropsKeys` and `PropsValues` will need to be updated. You can use `FormNavigatorCategoryUtil` and `FormNavigatorEntryUtil` to obtain a list of the available sections and categories for a form navigator instance.

Changes to remove or reorder specific sections will need to be done through the OSGi console to update the service ranking or stop the components.

Adding new sections with Liferay Hooks will still work as a legacy feature, but the recommended way is using OSGi components to add new sections.

Why was this change made? The old mechanism to add new sections to `form-navigator` tags was very limited because it could only depend on portal for services and utils due to the new section that was rendered from the portal classloader.

There was a need to add new sections and categories to `form-navigator` tags via OSGi plugins in a more extensible way, allowing the developer to include new sections to access to their own utils and services.

Removed the Type Setting breadcrumbShowParentGroups from Groups

- **Date:** 2015-Apr-21
- **JIRA Ticket:** LPS-54791

What changed? The type setting `breadcrumbShowParentGroups` was removed from groups and is no longer available in the site configuration. Now, it is only available in the breadcrumb configuration.

Who is affected? This affects all site administrators that have set the `showParentGroups` preference in Site Administration.

How should I update my code? There are no code updates required. This should only be updated at the portlet instance level.

Why was this change made? This change was introduced to support the new Settings API.

Changed Return Value of the Method getText of the Editor's Window API

- **Date:** 2015-Apr-28
- **JIRA Ticket:** LPS-52698

What changed? The method `getText` now returns the editor's content, without any HTML markup.

Who is affected? This affects developers that are using the `getText` method of the editor's window API.

How should I update my code? To continue using the editor the same way you did before this change was implemented, you should change calls to the `getText` method to instead call the `getHTML` method.

Why was this change made? This change was made in the editor's window API to provide a proper `getText` method that returns just the editor's content, without any HTML markup. This change is used for the blog abstract field.

Moved the Contact Name Exception Classes to Inner Classes of ContactNameException

- **Date:** 2015-May-05
- **JIRA Ticket:** LPS-55364

What changed? The use of classes `ContactFirstNameException`, `ContactFullNameException`, and `ContactLastNameException` has been moved to inner classes in a new class called `ContactNameException`.

Who is affected? This affects developers who may have included one of the three classes above in their code.

How should I update my code? While the old classes remain for backwards-compatibility, they are being deprecated. You're encouraged to use the new pattern of inner classes for exceptions wherever possible. For example, instead of using `ContactFirstNameExeception`, use `ContactNameException.MustHaveFirstName`.

Why was this change made? This change was made in accordance with the new exceptions pattern being applied throughout Portal. It also allows the new localized user name configuration feature to be thoroughly covered by exceptions for different configurations.

Removed USERS_LAST_NAME_REQUIRED from portal.properties in Favor of language.properties Configurations

- **Date:** 2015-May-07
- **JIRA Ticket:** LPS-54956

What changed? The `USERS_LAST_NAME_REQUIRED` property has been removed from `portal.properties` and the corresponding UI. Required names are now handled on a per-language basis via the `language.properties` files. It has also been removed as an option from the Portal Settings section of the Control Panel.

Who is affected? This affects anyone who uses the `USERS_LAST_NAME_REQUIRED` portal property.

How should I update my code? If you need to require the user's last name, list it on the `lang.user.name.required.field.names` line of the appropriate `language.properties` files:

```
lang.user.name.required.field.names=last-name
```

Why was this change made? Portal property `USERS_LAST_NAME_REQUIRED` didn't support the multicultural user name configurations introduced in LPS-48406. Language property files (e.g., `language.properties`) now support these configurations. Control of all user name configuration, except with regards to first name, is relegated to language property files. First name is required and always present.

Removed Methods getGroupLocalRepositoryImpl and getLocalRepositoryImpl from RepositoryLocalService and RepositoryService

- **Date:** 2015-May-14
- **JIRA Ticket:** LPS-55566

What changed? The methods `getGroupLocalRepositoryImpl(...)` and `getLocalRepositoryImpl(...)` have been removed from `RepositoryLocalService` and `RepositoryService`. Although the methods are related to the service, they belong in a different level of abstraction.

Who is affected? This affects anyone who uses those methods.

How should I update my code? The removed methods were generic and had long signatures with optional parameters. They now have one specialized version per parameter and are in the `RepositoryProvider` service.

Example

Old call:

```
RepositoryLocalServiceUtil.getRepositoryImpl(0, fileEntryId, 0)
```

New call:

```
RepositoryProviderUtil.getLocalRepositoryByFileEntryId(fileEntryId)
```

Why was this change made? This change was made to enhance the Repository API and facilitate decoupling the API from the Document Library, as a part of the portal modularization effort.

Removed addFileEntry Method from DLAppHelperLocalService

- **Date:** 2015-May-20
- **JIRA Ticket:** LPS-47645

What changed? The `addFileEntry` method has been removed from `DLAppHelperLocalService`.

Who is affected? This affects anyone who calls the `addFileEntry` method.

How should I update my code? If you need to invoke the `addFileEntry` method as part of a custom repository implementation, use the provided repository capabilities instead. See `LiferayRepositoryDefiner` for examples on their use.

For other use cases, you may need to explicitly invoke each of the service methods used by `addFileEntry`.

Why was this change made? The logic inside the `addFileEntry` method was moved, out from `DLAppHelperLocalService` and into repository capabilities, to further decouple core repository implementations from additional (optional) functionality.

Indexers Called from Document Library Now Receive FileEntry Instead of DLFileEntry

- **Date:** 2015-May-20
- **JIRA Ticket:** LPS-55613

What changed? Indexers that previously received a `DLFileEntry` object (e.g., in the `addRelatedEntryFields` method) no longer receive a `DLFileEntry`, but a `FileEntry`.

Who is affected? This affects anyone who implements an Indexer handling `DLFileEntry` objects.

How should I update my code? You should try to use methods in `FileEntry` or exported repository capabilities to obtain the value you were using. If no capability exists for your use case, you can resort to calling `fileEntry.getModel()` and casting the result to a `DLFileEntry`. However, this breaks all encapsulation and may result in future failures or compatibility problems.

Old code:

```
@Override
public void addRelatedEntryFields(Document document, Object obj)
    throws Exception {

    DLFileEntry dlFileEntry = (DLFileEntry)obj;

    long fileEntryId = dlFileEntry.getFileEntryId();
```

New Code:

```
@Override
public void addRelatedEntryFields(Document document, Object obj)
    throws Exception {

    FileEntry fileEntry = (FileEntry)obj;

    long fileEntryId = fileEntry.getFileEntryId();
```

Why was this change made? This change was made to enhance the Repository API and make decoupling from Document Library easier when modularizing the portal.

Removed permissionClassName, permissionClassPK, and permissionOwner Parameters from MBMessage API

- **Date:** 2015-May-27
- **JIRA Ticket:** LPS-55877

What changed? The parameters `permissionClassName`, `permissionClassPK`, and `permissionOwner` have been removed from the Message Boards API and Discussion tag.

Who is affected? This affects anyone who invokes the affected methods (locally or remotely) and any view that uses the Discussion tag.

How should I update my code? It suffices to remove the parameters from the method calls (for consumers of the API) or the attributes in tag invocations.

Why was this change made? Those API methods were exposed in the remote services, allowing any consumer to bypass the permission system by providing customized `className`, `classPK`, or `ownerId` parameters.

Moved Indexer.addRelatedEntryFields and Indexer.reindexDDMStructures, and Removed Indexer.getQueryString

- **Date:** 2015-May-27
- **JIRA Ticket:** LPS-55928

What changed? Method `Indexer.addRelatedEntryFields(Document, Object)` has been moved into `RelatedEntryIndexer`.

`Indexer.reindexDDMStructures(List<Long>)` has been moved into `DDMStructureIndexer`.

`Indexer.getQueryString(SearchContext, Query)` has been removed, in favor of calling `SearchEngineUtil.getQueryString(S`
`Query)`

Who is affected? This affects any code that invokes the affected methods, as well as any code that implements the interface methods.

How should I update my code? Any code implementing `Indexer.addRelatedEntryFields(...)` should implement the `RelatedEntryIndexer` interface.

Any code calling `Indexer.addRelatedEntryFields(...)` should determine first if the `Indexer` is an instance of `RelatedEntryIndexer`.

Old code:

```
mbMessageIndexer.addRelatedEntryFields(...);
```

New code:

```
if (mbMessageIndexer instanceof RelatedEntryIndexer) {
    RelatedEntryIndexer relatedEntryIndexer =
        (RelatedEntryIndexer)mbMessageIndexer;

    relatedEntryIndexer.addRelatedEntryFields(...);
}
```

Any code implementing `Indexer.reindexDDMStructures(...)` should implement the `DDMStructureIndexer` interface.

Any code calling `Indexer.reindexDDMStructures(...)` should determine first if the `Indexer` is an instance of `DDMStructureIndexer`.

Old code:

```
mbMessageIndexer.reindexDDMStructures(...);
```

New code:

```
if (journalIndexer instanceof DDMStructureIndexer) {
    DDMStructureIndexer ddmStructureIndexer =
        (DDMStructureIndexer)journalIndexer;

    ddmStructureIndexer.reindexDDMStructures(...);
}
```

Any code calling `Indexer.getQueryString(...)` should call `SearchEngineUtil.getQueryString(...)`.

Old code:

```
mbMessageIndexer.getQueryString(...);
```

New code:

```
SearchEngineUtil.getQueryString(...);
```

Why was this change made? The `addRelatedEntryFields` and `reindexDDMStructures` methods were not related to core indexing functions. They were functions of specialized indexers.

The `getQueryString` method was an unnecessary convenience method.

Removed mbMessages and fileEntryTuples Attributes from app-view-search-entry Tag

- **Date:** 2015-May-27
- **JIRA Ticket:** LPS-55886

What changed? The `mbMessages` and `fileEntryTuples` attributes from the `app-view-search-entry` tag have been removed. Related methods `getMbMessages`, `getFileEntryTuples`, and `addMbMessage` have also been removed from the `SearchResult` class.

Who is affected? This affects developers that use the `app-view-search-entry` tag in their views, have developed hooks to customize the tag JSP, or have developed a portlet that uses that tag. Also, any custom code that uses the `SearchResult` class may be affected.

How should I update my code? The new attributes `commentRelatedSearchResults` and `fileEntryRelatedSearchResults` should be used instead. The expected value is the one returned by the `getCommentRelatedSearchResults` and `getFileEntryRelatedSearchResults` methods in `SearchResult`.

When adding comments to the `SearchResult`, the new `addComment` method should be used instead of the `addMbMessage` method.

Why was this change made? As part of the modularization efforts, references to `MBMessage` needed to be removed for the Message Boards portlet to be placed into its own OSGi bundle.

Replaced Method getPermissionQuery with getPermissionFilter in SearchPermissionChecker, and getFacetQuery with getFacetBooleanFilter in Indexer

- **Date:** 2015-Jun-02
- **JIRA Ticket:** LPS-56064

What changed? Method `SearchPermissionChecker.getPermissionQuery( long, long[], long, String, Query, SearchContext)` has been replaced by `SearchPermissionChecker.getPermissionBooleanFilter( long, long[], long, String, BooleanFilter, SearchContext)`.

Method `Indexer.getFacetQuery(String, SearchContext)` has been replaced by `Indexer.getFacetBooleanFilter(String, SearchContext)`.

Who is affected? This affects any code that invokes the affected methods, as well as any code that implements the interface methods.

How should I update my code? Any code calling/implementing `SearchPermissionChecker.getPermissionQuery(...)` should instead call/implement `SearchPermissionChecker.getPermissionBooleanFilter(...)`.

Any code calling/implementing `Indexer.getFacetQuery(...)` should instead call/implement `Indexer.getFacetBooleanFilter(...)`.

Why was this change made? Permission constraints placed on search should not affect the score for returned search results. Thus, these constraints should be applied as search filters. `SearchPermissionChecker` is also a very deep internal interface within the permission system. Thus, to limit confusion in the logic for maintainability, the `SearchPermissionChecker.getPermissionQuery(...)` method was removed as opposed to deprecated.

Similarly, constraints applied to facets should not affect the scoring or facet counts. Since `Indexer.getFacetQuery(...)` was only utilized by the `AssetEntriesFacet`, and used to reduce the impact of changes for `SearchPermissionChecker.getPermissionBooleanFilter(...)`, the method was removed as opposed to deprecated.

Added userId Parameter to Update Operations of DDMStructureLocalService and DDMTemplateLocalService

- **Date:** 2015-Jun-05
- **JIRA Ticket:** LPS-50939

What changed? A new parameter `userId` has been added to the `updateStructure` and `updateTemplate` methods of the `DDMStructureLocalService` and `DDMTemplateLocalService` classes, respectively.

Who is affected? This affects any code that invokes the affected methods, as well as any code that implements the interface methods.

How should I update my code? Any code calling/implementing `DDMStructureLocalServiceUtil.updateStructure(...)` or `DDMTemplateLocalServiceUtil.updateTemplate(...)` should pass the new `userId` parameter.

Why was this change made? For the service to keep track of which user is modifying the structure or template, the `userId` parameter was required. In order to add support to structure and template versions, audit columns were also added to such models.

Removed Method getEntries from DL, DLImpl, and DLUtil Classes

- **Date:** 2015-Jun-10
- **JIRA Ticket:** LPS-56247

What changed? The method `getEntries` has been removed from the `DL`, `DLImpl`, and `DLUtil` classes.

Who is affected? This affects any caller of the `getEntries` method.

How should I update my code? You may use the `SearchResultUtil` class to process the search results. Note that this class is not completely equivalent; if you need exactly the same behavior as the removed method, you will need to add custom code.

Why was this change made? The `getEntries` method was no longer used, and contained hardcoded references to classes that will be moved into OSGi bundles.

Removed WikiUtil.getEntries Method

- **Date:** 2015-Jun-10
- **JIRA Ticket:** LPS-56242

What changed? The method `getEntries()` has been removed from class `WikiUtil`.

Who is affected? Any JSP hook or ext plugin that uses this method is affected. As the class was located in portal-impl, regular portlets and other safe extension points won't be affected.

How should I update my code? You should review the JSP or ext plugin, updating it to remove any reference to the new class and mimicking the original JSP code. In case you need equivalent functionality to the one provided by `WikiUtil.getEntries()` you may use the `SearchResultUtil` class. While not totally equivalent, it offers similar functionality.

Why was this change made? The `WikiUtil.getEntries()` method was no longer used, and it contained hardcoded references to classes that will be moved into OSGi modules.

Removed render Method from ConfigurationAction API

- **Date:** 2015-Jun-14
- **JIRA Ticket:** LPS-56300

What changed? The method `render` has been removed from the interface `ConfigurationAction`.

Who is affected? This affects any Java code calling the method `render` on a `ConfigurationAction` class, or Java classes overriding the `render` method of a `ConfigurationAction` class.

How should I update my code? The method `render` was used to return the path of a JSP, including the configuration of a portlet. That method is now available for configurations extending the `BaseJSPSettingsConfigurationAction` class, and is called `getJspPath`.

If any logic was added to override the `render` method, it can now be added in the `include` method.

Why was this change made? This change was part of needed modifications to support adding configuration for portlets based on other technology different than JSP (e.g., FreeMarker). The method `include` can now be used to create configuration UIs written in FreeMarker or any other framework.

Removed ckconfig Files Used for CKEditor Configuration

- **Date:** 2015-Jun-16
- **JIRA Ticket:** LPS-55518

What changed? The files `ckconfig.jsp`, `ckconfig-ext.jsp`, `ckconfig_bbcode.jsp`, `ckconfig_bbcode-ext.jsp`, `ckconfig_creole.jsp`, and `ckconfig_creole-ext.jsp` have been removed and are no longer used to configure the CKEditor instances created using the `liferay-ui:input-editor` tag.

Who is affected? This affects any hook or plugin-ext overriding these files to modify the editor configuration.

How should I update my code? Depending on the changes, different extension methods are available:

- For CKEditor configuration options, an implementation of `EditorConfigContributor` can be created to pass or modify the expected parameters.
- For CKEditor instance manipulation (setting attributes, adding listeners, etc.), the `DynamicInclude` extension point `#ckeditor[_creole|_bbcode]#onEditorCreated` has been added to provide the possibility of injecting JavaScript, when needed.

Why was this change made? This change is part of a greater effort to provide mechanisms to extend and configure any editor in Liferay Portal in a coherent and extensible way.

Renamed ActionCommand Classes Used in the MVCPortlet Framework

- **Date:** 2015-Jun-16
- **JIRA Ticket:** LPS-56372

What changed? The classes located in the `com.liferay.portal.kernel.portlet.bridges.mvc` package have been renamed to include the *MVC* prefix.

Old Classes:

- `BaseActionCommand`
- `BaseTransactionalActionCommand`
- `ActionCommand`
- `ActionCommandCache`

New Classes:

- `BaseMVCActionCommand`
- `BaseMVCTransactionalActionCommand`
- `MVCActionCommand`
- `MVCActionCommandCache`

Also, the property `action.command.name` has been renamed to `mvc.command.name`. The code snippet below shows the new property in its context.

```
@Component(
    immediate = true,
    property = {
            "javax.portlet.name=" + InvitationPortletKeys.INVITATION,
            "mvc.command.name=view"
    },
    service = MVCActionCommand.class
)
```

Who is affected? This affects any Java code calling the `ActionCommand` classes used in the `MVCPortlet` framework.

How should I update my code? You should update the old `ActionCommand` class names with the new *MVC* prefix.

Why was this change made? This change adds consistency to the MVC framework, and makes it self-explanatory what classes should be used for the MVC portlet.

Extended MVC Framework to Use Same Key for Registering ActionURL and ResourceURL

- **Date:** 2015-Jun-16
- **JIRA Ticket:** LPS-56372

What changed? Previously, a single `ActionCommand` was valid for both `ActionURL` and `ResourceURL`. Now you must distinguish both an `ActionURL` and `ResourceURL` as different actions, which means you can register both with the same key.

Who is affected? This affects developers that were using the `ActionCommand` for `actionURLs` and `resourceURLs`.

How should I update my code? You should replace the `ActionCommands` used for `actionURLs` and `resourceURLs` to use `MVCActionCommand` and `MVCResourceCommand`, respectively. For example, for the new `MVCResourceCommand`, you'll need to use the `resourceID` of the `resourceURL` instead of using `ActionRequest.ACTION_NAME`.

Old Code:

```
<liferay-portlet:resourceURL copyCurrentRenderParameters="<%= false %>" var="exportRecordSetURL">
    <portlet:param name="<%= ActionRequest.ACTION_NAME %>" value="exportRecordSet" />
    <portlet:param name="recordSetId" value="<%= String.valueOf(recordSet.getRecordSetId()) %>" />
</liferay-portlet:resourceURL>
```

New Code:

```
<liferay-portlet:resourceURL copyCurrentRenderParameters="<%= false %>" id="exportRecordSet" var="exportRecordSetURL">
    <portlet:param name="recordSetId" value="<%= String.valueOf(recordSet.getRecordSetId()) %>" />
</liferay-portlet:resourceURL>
```

Why was this change made? This change was made to extend the MVC framework to have better support for `actionURLs` and `resourceURLs`.

Changed Java Package Names for Portlets Extracted as Modules

- **Date:** 2015-Jun-29
- **JIRA Ticket:** LPS-56383 and others

What changed? The Java package names changed for portlets that were extracted as OSGi modules in 7.0. Here is the complete list:

- `com.liferay.portlet.bookmarks` → `com.liferay.bookmarks`
- `com.liferay.portlet.dynamicdatalists` → `com.liferay.dynamicdatalists`
- `com.liferay.portlet.journal` → `com.liferay.journal`
- `com.liferay.portlet.polls` → `com.liferay.polls`
- `com.liferay.portlet.wiki` → `com.liferay.wiki`

Who is affected? This affects developers using the portlets API from their own plugins.

How should I update my code? Update the package imports to use the new package names. Any literal usage of the portlet `className` should also be updated.

Why was this change made? Package names have been adapted to the new condition of Liferay portlets as OSGi services.

Removed the DLFileEntryTypes_DDMStructures Mapping Table

- **Date:** 2015-Jul-01
- **JIRA Ticket:** LPS-56660

What changed? The `DLFileEntryTypes_DDMStructures` mapping table is no longer available.

Who is affected? This affects developers using the Document Library File Entry Type Local Service API.

How should I update my code? Update the calls to `addDDMStructureLinks`, `deleteDDMStructureLinks`, and `updateDDMStructureLinks` if you want to add, delete, or update references between `DLFileEntryType` and `DDMStructures`.

Why was this change made? This change was made to reduce the coupling between the two applications.

Removed render Method from AssetRenderer API and WorkflowHandler API

- **Date:** 2015-Jul-03
- **JIRA Ticket:** LPS-56705

What changed? The method `render` has been removed from the interfaces `AssetRenderer` and `WorkflowHandler`.

Who is affected? This affects any Java code calling the method `render` on an `AssetRenderer` or `WorkflowHandler` class, or Java classes overriding the `render` method of these classes.

How should I update my code? The method `render` was used to return the path of a JSP, including the configuration of a portlet. That method is now available for the same AssetRender API extending the `BaseJSPAssetRenderer` class, and is called `getJspPath`.

If any logic was added to override the `render` method, it can now be added in the `include` method.

Why was this change made? This change was part of needed modifications to support adding asset renderers and workflow handlers for portlets based on other technology different than JSP (e.g., FreeMarker). The method `include` can now be used to create asset renderers or workflow handlers with UIs written in FreeMarker or any other framework.

Renamed ADMIN_INSTANCE to PORTAL_INSTANCES in PortletKeys

- **Date:** 2015-Jul-08
- **JIRA Ticket:** LPS-56867

What changed? The constant `PortletKeys.ADMIN_INSTANCE` has been renamed as `PortletKeys.PORTAL_INSTANCES`.

Who is affected? This affects developers using the old constant in their code; for example, creating a direct link to it. This is not common and usually not a good practice, so this should not affect many people.

How should I update my code? You should rename the constant `ADMIN_INSTANCE` to `PORTAL_INSTANCES` everywhere it is used.

Why was this change made? This change was part of needed modifications to extract the Portal Instances portlet from the Admin portlet. The constant's old name was not accurate, since it originated from the old Admin portlet. Since the Portal Instances portlet is now extracted to its own module, the old name no longer resembles its usage.

Removed Support for filterFindBy Generation or InlinePermissionUtil Usage for Tables When the Primary Key Type Is Not long

- **Date:** 2015-Jul-21
- **JIRA Ticket:** LPS-54590

What changed? ServiceBuilder and inline permission filter support has been removed for non-`long` primary key types.

Who is affected? This affects code that is using `int`, `float`, `double`, `boolean`, or `short` type primary keys in the `service.xml` with inline permissions.

How should I update my code? You should change the primary key type to `long`.

Why was this change made? Inline permissioning was using the `join` method between two different data types and that caused significant performance degradation with `filterFindBy` queries.

Removed Vaadin 6 from Liferay Core

- **Date:** 2015-Jul-31
- **JIRA Ticket:** LPS-57525

What changed? The bundled Vaadin 6.x JAR file has been removed from portal core.

Who is affected? This affects developers who are creating Vaadin portlet applications in Liferay Portal.

How should I update my code? You should upgrade to Vaadin 7, bundle your `vaadin.jar` with your plugin, or deploy Vaadin libraries to Liferay's OSGi container.

Why was this change made? Vaadin 6.x is outdated and there are no plans for any new projects to be created with it. Therefore, developers should begin using Vaadin 7.x.

Replaced the Navigation Menu Portlet's Display Styles with ADTs

- **Date:** 2015-Jul-31
- **JIRA Ticket:** LPS-27113

What changed? The custom display styles of the navigation tag added using JSPs no longer work. They have been replaced by Application Display Templates (ADT).

Who is affected? This affects developers that use portlet properties with the following prefix:

```
navigation.display.style
```

This also affects developers that use the following attribute in the navigation tag:

```
displayStyleDefinition
```

How should I update my code? To style the Navigation portlet, you should use ADTs instead of using custom styles in your JSPs. ADTs can be created from the UI of the portal by navigating to *Site Settings → Application Display Templates*. ADTs can also be created programatically.

Developers should use the `ddmTemplateGroupId` and `ddmTemplateKey` attributes of the navigation tag to set the ADT that defines the style of the navigation.

Why was this change made? ADTs allow you to change an application's look and feel without changing its JSP code.

Renamed URI Attribute Used to Generate AUI Tag Library

- **Date:** 2015-Aug-12
- **JIRA Ticket:** LPS-57809

What changed? The URI attribute used to identify the AUI taglib has been renamed.

Who is affected? This affects developers that use the URI `http://alloy.liferay.com/tld/aui` in their JSPs, XMLs, etc.

How should I update my code? You should use the new AUI URI declaration:

Old:

```
http://alloy.liferay.com/tld/aui
```

New:

```
http://liferay.com/tld/aui
```

Why was this change made? To stay consistent with other taglibs provided by Liferay, the AUI `.tld` file was modified to start with the prefix `liferay-`. Due to this change, the XML files used to automatically generate the AUI taglib were modified, changing the AUI URI declaration.

Removed Support for runtime-portlet Tag in Body of Web Content Articles

- **Date:** 2015-Sep-17
- **JIRA Ticket:** LPS-58736

What changed? The tag `runtime-portlet` is no longer replaced by a portlet if it is found in the body of a web content article.

Who is affected? This affects any web content in the database (`JournalArticle` table) that uses this tag.

How should I update my code? Embedding another portlet is only supported from a template. You should embed the portlet by passing its name in a call to `theme.runtime` or using the right tag in FreeMarker.

Example

In Velocity:

```
$theme.runtime("145")
```

In FreeMarker:

```
<#assign liferay_portlet = PortalJspTagLibs["/WEB-INF/tld/liferay-portlet-ext.tld"] />

<@liferay_portlet["runtime"] portletName="145" />
```

Why was this change made? This change improves the performance of web content articles while enforcing a single way to embed portlets into the page for better testing.

Removed the liferay-ui:control-panel-site-selector Tag

- **Date:** 2015-Sep-23
- **JIRA Ticket:** LPS-58210

What changed? The tag `liferay-ui:control-panel-site-selector` has been deleted.

Who is affected? This affects developers who use this tag in their code.

How should I update my code? You should consider using the tag `liferay-ui:my-sites`, or create your own markup using the `GroupService` API.

Why was this change made? This tag is no longer used and will no longer be maintained properly.

Removed Methods Related to Control Panel in PortalUtil

- **Date:** 2015-Sep-23
- **JIRA Ticket:** LPS-58210

What changed? The following methods have been deleted:

- `getControlPanelCategoriesMap`
- `getControlPanelCategory`
- `getControlPanelPortlets`
- `getFirstMyAccountPortlet`
- `getFirstSiteAdministrationPortlet`
- `getSiteAdministrationCategoriesMap`
- `getSiteAdministrationURL`
- `isCompanyControlPanelVisible`

Who is affected? This affects developers that use any of the methods listed above.

How should I update my code? In order to work with applications displayed in the Product Menu, developers should call the `PanelCategoryRegistry` and `PanelAppRegistry` classes located in the `application-list-api` module. These classes allow developers to interact with categories and applications in the Control Panel.

Why was this change made? These methods are no longer used and they will not work properly since they cannot call the `application-list-api` from the portal context.

Removed ThemeDisplay Methods Related to Control Panel and Site Administration

- **Date:** 2015-Sep-23
- **JIRA Ticket:** LPS-58210

What changed? The following methods have been deleted:

- `getControlPanelCategory`
- `getURLSiteAdministration`

Who is affected? This affects developers that use either of the methods listed above.

How should I update my code? Site Administration is not a site per se; some applications are displayed in that context. To create a link to an application that is displayed in Site Administration, developers should use the method `PortalUtil.getControlPanelURL`. In order to obtain the first application displayed in a section of the Product Menu, developers should use the `application-list-api` module to call the `PanelCategoryRegistry` and `PanelAppRegistry` classes.

Why was this change made? These methods are no longer used and they will not work properly since they cannot call the `application-list-api` from the portal context.

Removed Control Panel from List of Sites Returned by Methods Group.getUserSitesGroups and User.getMySiteGroups

- **Date:** 2015-Sep-23
- **JIRA Ticket:** LPS-58862

What changed? The following methods had a boolean parameter to determine whether to include the Control Panel group:

- `Group.getUserSitesGroups`
- `User.getMySiteGroups`

This boolean parameter should no longer be used.

Who is affected? This affects developers that use either of the methods listed above passing the `includeControlPanel` parameter as `true`.

How should I update my code? If you don't need the Control Panel, remove the `false` parameter. If you still want to obtain a link to the Control Panel, you should do it in a different way.

The Control Panel is not a site per se; some applications are displayed in that context. To create a link to an application that is displayed in the Control Panel, developers should use the method `PortalUtil.getControlPanelURL`. In order to obtain the first application displayed in a section of the Product Menu, developers should use the `application-list-api` module to call the `PanelCategoryRegistry` and `PanelAppRegistry` classes.

Why was this change made? The Control Panel is no longer a site per se, but just a context in which some applications are displayed. This concept conflicts with the idea of returning a site called Control Panel in the Sites API.

Changed Exception Thrown by Documents and Media Services When Duplicate Files are Found

- **Date:** 2015-Sep-24
- **JIRA Ticket:** LPS-53819

What changed? When a duplicate file entry is found by Documents and Media (D&M) services, a `DuplicateFileEntryException` will be thrown. Previously, the exception `DuplicateFileException` was used.

The `DuplicateFileException` is now raised only by `Store` implementations.

Who is affected? Any caller of the `addFileEntry` methods in `DLApp` and `DLFileEntry` local and remote services is affected.

How should I update my code? Change the exception type from `DuplicateFileException` to `DuplicateFileEntryException` in `try-catch` blocks surrounding calls to D&M services.

Why was this change made? The `DuplicateFileException` exception was used in two different contexts:

- When creating a new file through D&M and a row in the database already existed for a file entry with the same title.
- When the stores tried to save a file and the underlying storage unit (a file in the case of `FileSystemStore`) already existed.

This made it impossible to detect and recover from store corruption issues, as they were undifferentiable from other errors.

Removed All References to Windows Live Messenger

- **Date:** 2015-Oct-15
- **JIRA Ticket:** LPS-30883

What changed? All references to the `msnSn` column in the Contacts table have been removed from portal. All references to Windows Live Messenger have been removed from properties, tests, classes, and the frontend. Also, the `getMsnSn` and `setMsnSn` methods have been removed from the `Contact` and `LDAPUser` models.

The following classes have been removed:

- `MSNConnector`

- `MSNMessageAdapter`

The following constants have been removed:

- `CalEventConstants.REMIND_BY_MSN`
- `ContactConverterKeys.MSN_SN`
- `PropsKeys.MSN_LOGIN`
- `PropsKeys.MSN_PASSWORD`

The following methods have been removed:

- `Contact.getMsnSn`
- `Contact.setMsnSn`
- `LDAPUser.getMsnSn`
- `LDAPUser.setMsnSn`

The following methods have been changed:

- `AdminUtil.updateUser`
- `ContactLocalServiceUtil.addContact`
- `ContactLocalServiceUtil.updateContact`
- `UserLocalServiceUtil.addContact`
- `UserLocalServiceUtil.updateContact`
- `UserLocalServiceUtil.updateUser`
- `UserServiceUtil.updateUser`

Who is affected? This affects developers who use any of the classes, constants, or methods listed above.

How should I update my code? When updating or adding a user or contact using one of the changed methods above, remove the `msnSn` argument from the method call. If you are using one of the removed items above, you should remove all references to them from your code and look for alternatives, if necessary. Lastly, remove any references to the `msnSN` column in the Contacts table from your SQL queries.

Why was this change made? Since Microsoft dropped support for Windows Live Messenger, Liferay will no longer continue to support it.

Removed Support for AIM, ICQ, MySpace, and Yahoo Messenger

- **Date:** 2015-Oct-22
- **JIRA Ticket:** LPS-59716

What changed? Liferay no longer supports integration with MySpace and AIM, ICQ, and Yahoo Messenger instant messaging services. The corresponding `aimSn`, `icqSn`, `mySpaceSn`, and `ymSn` columns have been removed from the `Contacts` table.

The following classes have been removed:

- `AIMConnector`
- `ICQConnector`
- `YMConnector`

The following constants have been removed:

- `CalEventConstants.REMIND_BY_AIM`
- `CalEventConstants.REMIND_BY_ICQ`
- `CalEventConstants.REMIND_BY_YM`
- `ContactConverterKeys.AIM_SN`
- `ContactConverterKeys.ICQ_SN`
- `ContactConverterKeys.MYSPACE_SN`
- `ContactConverterKeys.YM_SN`
- `PropsKeys.AIM_LOGIN`
- `PropsKeys.AIM_PASSWORD`
- `PropsKeys.ICQ_JAR`
- `PropsKeys.ICQ_LOGIN`
- `PropsKeys.ICQ_PASSWORD`
- `PropsKeys.YM_LOGIN`
- `PropsKeys.YM_PASSWORD`

The following methods have been removed:

- `getAimSn`
- `getIcqSn`
- `getMySpaceSn`
- `getYmSn`
- `setAimSn`
- `setIcqSn`
- `setMySpaceSn`
- `setYmSn`

The following methods have been changed:

- `updateUser`
- `addContact`

The following portal properties have been removed:

- `aim.login`
- `aim.password`
- `icq.jar`
- `icq.login`
- `icq.password`
- `ym.login`
- `ym.password`

Who is affected? This affects developers who use any of the classes, constants, methods, or properties listed above.

How should I update my code? When updating or adding a user or contact using one of the changed methods above, remove the `aimSn`, `icqSn`, `mySpaceSn`, and `ymSn` arguments from the method call. If you are using one of the removed items above, you should remove all references to them from your code and look for alternatives, if necessary. Lastly, remove from your SQL queries any references to former `Contacts` table columns `aimSn`, `icqSn`, `mySpaceSn`, and `ymSn`.

Also, a reference to any one of the removed portal properties above no longer returns a value.

Why was this change made? The services removed in this change are no longer popular enough to merit continued support.

Removed All Methods from SchedulerEngineHelper that Explicitly Schedule Jobs Using SchedulerEntry or Specify MessageListener Class Names

- **Date:** 2015-Oct-29
- **JIRA Ticket:** LPS-59681

What changed? The following methods were removed from `SchedulerEngine`:

- `SchedulerEngineHelper.addJob(Trigger, StorageType, String, String, Message, String, String, int)`
- `SchedulerEngineHelper.addJob(Trigger, StorageType, String, String, Object, String, String, int)`
- `SchedulerEngineHelper.schedule(SchedulerEntry, StorageType, String, int)`

Who is affected? This affects developers that use the above methods to schedule jobs into the `SchedulerEngine`.

How should I update my code? You should update your code to call one of these methods:

- `SchedulerEngineHelper.schedule(Trigger, StorageType, String, String, Message, int)`
- `SchedulerEngineHelper.schedule(Trigger, StorageType, String, String, Object, int)`

Instead of simply providing the class name of your scheduled job listener, you should follow these steps:

1. Instantiate your MessageListener.
2. Call `SchedulerEngineHelper.register(MessageListener, SchedulerEntry)` to register your `SchedulerEventMessageListen`

Why was this change made? The deleted methods provided facilities that aren't compatible with using declarative services in an OSGi container. The new approach allows for proper injection of dependencies into scheduled event message listeners.

Removed the asset.publisher.query.form.configuration Property

- **Date:** 2015-Nov-03
- **JIRA Ticket:** LPS-60119

What changed? The `asset.publisher.query.form.configuration` property has been removed from `portal.properties`.

Who is affected? This affects any hook that uses the `asset.publisher.query.form.configuration` property.

How should I update my code? If you are using this property to generate the UI for an Asset Entry Query Processor, your Asset Entry Query Processor must now implement the `include` method to generate the UI.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal.

Removed Hover and Alternate Style Features of Search Container Tag

- **Date:** 2015-Nov-03
- **JIRA Ticket:** LPS-58854

What changed? The following attributes and methods have been removed:

- The attribute `hover` of the `liferay-ui:search-container` tag.
- The method `isHover()` of the `SearchContainerTag` class.
- The attributes `classNameHover`, `hover`, `rowClassNameAlternate`, `rowClassNameAlternateHover`, `rowClassNameBody`, `rowClassNameBodyHover` of the `liferay-search-container` JavaScript module.

Who is affected? This affects developers that use the `hover` attribute of the `liferay-ui:search-container` tag.

How should I update my code? You should update your code changing the CSS selector that defines how rows look on hover to use the `:hover` and `:nth-of-type` CSS pseudo selectors instead.

Why was this change made? Browsers support better ways to style content on hover in a way that doesn't penalize performance. Therefore, this change was made to increase the performance of hovering over content in Liferay.

Removed AppViewMove and AppViewSelect JavaScript Modules

- **Date:** 2015-Nov-03
- **JIRA Ticket:** LPS-58854

What changed? The JavaScript modules `AppViewMove` and `AppViewSelect` have been removed.

Who is affected? This affects developers that use these modules to configure *select* and *move* actions inside their applications.

How should I update my code? If you are using any of these modules, you can make use of the following `SearchContainer` APIs:

- Listen to the `rowToggled` event of the search container to be notified about changes to the search container state.
- Configure your search container *move* options creating a `RowMover` and define the allowed *move* targets and associated actions.
- Use the `registerAction` method of the search container to execute your *move* logic when the user completes a *move* action.

Why was this change made? The removed JavaScript modules contained too much logic and were difficult to decipher. It was also difficult to add this to an existing app. With this change, every app using a search container can use this functionality much easier.

Removed the mergeLayoutTags Preference from Asset Publisher

- **Date:** 2015-Nov-20
- **JIRA Ticket:** LPS-60677

What changed? The `mergeLayoutTags` preference has been removed from the Asset Publisher.

Who is affected? This affects any Asset Publisher portlet that uses this preference.

How should I update my code? There is nothing to update since this functionality is no longer used.

Why was this change made? In previous versions of Liferay, some applications such as Blogs and Wiki shared the tags of their entries within the page. The Asset Publisher was able to use them to show other assets with the same tags. This functionality has changed, so the preference is no longer used.

Removed the liferay-ui:navigation Tag and Replaced with liferay-site-navigation:navigation Tag

- **Date:** 2015-Nov-20
- **JIRA Ticket:** LPS-60328

What changed? The `liferay-ui:navigation` tag has been removed and replaced with the `liferay-site-navigation:navigation` tag.

Who is affected? Plugins or templates that are using the `liferay-ui:navigation` tag need to update their usage of the tag.

How should I update my code? You should import the `liferay-site-navigation` tag library (if necessary) and update the tag namespace from `liferay-ui:navigation` to `liferay-site-navigation:navigation`.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Removed Software Catalog Portlet and Services

- **Date:** 2015-Nov-21
- **JIRA Ticket:** LPS-60705

What changed? The Software Catalog portlet and its associated services are no longer part of Liferay's source code or binaries.

Who is affected? This affects portals which were making use of the Software Catalog portlet to manage a catalog of their software. Developers who were making use of the software catalog services from their custom code are also affected.

How should I update my code? There is no direct replacement for invocations to the Software Catalog services. In cases where it is really needed, it is possible to obtain the code from a previous release and include it in the custom product (subject to licensing).

Why was this change made? The Software Catalog was developed to implement the very first versions of what later become Liferay's Marketplace. It was later replaced and has not been used by Liferay since then. It has also been used minimally outside of Liferay. The decision was made to remove it so Liferay could be more lightweight and free time to focus on other areas of the product that add more value.

Removed the liferay-ui:asset-categories-navigation Tag and Replaced with liferay-asset:asset-categories-navigation

- **Date:** 2015-Nov-25
- **JIRA Ticket:** LPS-60753

What changed? The `liferay-ui:asset-categories-navigation` tag has been removed and replaced with the `liferay-asset:asset-categories-navigation` tag.

Who is affected? Plugins or templates that are using the `liferay-ui:asset-categories-navigation` tag need to update their usage of the tag.

How should I update my code? You should import the `liferay-asset` tag library (if necessary) and update the tag namespace from `liferay-ui:asset-categories-navigation` to `liferay-asset:asset-categories-navigation`.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Removed the liferay-ui:trash-empty Tag and Replaced with liferay-trash:empty

- **Date:** 2015-Nov-30
- **JIRA Ticket:** LPS-60779

What changed? The `liferay-ui:trash-empty` tag has been removed and replaced with the `liferay-trash:empty` tag.

Who is affected? Plugins and templates that are using the `liferay-ui:trash-empty` tag need to update their usage of the tag.

How should I update my code? You should import the `liferay-trash` tag library (if necessary) and update the tag namespace from `liferay-ui:trash-empty` to `liferay-trash:empty`.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Removed the liferay-ui:trash-undo Tag and Replaced with liferay-trash:undo

- **Date:** 2015-Nov-30
- **JIRA Ticket:** LPS-60779

What changed? The `liferay-ui:trash-undo` taglib has been removed and replaced with the `liferay-trash:undo` tag.

Who is affected? Plugins and templates that are using the `liferay-ui:trash-undo` tag need to update their usage of the tag.

How should I update my code? You should import the `liferay-trash` tag library (if necessary) and update the tag namespace from `liferay-ui:trash-undo` to `liferay-trash:undo`.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Removed the getPageOrderByComparator Method from WikiUtil

- **Date:** 2015-Dec-01
- **JIRA Ticket:** LPS-60843

What changed? The `getPageOrderByComparator` method has been removed from `WikiUtil`.

Who is affected? This affects developers that use this method in their code.

How should I update my code? You should update your code to invoke `WikiPortletUtil.getPageOrderByComparator(String, String)`.

Why was this change made? As part of the modularization efforts it has been considered that that this logic belongs to wiki-web module.

Custom AUI Validators Are No Longer Implicitly Required

- **Date:** 2015-Dec-02
- **JIRA Ticket:** LPS-60995

What changed? The AUI Validator tag no longer forces custom validators (e.g., `name="custom"`) to be required, and are now optional by default.

Who is affected? This affects developers using custom validators, especially ones who relied on the field being implicitly required via the custom validator.

How should I update my code? There are several cases where you should update your code to compensate for this change. First, blank value checking is no longer necessary, so places where blank values are checked should be updated.

Old Code:

```
<aui:input name="privateVirtualHost">
    <aui:validator errorMessage="please-enter-a-unique-virtual-host" name="custom">
        function(val, fieldNode, ruleValue) {
            return !val || val != A.one('#<portlet:namespace />publicVirtualHost').val();
        }
    </aui:validator>
</aui:input>
```

New Code:

```
<aui:input name="privateVirtualHost">
    <aui:validator errorMessage="please-enter-a-unique-virtual-host" name="custom">
        function(val, fieldNode, ruleValue) {
            return val != A.one('#<portlet:namespace />publicVirtualHost').val();
        }
    </aui:validator>
</aui:input>
```

Also, instead of using custom validators to determine if a field is required, you should now use a conditional `required` validator.

Old Code:

```
<aui:input name="file" type="file" />

<aui:input name="title">
    <aui:validator errorMessage="you-must-specify-a-file-or-a-title" name="custom">
        function(val, fieldNode, ruleValue) {
            return !!val || !!A.one('#<portlet:namespace />file').val();
        }
    </aui:validator>
</aui:input>
```

New Code:

```
<aui:input name="file" type="file" />

<aui:input name="title">
    <aui:validator errorMessage="you-must-specify-a-file-or-a-title" name="required">
        function(fieldNode) {
            return !A.one('#<portlet:namespace />file').val();
        }
    </aui:validator>
</aui:input>
```

Lastly, custom validators that assumed validation would always run must now explicitly pass the `required` validator. This is done by passing in the `<aui:validator name="required" />` element. The `<aui:input>` tag listed below is an example of how to explicity pass the `required` validator:

```
<aui:input name="vowelsOnly">
    <aui:validator errorMessage="must-contain-only-the-following-characters" name="custom">
        function(val, fieldNode, ruleValue) {
            var allowedCharacters = 'aeiouy';
            var regex = new RegExp('[^' + allowedCharacters + ']');

            return !regex.test(val);
        }
    </aui:validator>
    <aui:validator name="required" />
</aui:input>
```

Why was this change made? A custom validator caused the field to be implicitly required. This meant that all validators for the field would be evaluated. This created a condition where you could not combine custom validators with another validator for an optional field.

For example, imagine an optional field which has an email validator, plus a custom validator which checks for email addresses within a specific domain (e.g., `example.com`). There was no way for this optional field to pass validation. Even if you handled blank values in your custom validator, that blank value would fail the email validator.

This change requires most custom validators to be refactored, but allows greater flexibility for all developers.

Moved Recycle Bin Logic Into a New DLTrashService Interface

- **Date:** 2015-Dec-02
- **JIRA Ticket:** LPS-60810

What changed? All Recycle Bin logic in Documents and Media services was moved from `DLAppService` into the new `DLTrashService` service interface. All moved methods have the same name and signatures.

Who is affected? This affects any local or remote caller of `DLAppService`.

How should I update my code? As all methods have been simply moved into the new service, calling the equivalent method on `DLTrashService` suffices.

Why was this change made? Documents and Media services have complex interdependencies that result in circular dependencies. Until now, `DLAppService` was responsible for exposing the Recycle Bin logic, delegating it to other components. The problem was, the components depended on `DLAppService` to implement their logic. Extracting the services from `DLAppService` was the only sensible solution to this circularity.

Deprecated the liferay-ui:flags Tag and Replaced with liferay-flags:flags

- **Date:** 2015-Dec-02
- **JIRA Ticket:** LPS-60967

What changed? The `liferay-ui:flags` tag has been deprecated and replaced with the `liferay-flags:flags` tag.

Who is affected? Plugins or templates that are using the `liferay-ui:flags` tag need to update their usage of the tag.

How should I update my code? You should import the `liferay-flags` tag library (if necessary) and update the tag namespace from `liferay-ui:flags` to `liferay-flags:flags`.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Removed the liferay-ui:diff Tag and Replaced with liferay-frontend:diff

- **Date:** 2015-Dec-14
- **JIRA Ticket:** LPS-61326

What changed? The `liferay-ui:diff` tag has been removed and replaced with the `liferay-frontend:diff` tag.

Who is affected? Plugins and templates that are using the `liferay-ui:diff` tag need to update their usage of the tag.

How should I update my code? You should import the `liferay-frontend` tag library (if necessary) and update the tag namespace from `liferay-ui:diff` to `liferay-frontend:diff`.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Taglibs Are No Longer Accessible via the theme Variable in FreeMarker

- **Date:** 2016-Jan-06
- **JIRA Ticket:** LPS-61683

What changed? The `${theme}` variable previously injected in the FreeMarker context providing access to various tags and utilities is no longer available.

Who is affected? This affects FreeMarker templates that are using the `${theme}` variable.

How should I update my code? All the tags and utility methods formerly accessed via the `${theme}` variable should now be accessed directly via tags.

Example 1

```
${theme.runtime("com.liferay.portal.kernel.servlet.taglib.ui.BreadcrumbEntry", portletProviderAction.VIEW, "", default_preferences)}
```

can be replaced by:

```
<@liferay_portlet["runtime"]
    defaultPreferences=default_preferences
    portletProviderAction=portletProviderAction.VIEW
    portletProviderClassName="com.liferay.portal.kernel.servlet.taglib.ui.BreadcrumbEntry"
/>
```

Example 2

```
${theme.include(content_include)}
```

can be replaced by:

```
<@liferay_util["include"] page=content_include />
```

Example 3

```
${theme.wrapPortlet("portlet.ftl", content_include)}
```

can be replaced by:

```
<@liferay_theme["wrap-portlet"] page="portlet.ftl">
    <@liferay_util["include"] page=content_include />
</@>
```

Example 4

```
${theme.iconHelp(portlet_description)}
```

can be replaced by:

```
<@liferay_ui["icon-help"] message=portlet_description />
```

Example 5

```
${nav_item.icon()}
```

can be replaced by:

```
<@liferay_theme["layout-icon"] layout=${nav_item.getLayout()} />
```

Why was this change made? Previously, the `{$theme}` variable was being injected with the `VelocityTaglibImpl` class. This created coupling between template engines and coupling between specific tags and template engines at the same time.

FreeMarker already offers native support for tags which cover all the functionality originally provided by the `{$theme}` variable. Removing this coupling helps future development while still keeping all the existing functionality.

Portlet Configuration Options May Not Always Be Displayed

- **Date:** 2016-Jan-07
- **JIRA Ticket:** LPS-54620 and LPS-61820

What changed? The portlet configuration options (e.g., configuration, export/import, look and feel, etc.) were always displayed in every view of the portlet and couldn't be customized.

With Lexicon, the configuration options displayed are based on the portlet's context, so not all options will always be displayed.

Who is affected? This affects portlets that should always display all configuration options no matter which view of the portlet is rendered.

How should I update my code? If you don't apply any change to your source code, you will experience the following behaviors based on the portlet type:

- **Struts Portlet:** If you've defined a `view-action` init parameter, the configuration options are only displayed for that particular view when invoking a URL with a parameter `struts_action` with the value indicated in the `view-action` init parameter and also in the default view of the portlet (when there is no `struts_action` parameter in the request).

- **Liferay MVC Portlet:** If you've defined a `view-template` init parameter, the configuration options are only displayed when that template is rendered by invoking a URL with a parameter `mvcPath` with the value indicated in the `view-template` init parameter. and also in the default view of the portlet (when there is no `mvcPath` parameter in the request).
- If it's a portlet using any other framework, the configuration options are always displayed.

In order to keep the old behavior of adding the configuration options in every view, you need to add the init parameter `always-display-default-configuration-icons` with the value `true`.

Why was this change made? Lexicon patterns require the ability to specify different configuration options depending on the view of the portlet by adding or removing options. This can be easily achieved by using the `PortletConfigurationIcon` classes.

The getURLView Method of AssetRenderer Returns String Instead of PortletURL

- **Date:** 2016-Jan-08
- **JIRA Ticket:** LPS-61853

What changed? The `AssetRenderer` interface's `getURLView` method has changed and now returns `String` instead of `PortletURL`.

Who is affected? This affects all custom assets that implement the `AssetRenderer` interface.

How should I update my code? You should update the method signature to reflect that it returns a `String` and you should adapt your implementation accordingly.
In general, it should be as easy as returning `portletURL.toString()`.

Why was this change made? The API was forcing implementations to return a `PortletURL`, making it difficult to return another type of link. For example, in the case of Bookmarks, developers wanted to automatically redirect to other potential URLs.

Removed the icon Method from NavItem

- **Date:** 2016-Jan-11
- **JIRA Ticket:** LPS-61900

What changed? The `NavItem` interface has changed and the method `icon` that would render the nav item icon has been removed.

Who is affected? This affects all themes using the `nav_item.icon()` method.

How should I update my code? You should update your code to call the method `nav_item.iconURL` to return the image's URL and then use it as you prefer.
Example:

```
<img alt="Page Icon" class="layout-logo" src="<%= nav_item.iconURL()" />
```

To keep the previous behavior in Velocity:

```
$theme.layoutIcon($nav_item.getLayout())
```

To keep the previous behavior in FreeMarker:

```
<@liferay_theme["layout-icon"] layout=nav_item_layout />
```

Why was this change made? The API was forcing developers to have a dependency on a taglib, which didn't allow for much flexibility.

Renamed Packages to Fix the Split Packages Problem

- **Date:** 2016-Jan-19
- **JIRA Ticket:** LPS-61952

What changed? Split packages are caused when two or more bundles export the same package name and version. When the classloader loads a package, exactly one exporter of that package is chosen; so if a package is split across multiple bundles, then an importer only sees a subset of the package.

Who is affected? The `portal-kernel` and `portal-impl` folders have many packages with the same name. Therefore, all of these packages are affected by the split package problem.

How should I update my code? You should rename duplicated package names if they currently exist somewhere else.

Example

- `com.liferay.counter` → `com.liferay.counter.kernel`
- `com.liferay.mail.model` → `com.liferay.mail.kernel.model`
- `com.liferay.mail.service` → `com.liferay.mail.kernel.service`
- `com.liferay.mail.util` → `com.liferay.mail.kernel.util`
- `com.liferay.portal.exception` → `com.liferay.portal.kernel.exception`
- `com.liferay.portal.jdbc.pool.metrics` → `com.liferay.portal.kernel.jdbc.pool.metrics`
- `com.liferay.portal.kernel.mail` → `com.liferay.mail.kernel.model`
- `com.liferay.portal.layoutconfiguration.util` → `com.liferay.portal.kernel.layoutconfiguration.util`
- `com.liferay.portal.layoutconfiguration.util.xml` → `com.liferay.portal.kernel.layoutconfiguration.util.xml`
- `com.liferay.portal.mail` → `com.liferay.portal.kernel.mail`
- `com.liferay.portal.model` → `com.liferay.portal.kernel.model`
- `com.liferay.portal.model.adapter` → `com.liferay.portal.kernel.model.adapter`
- `com.liferay.portal.model.impl` → `com.liferay.portal.kernel.model.impl`
- `com.liferay.portal.portletfilerepository` → `com.liferay.portal.kernel.portletfilerepository`

- com.liferay.portal.repository.proxy → com.liferay.portal.kernel.repository.proxy
- com.liferay.portal.security.auth → com.liferay.portal.kernel.security.auth
- com.liferay.portal.security.exportimport → com.liferay.portal.kernel.security.exportimport
- com.liferay.portal.security.ldap → com.liferay.portal.kernel.security.ldap
- com.liferay.portal.security.membershippolicy → com.liferay.portal.kernel.security.membershippolicy
- com.liferay.portal.security.permission → com.liferay.portal.kernel.security.permission
- com.liferay.portal.security.permission.comparator → com.liferay.portal.kernel.security.permission.comparator
- com.liferay.portal.security.pwd → com.liferay.portal.kernel.security.pwd
- com.liferay.portal.security.xml → com.liferay.portal.kernel.security.xml
- com.liferay.portal.service.configuration → com.liferay.portal.kernel.service.configuration
- com.liferay.portal.service.http → com.liferay.portal.kernel.service.http
- com.liferay.portal.service.permission → com.liferay.portal.kernel.service.permission
- com.liferay.portal.service.persistence.impl → com.liferay.portal.kernel.service.persistence.impl
- com.liferay.portal.theme → com.liferay.portal.kernel.theme
- com.liferay.portal.util → com.liferay.portal.kernel.util
- com.liferay.portal.util.comparator → com.liferay.portal.kernel.util.comparator
- com.liferay.portal.verify.model → com.liferay.portal.kernel.verify.model
- com.liferay.portal.webserver → com.liferay.portal.kernel.webserver
- com.liferay.portlet → com.liferay.portal.kernel.portlet
- com.liferay.portlet.admin.util → com.liferay.admin.kernel.util
- com.liferay.portlet.announcements → com.liferay.announcements.kernel
- com.liferay.portlet.asset → com.liferay.asset.kernel
- com.liferay.portlet.backgroundtask.util.comparator → com.liferay.background.task.kernel.util.comparator
- com.liferay.portlet.blogs → com.liferay.blogs.kernel
- com.liferay.portlet.blogs.exception → com.liferay.blogs.kernel.exception
- com.liferay.portlet.blogs.model → com.liferay.blogs.kernel.model
- com.liferay.portlet.blogs.service → com.liferay.blogs.kernel.service
- com.liferay.portlet.blogs.service.persistence → com.liferay.blogs.service.persistence
- com.liferay.portlet.blogs.util.comparator → com.liferay.blogs.kernel.util.comparator

- `com.liferay.portlet.documentlibrary` → `com.liferay.document.library.kernel`
- `com.liferay.portlet.dynamicdatamapping` → `com.liferay.dynamic.data.mapping.kernel`
- `com.liferay.portlet.expando` → `com.liferay.expando.kernel`
- `com.liferay.portlet.exportimport` → `com.liferay.exportimport.kernel`
- `com.liferay.portlet.imagegallerydisplay.display.context` → `com.liferay.image.gallery.display.kernel.display.co`
- `com.liferay.portlet.journal.util` → `com.liferay.journal.kernel.util`
- `com.liferay.portlet.layoutsadmin.util` → `com.liferay.layouts.admin.kernel.util`
- `com.liferay.portlet.messageboards` → `com.liferay.message.boards.kernel`
- `com.liferay.portlet.messageboards.constants` → `com.liferay.message.boards.kernel.constants`
- `com.liferay.portlet.messageboards.exception` → `com.liferay.message.boards.kernel.exception`
- `com.liferay.portlet.messageboards.model` → `com.liferay.message.boards.kernel.model`
- `com.liferay.portlet.messageboards.service` → `com.liferay.message.boards.kernel.service`
- `com.liferay.portlet.messageboards.service.persistence` → `com.liferay.message.boards.kernel.service.persistence`
- `com.liferay.portlet.messageboards.util` → `com.liferay.message.boards.kernel.util`
- `com.liferay.portlet.messageboards.util.comparator` → `com.liferay.message.boards.kernel.util.comparator`
- `com.liferay.portlet.mobiledevicerules` → `com.liferay.mobile.device.rules`
- `com.liferay.portlet.portletconfiguration.util` → `com.liferay.portlet.configuration.kernel.util`
- `com.liferay.portlet.rolesadmin.util` → `com.liferay.roles.admin.kernel.util`
- `com.liferay.portlet.sites.util` → `com.liferay.sites.kernel.util`
- `com.liferay.portlet.social` → `com.liferay.social.kernel`
- `com.liferay.portlet.trash` → `com.liferay.trash.kernel`
- `com.liferay.portlet.useradmin.util` → `com.liferay.users.admin.kernel.util`
- `com.liferay.portlet.ratings` → `com.liferay.ratings.kernel`
- `com.liferay.portlet.ratings.definition` → `com.liferay.ratings.kernel.definition`
- `com.liferay.portlet.ratings.display.context` → `com.liferay.ratings.kernel.display.context`
- `com.liferay.portlet.ratings.exception` → `com.liferay.ratings.kernel.exception`
- `com.liferay.portlet.ratings.model` → `com.liferay.ratings.kernel.model`
- `com.liferay.portlet.ratings.service` → `com.liferay.ratings.kernel.service`
- `com.liferay.portlet.ratings.service.persistence` → `com.liferay.ratings.kernel.service.persistence`
- `com.liferay.portlet.ratings.transformer` → `com.liferay.ratings.kernel.transformer`

Why was this change made? This change was necessary to solve the current split package problems and prevent future ones.

Removed the aui:column Tag and Replaced with aui:col

- **Date:** 2016-Jan-19
- **JIRA Ticket:** LPS-62208

What changed? The `aui:column` tag has been removed and replaced with the `aui:col` tag.

Who is affected? Plugins or templates that are using the `aui:column` tag must update their usage of the tag.

How should I update my code? You should import the `aui` tag library (if necessary) and update the tag namespace from `aui:column` to `aui:col`.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

The title Field of FileEntry Models is Now Mandatory

- **Date:** 2016-Jan-25
- **JIRA Ticket:** LPS-62251

What changed? The `title` field of file entries was optional as long as a source file name was provided. To avoid confusion, the title is now required by the API and is filled automatically by the UI when a source file name is present.

Who is affected? This affects any user of the local or remote API. Users of the Web UI are unaffected.

How should I update my code? You should pass a non-null, non-empty string for the `title` parameter of the `addFileEntry` and `updateFileEntry` methods.

Why was this change made? The `title` field was marked as mandatory, but it was possible to create a document without filling it, as the backend would infer a value from the source file name automatically. This was considered confusing from a UX perspective.

DLUtil.getImagePreviewURL and DLUtil.getThumbnailSrc Can Return Empty Strings

- **Date:** 2016-Jan-28
- **JIRA Ticket:** LPS-62643

What changed? The `DLUtil.getImagePreviewURL` and `DLUtil.getThumbnailSrc` methods return an empty string if there are no previews or thumbnails for the specific image, video, or document.

Previously, if there were no previews or thumbnails, these methods would return a URL to an image based on the document.

Who is affected? This affects any developer invoking `DLUtil.getImagePreviewURL` or `DLUtil.getThumbnailSrc`.

How should I update my code? You should be aware that the method could return an empty string and act accordingly. For example, you could display the `documents-and-media` Lexicon icon instead.

Why was this change made? In order to display the `documents-and-media` Lexicon icon in Documents and Media, this change was necessary.

Removed the aui:button-item Tag and Replaced with aui:button

- **Date:** 2016-Feb-04
- **JIRA Ticket:** LPS-62922

What changed? The `aui:button-item` tag has been removed and replaced with the `aui:button` tag.

Who is affected? Plugins or templates that are using the `aui:button-item` tag must update their usage of the tag.

How should I update my code? You should import the `aui` tag library (if necessary) and update the tag namespace from `aui:button-item` to `aui:button`.

Why was this change made? This change was made as a part of the ongoing strategy to remove deprecated code.

Removed the WAP Functionality

- **Date:** 2016-Feb-05
- **JIRA Ticket:** LPS-62920

What changed? The WAP functionality has been removed.

Who is affected? This affects developers that use the WAP functionality.

How should I update my code? If you are using any of the following methods, you need to remove the parameters in those methods related to WAP.

- `LayoutLocalServiceUtil.updateLookAndFeel`
- `LayoutRevisionLocalServiceUtil.addLayoutRevision`
- `LayoutRevisionLocalServiceUtil.updateLayoutRevision`
- `LayoutRevisionServiceUtil.addLayoutRevision`
- `LayoutServiceUtil.updateLookAndFeel`
- `LayoutSetLocalServiceUtil.updateLookAndFeel`
- `LayoutSetServiceUtil.updateLookAndFeel`
- `ThemeLocalServiceUtil.getColorScheme`
- `ThemeLocalServiceUtil.getControlPanelThemes`
- `ThemeLocalServiceUtil.getPageThemes`
- `ThemeLocalServiceUtil.getTheme`

Why was this change made? This change was made because WAP is an obsolete functionality.

Removed the aui:layout Tag with No Direct Replacement

- **Date:** 2016-Feb-08
- **JIRA Ticket:** LPS-62935

What changed? The `aui:layout` tag has been removed with no direct replacement.

Who is affected? Plugins or templates that are using the `aui:layout` tag must remove their usage of the tag.

How should I update my code? There is no direct replacement. You should remove all usages of the `aui:layout` tag.

Why was this change made? This change was made as a part of the ongoing strategy to remove deprecated tags.

Deprecated the liferay-portlet:icon-back Tag with No Direct Replacement

- **Date:** 2016-Feb-10
- **JIRA Ticket:** LPS-63101

What changed? The `liferay-portlet:icon-back` tag has been deprecated with no direct replacement.

Who is affected? Plugins or templates that are using the `liferay-portlet:icon-back` tag must remove their usage of the tag.

How should I update my code? There is no direct replacement. You should remove all usages of the `liferay-portlet:icon-back` tag.

Why was this change made? This change was made as a part of the ongoing strategy to deprecate unused tags.

Deprecated the liferay-security:encrypt Tag with No Direct Replacement

- **Date:** 2016-Feb-10
- **JIRA Ticket:** LPS-63106

What changed? The `liferay-security:encrypt` tag has been deprecated with no direct replacement.

Who is affected? Plugins or templates that are using the `liferay-security:encrypt` tag must remove their usage of the tag.

How should I update my code? There is no direct replacement. You should remove all usages of the `liferay-security:encrypt` tag.

Why was this change made? This change was made as a part of the ongoing strategy to deprecate unused tags.

Removed the Ability to Specify Class Loaders in Scripting

- **Date:** 2016-Feb-17
- **JIRA Ticket:** LPS-63180

What changed?

- `com.liferay.portal.kernel.scripting.ScriptingExecutor` no longer uses the provided class loaders in the eval methods.
- `com.liferay.portal.kernel.scripting.Scripting` no longer uses the provided class loaders and servlet context names in eval and exec methods.

Who is affected?

- All implementations of `com.liferay.portal.kernel.scripting.ScriptingExecutor` are affected.
- All classes that call `com.liferay.portal.kernel.scripting.Scripting` are affected.

How should I update my code? You should remove class loader and servlect context parameters from calls to the modified methods.

Why was this change made? This change was made since custom class loader management is no longer necessary in the OSGi container.

User Operation and Importer/Exporter Classes and Utilities Have Been Moved or Removed From portal-kernel

- **Date:** 2016-Feb-17
- **JIRA Ticket:** LPS-63205

What changed?

- `com.liferay.portal.kernel.security.exportimport.UserImporter`, `com.liferay.portal.kernel.security.exportimport` and `com.liferay.portal.kernel.security.exportimport.UserOperation` have been moved from portal-kernel to the portal-security-export-import-api module.

- `com.liferay.portal.kernel.security.exportimport.UserImporterUtil` and `com.liferay.portal.kernel.security.expor` have been removed with no replacement.

Who is affected?

- All implementations of `com.liferay.portal.kernel.security.exportimport.UserImporter` or `com.liferay.portal.kernel.security.exportimport.UserExporter` are affected.

- All code that uses `com.liferay.portal.kernel.security.exportimport.UserImporterUtil`, `com.liferay.portal.kernel.se` `com.liferay.portal.kernel.security.exportimport.UserImporter`, or `com.liferay.portal.kernel.security.exportimpo` is affected.

How should I update my code? If you are in an OSGi module, you can simply inject the UserImporter or UserExporter references

```
@Reference
private UserExporter_userExporter;

@Reference
private UserImporter _userImporter;
```

If you are in a legacy WAR or WAB, you will need a snippet like:

```
Bundle bundle = FrameworkUtil.getBundle(getClass());

BundleContext bundleContext = bundle.getBundleContext();

ServiceReference<UserImporter> serviceReference =
    bundleContext.getServiceReference(UserImporter.class);

UserImporter userImporter = bundleContext.getService(serviceReference);
```

Why was this change made? The change was made to improve modularity of the user import/export subsystem in the product.

Deprecated Category Entry for Users

- **Date:** 2016-Feb-22
- **JIRA Ticket:** LPS-63466

What changed? The category entry for Site Administration → Users has been deprecated in favor of Site Administration → Members.

Who is affected? All developers who specified a `control-panel-entry-category` to be visible in Site Administration → Users are affected.

How should I update my code? You should change the entry from `site_administration.users` to `site_administration.members` to make it visible in the category.

Why was this change made? This change standardizes naming conventions and separates concepts between Users in the Control Panel and Site Members.

Deprecated Category Entry for Pages

- **Date:** 2016-Feb-25
- **JIRA Ticket:** LPS-63667

What changed? The category entry for Site Administration → Pages has been deprecated in favor of Site Administration → Navigation.

Who is affected? All developers who specified a `control-panel-entry-category` to be visible in Site Administration → Pages are affected.

How should I update my code? You should change the entry from `site_administration.pages` to `site_administration.navigation` to make it visible in the category.

Why was this change made? This change standardizes naming conventions and separates concepts in Product Menu

Removed the com.liferay.dynamic.data.mapping.util.DDMXMLUtil Class

- **Date:** 2016-Mar-03
- **JIRA Ticket:** LPS-63928

What changed? The class `com.liferay.dynamic.data.mapping.util.DDMXMLUtil` has been removed with no replacement.

Who is affected? All code that uses `com.liferay.dynamic.data.mapping.util.DDMXMLUtil` is affected.

How should I update my code? In an OSGi module, simply inject the DDMXML reference:

```
@Reference
private DDMXML _ddmXML;
```

In a legacy WAR or WAB, you need to get a DDMXML service reference from the bundle context:

```
Bundle bundle = FrameworkUtil.getBundle(getClass());

BundleContext bundleContext = bundle.getBundleContext();

ServiceReference<UserImporter> serviceReference =
    bundleContext.getServiceReference(DDMXML.class);

DDMXML ddmXML = bundleContext.getService(serviceReference);
```

Why was this change made? This change was made to improve modularity of the dynamic data mapping subsystem.

FlagsEntryService.addEntry Method Throws PortalException

- **Date:** 2016-Mar-04
- **JIRA Ticket:** LPS-63109

What changed? The method `FlagsEntryService.addEntry` now throws a `PortalException` if the `reporterEmailAddress` is not a valid email address.

Who is affected? Any caller of the method `FlagsEntryService.addEntry` is affected.

How should I update my code? You should consider checking for the `PortalException` in try-catch blocks and adapt your code accordingly.

Why was this change made? This change prevents providing an incorrect email address when adding flag entries.

Removed PHP Portlet Support

- **Date:** 2016-Mar-10
- **JIRA Ticket:** LPS-64052

What changed? PHP portlets are no longer supported.

Who is affected? This affects any portlet using the class `com.liferay.util.bridges.php.PHPPortlet`.

How should I update my code? You should port your PHP portlet to a different technology.

Why was this change made? This change simplifies future maintenance of the portal. This support could be added back in the future as an independent module.

Removed Liferay Frontend Editor BBCode Web, Previously Known as Liferay BBCode Editor

- **Date:** 2016-Mar-16
- **JIRA Ticket:** LPS-48334

What changed? The following things have been changed:

- Removed the `com.liferay.frontend.editor.bbcode.web` OSGi bundle
- Removed all hardcoded references/logic for the editor
- Added a log warning and logic to upgrade the editor property to `ckeditor_bbcode` if the old `bbcode` is being used. This log warning and logic will be removed in the future, along with LPS-64099.

Who is affected? This affects anyone who has the property `editor.wysiwyg.portal-web.docroot.html.portlet.message_board` set to `bbcode` in portal properties (e.g., `portal-ext.properties`).

How should I update my code? You should modify your `portal-ext.properties` file to remove the property `editor.wysiwyg.portal-web.docroot.html.portlet.message_boards.edit_message.bb_code.jsp`.

Why was this change made? Since Liferay Frontend Editor BBCode Web has been deprecated since 6.1, it was time to remove it completely. This frees up development and support resources to focus on supported features.

Removed the asset.entry.validator Property

- **Date:** 2016-Mar-17
- **JIRA Ticket:** LPS-64370

What changed? The property `asset.entry.validator` has been removed from `portal.properties`.

Who is affected? This affects any installation with a customized asset validator.

How should I update my code? You should create a new OSGi component that implements `AssetEntryValidator` and define for which models it will be applicable by using the `model.class.name` OSGi property, or an asterisk if it applies to any model.

If you were using the `MinimalAssetEntryValidator`, this functionality can still be added by deploying the module `asset-tags-validator`.

Why was this change made? This change has been made as part of the modularization efforts to decouple different parts of the portal.

Removed the swfupload and video_player Utilities

- **Date:** 2016-May-13
- **JIRA Ticket:** LPS-54111

What changed? The utilities `swfupload` and `video_player` have been removed.

Who is affected? This affects anyone who is using the `swfupload` AlloyUI module or any of the associated `swfupload_f*.swf` and `mpw_player.swf` flash movies.

How should I update my code? There are better, more standard ways to achieve upload currently. For instance, you can use A.Uploader to manage your uploads consistently across browsers.

For audio/video reproduction, you should update your code to use A.Audio and A.Video.

Why was this change made? This change removes outdated code no longer being used in the platform. In addition, this change avoids future security issues from outdated flash movies.

Moved Journal Portlet Properties to OSGi Configuration

- **Date:** 2016-Jul-29
- **JIRA Ticket:** LPS-58672

What changed? All Journal portlet properties have been moved to an OSGi configuration.

Who is affected? This affects anyone who is overriding the Journal portlet's `portlet.properties` file.

How should I update my code? Instead of overriding the Journal portlet's `portlet.properties` file, you can manage the properties from Portal's configuration administrator. This can be accessed by navigating to Liferay's Control Panel → *System Settings* → *Web Experience* and selecting the appropriate Web Content category.

Why was this change made? This change was made as part of modularization efforts to ease portlet configuration changes.

Moved the liferay-ui:journal-article Tag to Journal

- **Date:** 2016-Nov-24
- **JIRA Ticket:** LPS-69321

What changed? The `liferay-ui:journal-article` tag has been moved to the Journal (Web Content) application.

Who is affected? This affects developers using the `liferay-ui:journal-article` tag.

How should I update my code? You should use the `liferay-journal:journal-article` tag instead.

Example

Old code:

```
<liferay-ui:journal-article
    articleId="<%= article.getArticleId() %>"
/>
```

New code:

```
<liferay-journal:journal-article
    articleId="<%= article.getArticleId() %>"
    groupId="<%= article.getGroupId() %>"
/>
```

If you still want to use the `liferay-ui:journal-article` tag, you must deploy the `journal-taglib` module to your Liferay installation.

Why was this change made? This change was made as part of the modularization efforts for the Web Content application.

Deprecated the liferay-ui:captcha Tag and Replaced with liferay-captcha:captcha

- **Date:** 2016-Nov-29
- **JIRA Ticket:** LPS-69383

What changed? The `liferay-ui:captcha` tag has been deprecated and replaced with the `liferay-captcha:captcha` tag.

Who is affected? Plugins or templates that are using the `liferay-ui:captcha` tag need to update their usage of the tag.

How should I update my code? You should import the `liferay-captcha` tag library (if necessary) and update the tag namespace from `liferay-ui:captcha` to `liferay-captcha:captcha`.

Why was this change made? This change was made as a part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Moved Shopping File Uploads Portlet Properties to OSGi Configuration

- **Date:** 2016-Dec-08
- **JIRA Ticket:** LPS-69210

What changed? The Shopping file uploads portlet properties have been moved from Server Administration to an OSGi configuration named `ShoppingFileUploadsConfiguration.java` in the `shopping-api` module.

Who is affected? This affects anyone who is using the following portlet properties:

- `shopping.image.extensions`
- `shopping.image.large.max.size`
- `shopping.image.medium.max.size`
- `shopping.image.small.max.size`

How should I update my code? Instead of overriding the `portal.properties` file, you can manage the properties from Portal's configuration administrator. This can be accessed by navigating to Liferay's *Control Panel → Configuration → System Settings → Shopping Cart Images* and editing the settings there.

If you would like to include the new configuration in your application, follow the instructions for making your applications configurable in Liferay 7.0.

Why was this change made? This change was made as part of the modularization efforts to ease portal configuration changes.

Moved the Expando Custom Field Tags to liferay-expando Taglib

- **Date:** 2016-Dec-12
- **JIRA Ticket:** LPS-69400

What changed? The following tags have been deprecated and replaced:

- `liferay-ui:custom-attribute`
- `liferay-ui:custom-attribute-list`
- `liferay-ui:custom-attributes-available`

Who is affected? Plugins and templates that are using the aforementioned tags must update their usage of the tag.

How should I update my code? You should import the `liferay-expando` tag library (if necessary) and update the tag namespace from `liferay-ui` to `liferay-expando`:

- `liferay-ui:custom-attribute` → `liferay-expando:custom-attribute`
- `liferay-ui:custom-attribute-list` → `liferay-expando:custom-attribute-list`
- `liferay-ui:custom-attributes-available` → `liferay-expando:custom-attributes-available`

Why was this change made? This change was made as part of the ongoing strategy to modularize Liferay Portal by means of an OSGi container.

Moved Journal File Uploads Portlet Properties to OSGi Configuration

- **Date:** 2017-Jan-04
- **JIRA Ticket:** LPS-69209

What changed? The Journal File Uploads portlet properties have been moved from Server Administration to an OSGi configuration named `JournalFileUploadsConfiguration.java` in the `journal-service` module.

Who is affected? This affects anyone who is using the following portlet properties:

- `journal.image.extensions`
- `journal.image.small.max.size`

How should I update my code? Instead of overriding the `portal.properties` file, you can manage the properties from Portal's configuration administrator. This can be accessed by navigating to Liferay's *Control Panel → Configuration → System Settings → Web Content File Uploads* and editing the settings there.

If you would like to include the new configuration in your application, follow the instructions for making your applications configurable in Liferay 7.0.

Why was this change made? This change was made as part of the modularization efforts to ease portal configuration changes.

Deprecated the aui:tool Tag with No Direct Replacement

- **Date:** 2017-Feb-02
- **JIRA Ticket:** LPS-70422

What changed? The `aui:tool` tag has been deprecated with no direct replacement.

Who is affected? Plugins or templates that are using the `aui:tool` tag must remove their usage of the tag.

How should I update my code? There is no direct replacement. You should remove all usages of the `aui:tool` tag.

Why was this change made? This change was made as a part of the ongoing strategy to deprecate unused tags.

Build Auto Upgrade

- **Date:** 2017-Aug-17
- **JIRA Ticket:** LPS-73967

What changed? The `build.auto.upgrade` property in `service.properties` for Liferay Portal 6.x Service Builder portlets applies Liferay Service schema changes on rebuilding the services and redeploying the portlets.

Since 7.0, the per portlet property `build.auto.upgrade` is deprecated.

This change reintroduces Build Auto Upgrade in a new global property `schema.module.build.auto.upgrade` in the `[Liferay_Home]/portal-developer.properties` file.

Setting global property `schema.module.build.auto.upgrade` to `true` applies module schema changes for redeployed modules whose service build numbers have incremented. The `build.number` property in the module's `service.properties` file indicates the service build number.

Who is affected? This feature is available for developers to use in development only.

WARNING: DO NOT USE the Build Auto Upgrade feature in production. Liferay DOES NOT support Build Auto Upgrade in production.

How should I update my code? To use this feature in development, set global property `schema.module.build.auto.upgrade` in `[Liferay_Home]/portal-developer.properties` to `true`, increment your module's `build.number` in the `service.properties` file, and deploy the module.

Why was this change made? This change was made so that 7.0 developers could test database schema changes on the fly, without having to write upgrade processes.

Removed Exports from Dynamic Data Lists Web

- **Date:** 2017-Nov-27
- **JIRA Ticket:** LPS-75778

What changed? The `Dynamic Data Lists Web` module no longer exports the `com.liferay.dynamic.data.lists.web.asset` package.

Who is affected? This change affects anyone who is using the `com.liferay.dynamic.data.lists.web.asset` package. This particularly affects anyone using `com.liferay.dynamic.data.lists.web.asset.DDLRecordAssetRendererFactory` and casting the return `AssetRenderer` to `com.liferay.dynamic.data.lists.web.asset.DDLRecordAssetRenderer`.

How should I update my code? There are no replacements for this package; you must remove all usages. `DDLRecordAssetRendererFactory` can still be used as an OSGi service; however, you can no longer cast the returned `AssetRenderer` to `DDLRecordAssetRenderer`.

Why was this change made? This change was made to clean up LPKG dependencies.

Deprecated the social.activity.sets.enabled Property with No Direct Replacement

- **Date:** 2018-Jan-24
- **JIRA Ticket:** LPS-63635

What changed? The `social.activity.sets.enabled` property is no longer recognized by the Social Activity portlet. From Liferay Portal 7.0 onwards, Social Activity Sets will always be used.

Who is affected? This change affects anyone who has set the `social.activity.sets.enabled` property to `false`.

How should I update my code? No changes are necessary.

Why was this change made? The Social Activity portlet had two different versions with slightly different behaviors; one used in Liferay Portal and the other one in Social Office. To sync both components, and simplify its internal logic, activity sets are always enabled by default, with no option to disable them.

Removed Description HTML Escaping in PortletDisplay

- **Date:** 2018-Jul-17
- **JIRA Ticket:** LPS-83185

What changed? The portlet description stored in `PortletDisplay.java` is no longer escaped automatically.

Who is affected? This affects anyone who used the portlet description's escaped value to generate HTML. A small UI change could occur, as some characters may be unescaped.

How should I update my code? If you were using the `portletDescription` value to generate HTML, you should escape it using the proper escape sequence: `HtmlUtil.escape`.

Why was this change made? This change corrects a best practice violation regarding content escaping.

Removed Cache Bootstrap Feature

- **Date:** 2020-Jan-8
- **JIRA Ticket:** LPS-96563

What changed? The cache bootstrap feature has been removed. These properties can no longer be used to enable/configure cache bootstrap:
`ehcache.bootstrap.cache.loader.enabled`, `ehcache.bootstrap.cache.loader.properties.default`, `ehcache.bootstrap.cache.loader.properties.${specific.cache.name}`.

Who is affected? This affects anyone using the properties listed above.

How should I update my code? There's no direct replacement for the removed feature. If you have code that depends on it, you must implement it yourself.

Why was this change made? This change was made to avoid security issues.

140.9 What Changed Between Liferay npm Bundler 1.x and 2.x

This reference doc outlines the key changes between liferay-npm-bundler version 1.x and 2.x.

Automatically Formatting Modules for AMD

In version series 1.x of the bundler it was the developer's responsibility to wrap project modules in an AMD `define()` call. However, since 2.x the bundler does it for you, so the only requisite is that the project's code is transpiled/written for CommonJS modules model (the standard model for module handling in Node.js, that uses `require()` calls to load modules).

Isolating Project Dependencies

Package names are prefixed with the bundle name since version 2.0.0 of the bundler, but were left intact in previous versions. This strategy is used to isolate packages from different bundles. You can still deploy bundler 1.x packages (without prefix), and they will still work as they did for previous versions of the bundler.

Improved Peer Dependency Support

In bundler 1.x, there was only one shared peer dependency package available between portlets. With isolated dependencies per portlet, it's easy to honor peer dependencies perfectly. Peer dependencies can be resolved exactly as stated in projects because their names are prefixed with the project's name. This is possible because of the new liferay-npm-bundler-plugin-inject-peer-dependencies plugin. It scans all JS modules for `require` calls. If the bundler finds a required package in the `main.js` file, but it is not declared in the `package.json`, it resolves it to the proper version that is found in the `node_modules` folder. The plugin then injects a new dependency in the output `package.json` for the required package.

Note that injected dependency version constraints are the specific version number required, without caret or any other semantic version operator. This is to honor the exact peer dependency found in the project. Injecting more relaxed semantic version expressions could lead to unstable results.

Manually De-duplicating Through Importing

Namespacing means that each portlet gets its own dependencies. Only using the bundler this way obtains the same functionality as standard bundlers like webpack or Browserify, so you wouldn't need a specific tool like liferay-npm-bundler. Since Liferay DXP is a portlet based architecture, sharing dependencies among different portlets would be very beneficial.

In bundler 1.x that deduplication was made automatically, but there was no control over it. However, with version 2.x, you may now import packages from an external OSGi bundle, instead of using your own. This lets you put shared dependencies in one project, and reference them from the rest. Though This new way of de-duplication is not automatic, it leads to full control (during build time) of how each package is resolved.