-and-panels

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

Developing Liferay DXP 7.1
by The Liferay Documentation Team

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.

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.

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

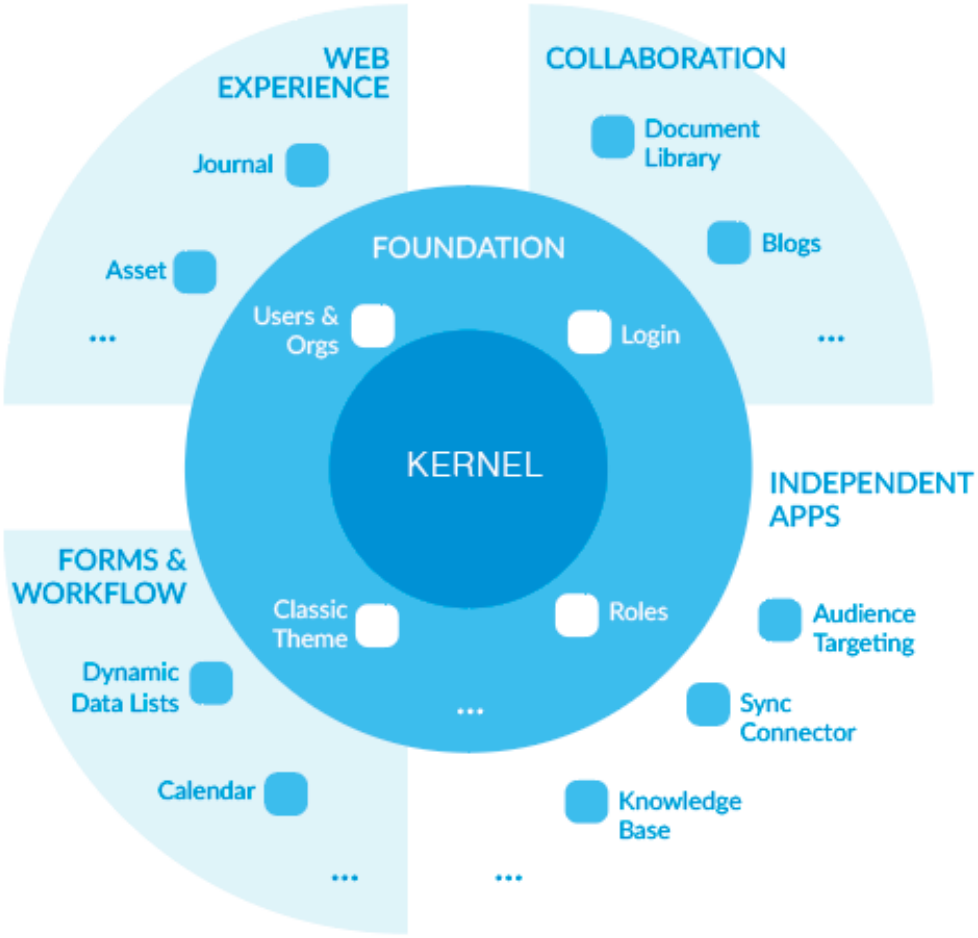


Figure 1.2: Liferay DXP ships with suites of applications to get you started building your site quickly.

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 it 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 the JVM supports, 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?

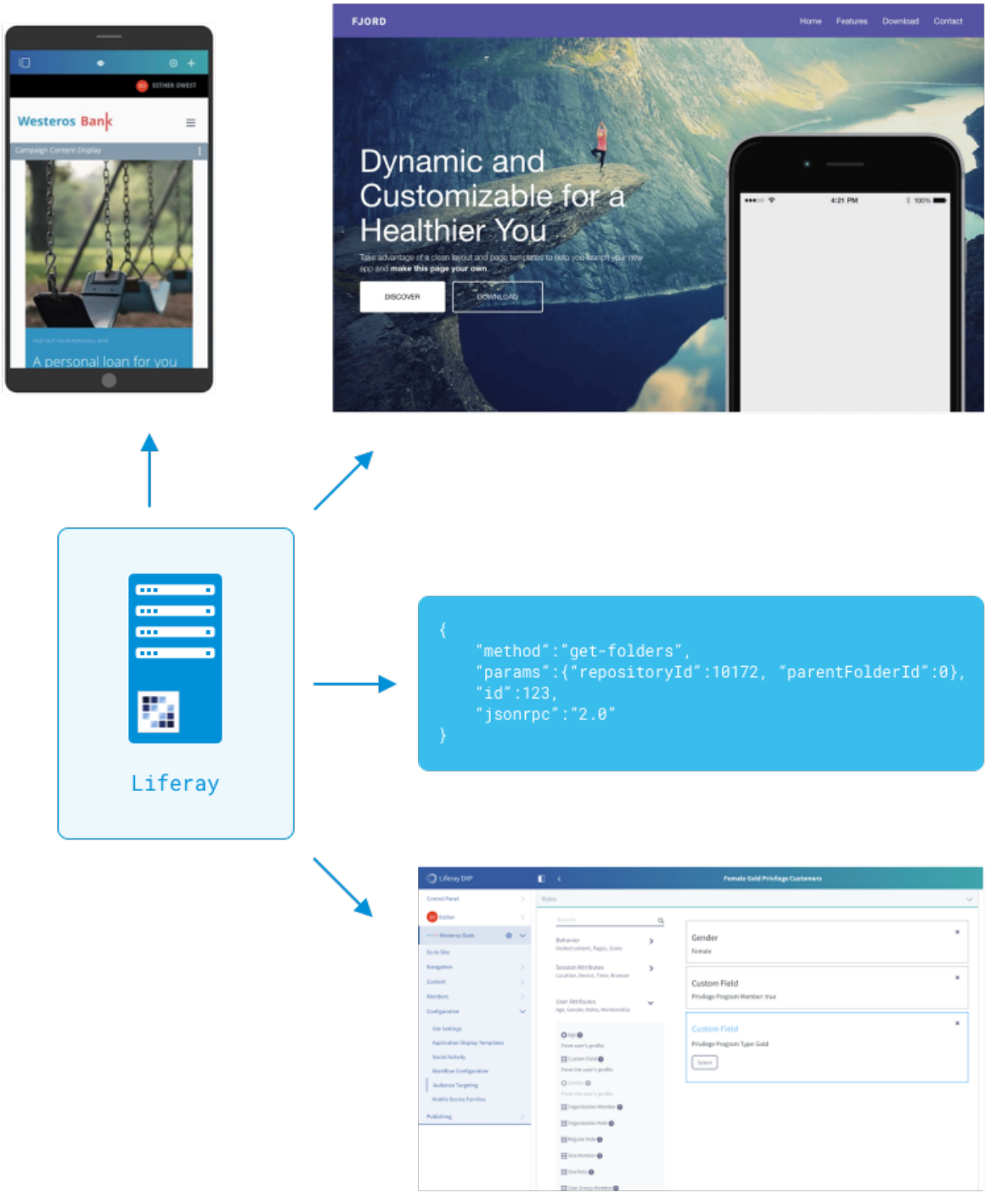


Figure 1.3: Developers use Liferay DXP in many ways.

1. It's Open Source and puts a strong emphasis on following standards, instead of reinventing the wheel.
2. It's based on Java EE 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 you can use tools you prefer.
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 product changes, 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 Liferay DXP work or for reporting bugs you find (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. There are options for all levels of expertise and time availability.

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 Liferay DXP supports:

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

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

develop and deploy enterprise-ready, scalable web and mobile-based applications in a dynamic, component-based environment.

With Java EE 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.

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, 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 our MVCPortlet framework.

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 JS or Angular
- 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 Clay.

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

For templating, Java EE'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.

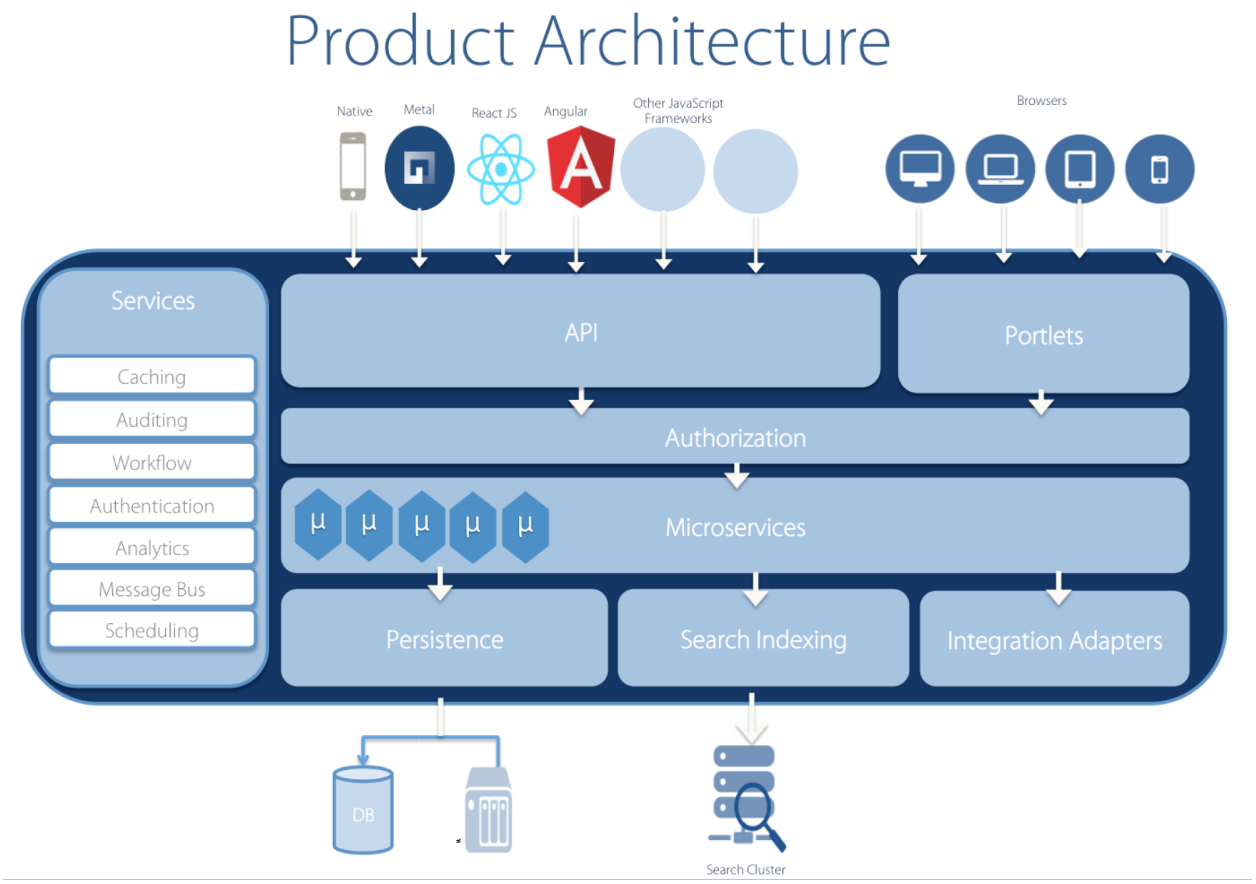
Liferay has also chosen build tools that give you freedom to use any development environment. Gradle along with bnd powers the product's build, but project layouts are dynamic, which means you can use anything from Maven to Ant/Ivy to build applications for Liferay.

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 you can use 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 you can build and share new best-of-breed apps



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 you 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 lets you not only 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. You can create applications by leveraging Liferay's many frameworks that encapsulate features that today's applications commonly need. For example, a commenting system lets you attach comments to any asset that you define, whether they be assets you develop or assets that ship with the system. Assets are shared by the system and are 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 you to integrate it with your applications. Many of the frameworks you might need when developing complex applications are already in Liferay; 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 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 CLI 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 returns Hello, John.

Here's what the project layout looks like:

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.

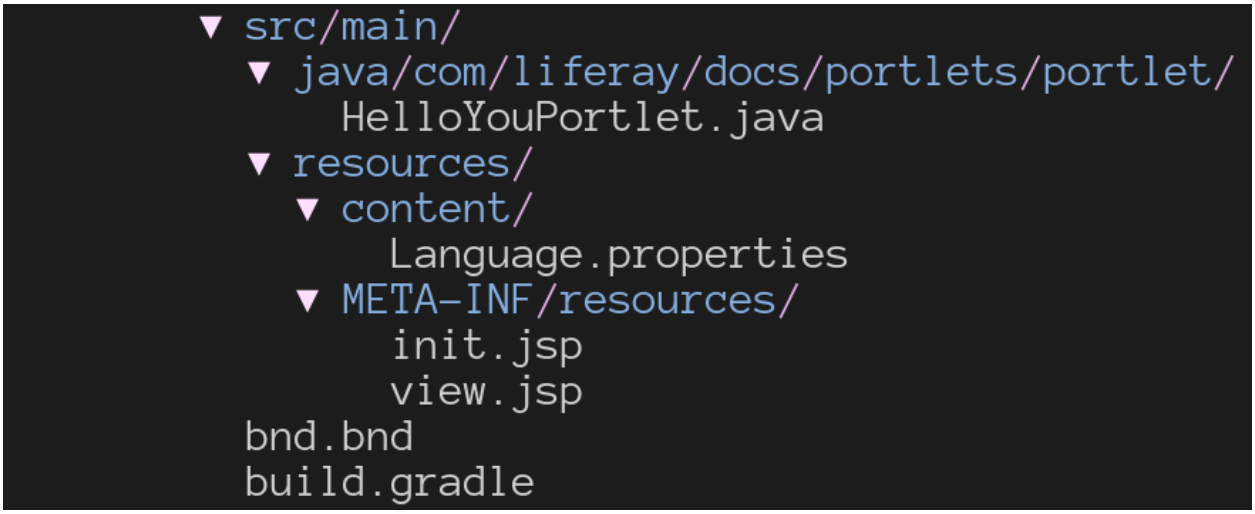


Figure 1.5: The Hello You portlet has a simple project structure.

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. You can extend the UI 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:

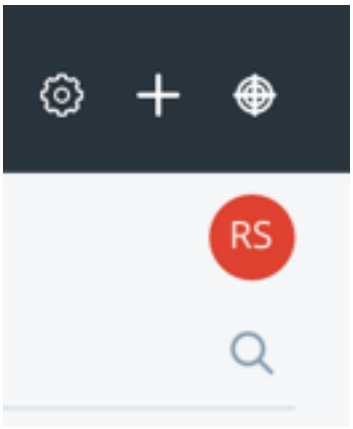


Figure 1.6: The product menu appears beneath the user's profile link.

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

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

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.

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
```

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

```
    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://liferay.com";

    }

    @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 and the module starts, the link appears in the menu. 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 Liferay's code, 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.

1.6 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 you're familiar with.
- **Writing code and configuration files**: A manifest, Java classes, and resources. Modules stubbed out with them let you focus on implementing logic.
- **Compilation**: Configuring 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. This tutorial demonstrates starting a new module using Liferay Workspace. It's Liferay's opinionated build environment based on Gradle and bnd that simplifies module development and automates much of it.

Note: Liferay lets you develop using your favorite tools. 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 in Liferay Dev Studio DXP, Maven tutorials, and samples
- Gradle in Liferay Dev Studio DXP and samples

Note: Themes and Layout Templates are not built as modules. The Themes and Layout Templates tutorials demonstrate creating them.

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.

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 projects and performs common tasks.

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 folder structure looks like the one shown in the figure below.

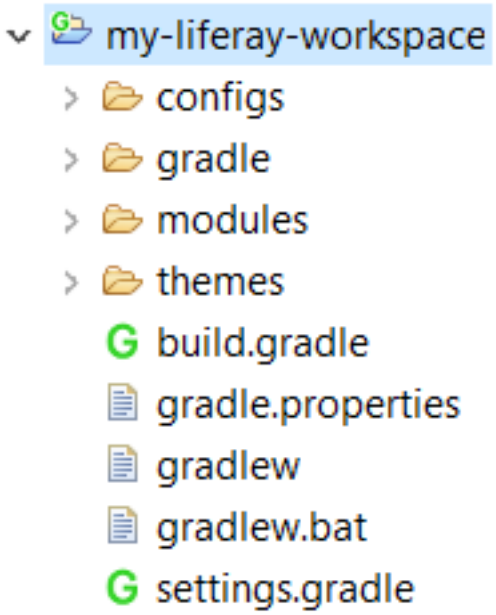


Figure 1.9: Liferay Workspace aggregates projects so they can leverage the Gradle build environment.

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 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 Dev Studio DXP let you create and import Liferay Workspace projects.

To create a project, follow the tutorial Creating a Liferay Workspace Project with Liferay Dev Studio DXP.

To import a project, use the wizard from *File → Import → Liferay → Liferay Workspace Project*.

The Workspace is ready for creating modules.

Creating a Module

Blade provides project *templates* and *sample* projects. The templates stub out files for different types of modules. The samples can be generated in a Workspace and demonstrate many module types. Templates and samples help you create modules fast.

Using Module Templates

The Blade command `blade create -l` lists the project 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 1.10: Blade's `create` command generates a project based on a template. Executing `create -l` lists the template names.

Note: Liferay Dev Studio DXP's module wizard lets you select a module project template.

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 properties 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 project.

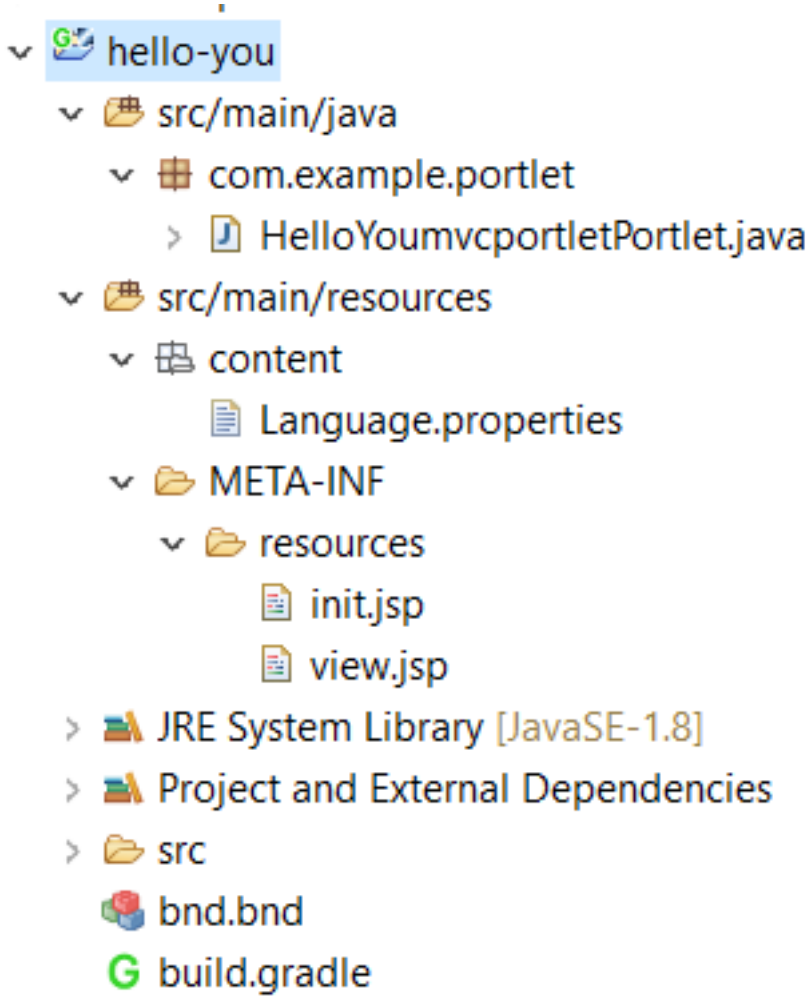


Figure 1.11: Liferay modules use the standard Maven folder structure.

Sample modules are another helpful development resource.

Using Sample Modules

An alternative to creating a module from a template is to generate a *sample* module. Examine them or modify them for your purposes.

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 1.12: The `blade samples` command lists the sample modules you can create, examine, and modify to meet your needs.

Here's the Blade samples command syntax:

```
blade samples [sampleName]
```

It creates the sample project in a subfolder.

Building a module and deploying it to Liferay is easy.

Building and Deploying a Module

Liferay Workspace provides Gradle tasks for building and deploying modules. Blade's `blade gw` command lets you invoke the Gradle wrapper from any project folder. 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 bnd 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.

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 deploys modules to the local Liferay server. It communicates with the OSGi framework using Felix Gogo shell and deploys modules directly to the OSGi container using Felix File Install commands. The command above uses the default port 11311.

To deploy the module, execute this command:

```
blade deploy
```

It deploys modules in the current folder tree. For example, executing `blade deploy` in the `[workspace]/modules` folder deploys all the modules in that folder and its subfolders.

Liferay Dev Studio DXP lets you deploy modules by dragging them from the Package Explorer onto the Liferay server. Dev Studio DXP provides access to Liferay Workspace Gradle tasks too.

Note: Blade CLI directly installs modules into the OSGi container. Blade stores the module differently in Liferay than if you were to copy the module into the `LIFERAY_HOME/deploy` folder.

Once you've deployed a portlet module, it's available in the Liferay 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.

Redeploying Module Changes Automatically

Blade lets you 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
: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
```

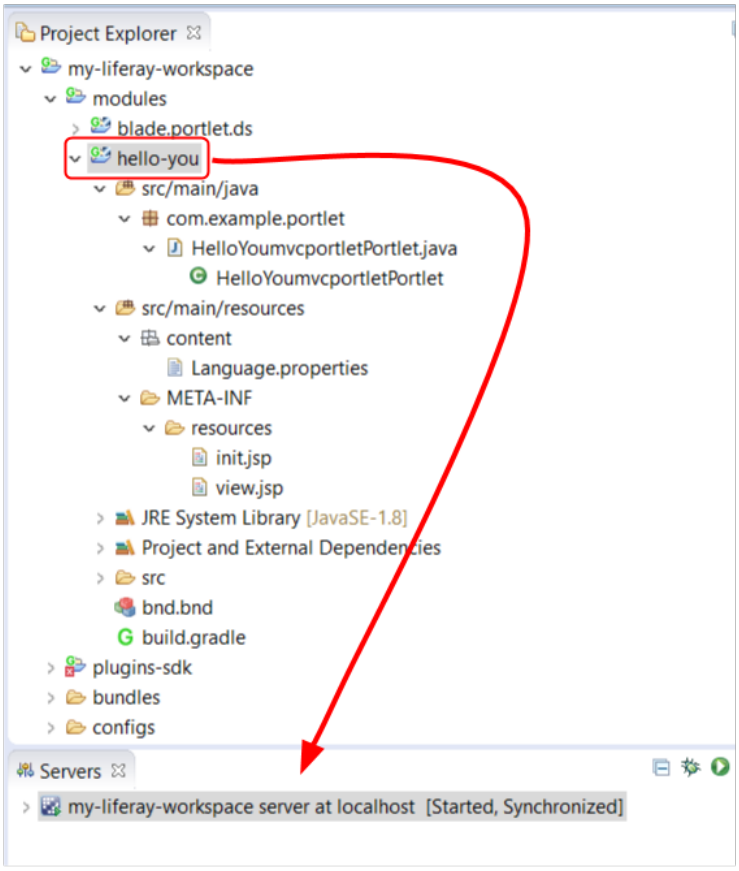


Figure 1.13: Liferay Dev Studio DXP lets you deploy modules using drag-and-drop.

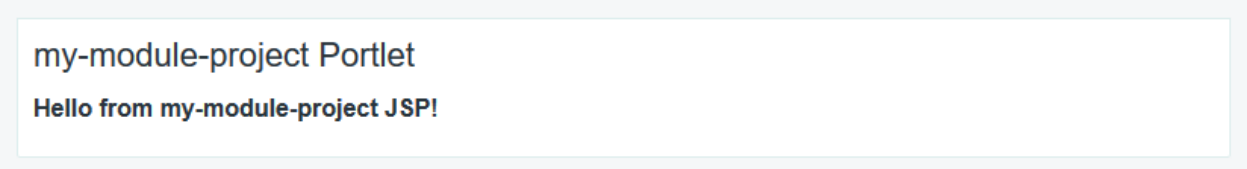


Figure 1.14: Here's a bare-bones portlet based on a project template.

```
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)
```

The command output indicates that the module is installed and started, reports the module's OSGi bundle ID, and stands ready to deploy changes to the module.

Congratulations on a great start to developing your module!

Related Articles

Configuring Dependencies
- Liferay Workspace
- Tooling
- OSGi Basics for Liferay Development
- Portlets

1.7 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

Note: The App Manager shows each module's version number.

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

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

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`

File	Group ID	Artifact ID	Version	Origin
`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.

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 of Liferay'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 instance. 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.

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

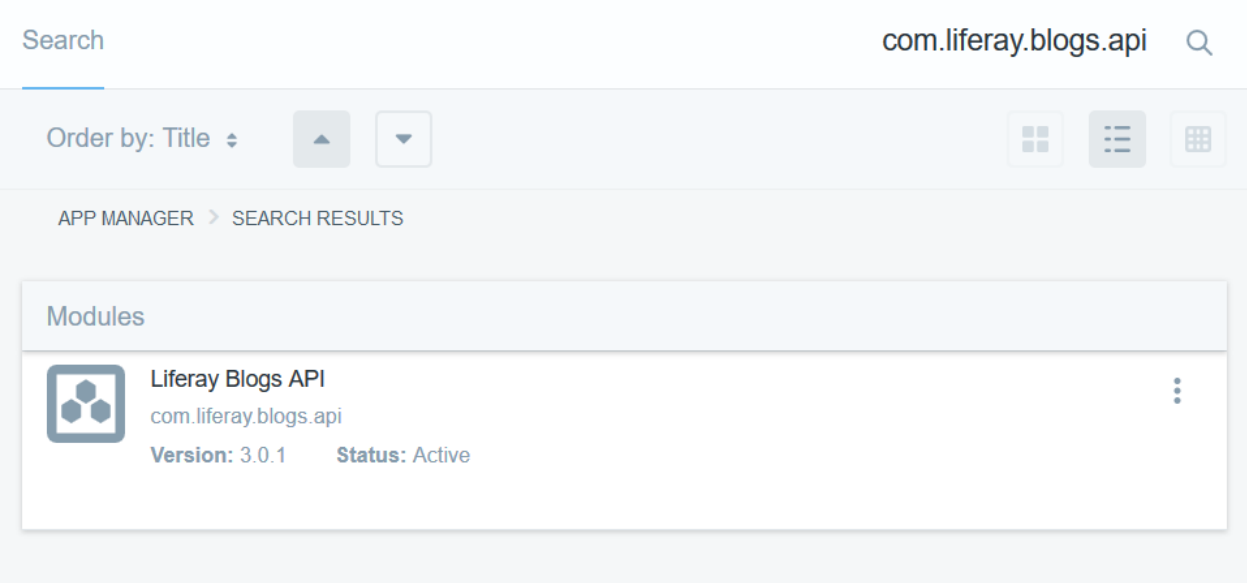


Figure 1.15: You can inspect deployed module artifact IDs and version numbers.

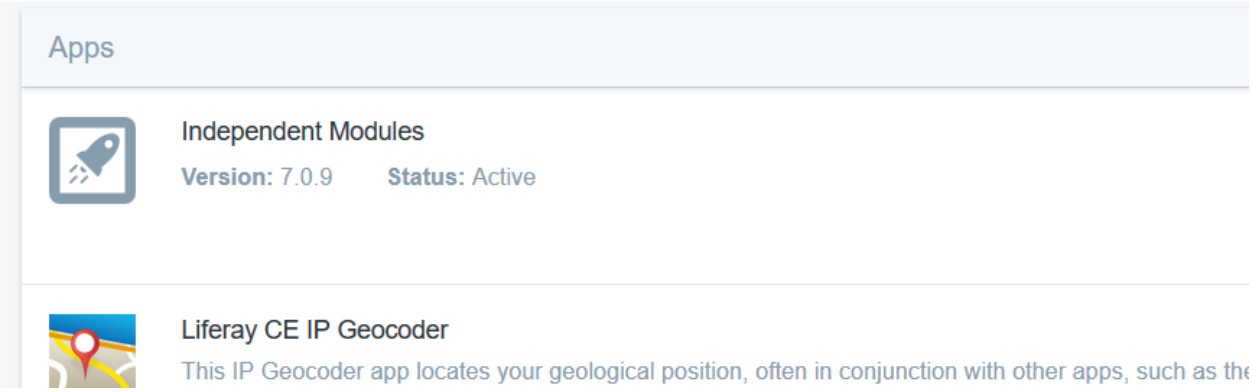


Figure 1.16: The App Manager aggregates Liferay and independent modules.

1. Navigate to the Gogo Shell portlet in the Control Panel → *Configuration* → *Gogo Shell*. You can enter commands in the provided 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 [https://docs.liferay.com/dxp/apps](

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`.

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.

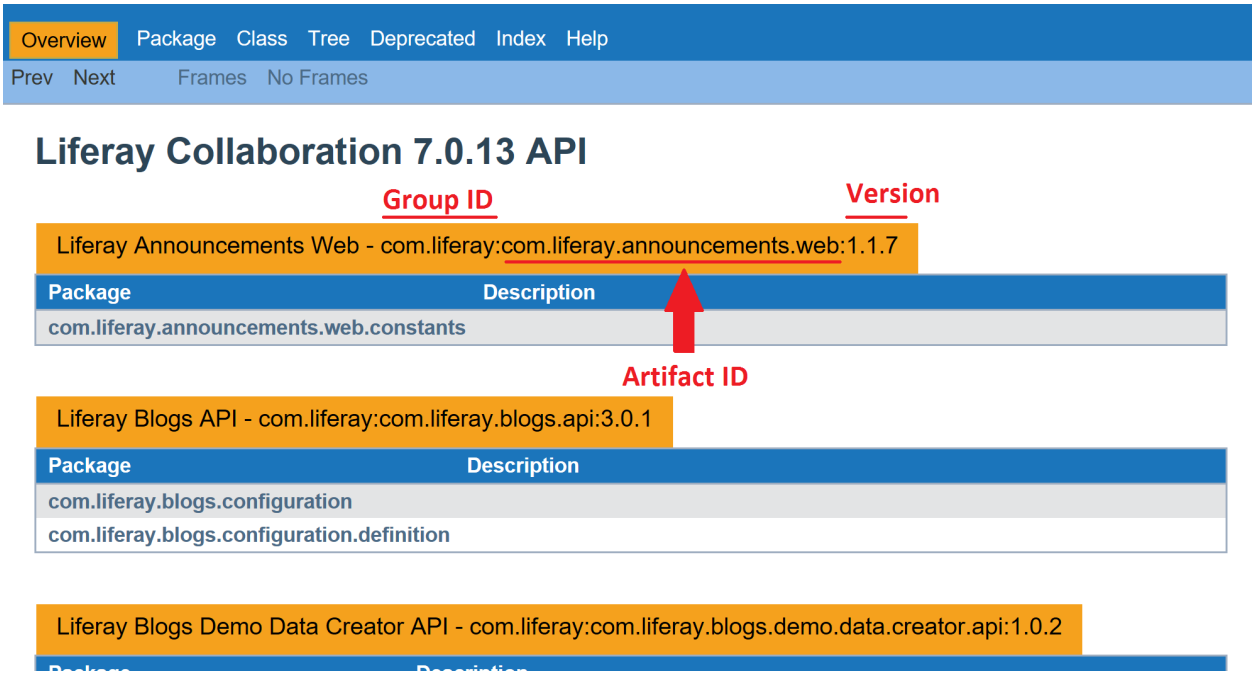


Figure 1.17: Liferay DXP app Javadoc overviews list each app module's display name, followed by its group ID, artifact ID, and version number in a colon-separated string. It's a Gradle artifact syntax.

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.

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/7-1/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!

Related Topics

Using the App Manager
Reference
Resolving Third Party Library Package Dependencies
Tooling
Portlets

1.8 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 it has 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 components you want to customize.

This tutorial demonstrates finding an extension point. It steps through a simple example that locates an extension point for importing LDAP users. The example includes using Liferay DXP's Application Manager and Felix Gogo Shell.

Locate the Related Module and Component

First think of words that describe the application behavior you want to change. The right keywords can help you 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 Liferay feature 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* is found in the Liferay Foundation app suite's list of apps and features. Select the app suite.

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

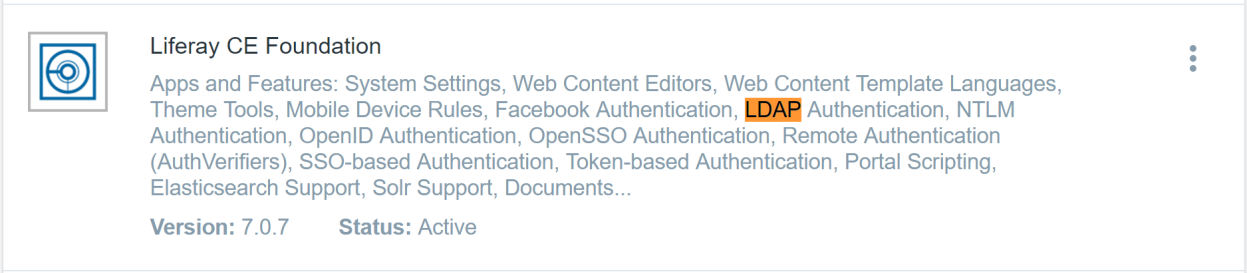


Figure 1.18: The Liferay Foundation app suite contains the LDAP Authentication application.

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.

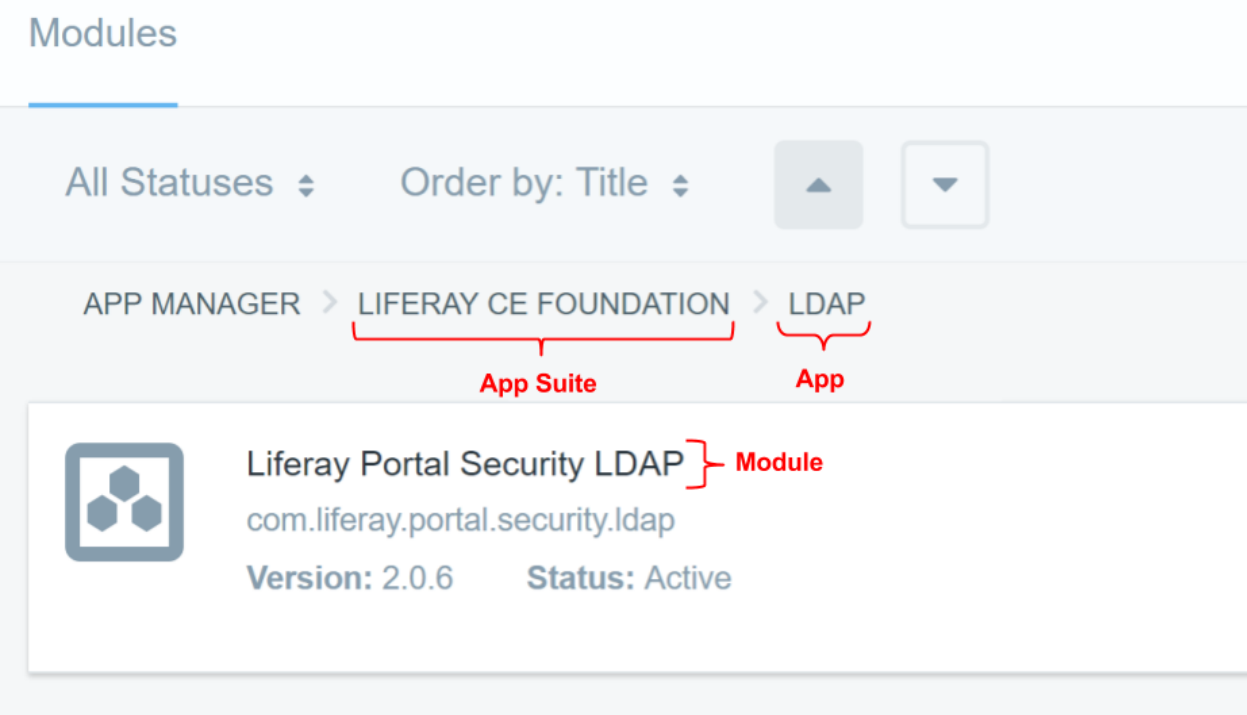


Figure 1.19: The App Manager lists the module, package name, version, and status.

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 you're looking for; Liferay's naming
conventions facilitate finding extension points in a manageable time frame.
```

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

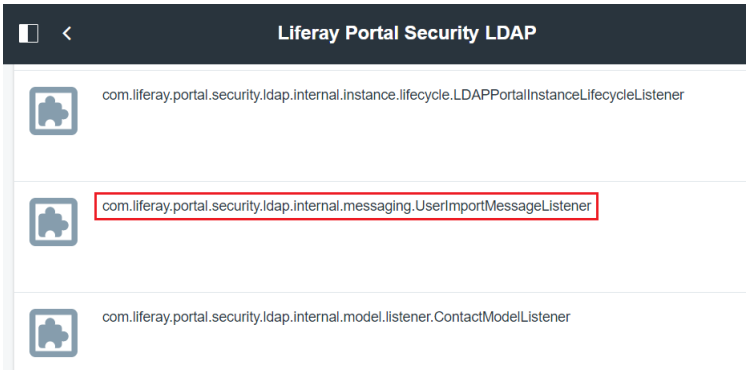


Figure 1.20: The component name can be found using the App Manager.

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. This tutorial assumes you're using the Gogo shell.

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 services the component references. They 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 for customizing the LDAP user import process! 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 tutorial here.

Important Not all Liferay extension points are available as referenced services. Service references are common in Declarative Services (DS) components, but extension points can be exposed in other ways too. 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! In the App Manager, you used keywords to find the module component whose behavior you wanted to change. Then you used Gogo shell to find the component extension point for implementing your customization.

CHAPTER 2

INTRODUCTION TO FRONT-END DEVELOPMENT

Liferay DXP offers complete developer front-end freedom. Whether you like coding JavaScript by hand, have used Liferay's front-end frameworks in the past, or prefer jQuery, Lodash, or modules, you can use your front-end framework of choice.

Prior users of Liferay DXP can continue to use Liferay's venerable Alloy UI, but you can also use the front-end technologies you love the most:

- EcmaScript ES2015+
- Metal.js (developed by Liferay)
- AlloyUI (developed by Liferay)
- jQuery (included)
- Lodash (included)

To load modules, you must know when they are needed, where they are at build time, if you want to bundle them together or load them independently, and you must assemble them at runtime. Keeping track of these tasks can be a hassle. Liferay DXP's Loaders (YUI/AUI, AMD, and npm in AMD format) handle loading for you, so you don't have to worry about all the details. Just provide a small bit of information about your module, and Liferay DXP's loaders take care of the rest.

If you want to use npm packages in your applications, you can use `liferay-npm-bundler`. It is built for just this purpose, and even provides several presets for common module types (AMD, React, Angular JS, etc.) to save you time. It creates an OSGi bundle for you, extracts all npm dependencies, and transpiles your code for the Liferay AMD Loader.

While developing JavaScript applications for Liferay DXP, you may need to access Liferay DXP-specific information or web services. The `Liferay` global JavaScript Object exposes this information for you, letting you harness the power of Liferay DXP in your JavaScript applications, while still using the front-end frameworks and technologies that you love.

2.1 Lexicon and Clay

Liferay DXP uses its own design language, called Lexicon, to provide a common framework for building consistent UIs and user experiences across the Liferay product ecosystem. The web implementation of Lexicon (CSS, JS, and HTML) is called Clay. It is automatically available to application developers through a set of CSS classes or our tag library.

2.2 Templates

For templating, you can use Java EE's JSP, FreeMarker, Google's Soy (aka Closure Templates), or whatever else you like.

2.3 Themes

A Liferay DXP Theme provides the 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.

From the Theme Builder Gradle Plugin, to the Liferay Theme Generator, to Dev Studio DXP, to Blade CLI's Theme Template, you can choose the development tools you like best, so you can focus on creating a well designed theme.

2.4 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
- Creating Configurable Styles for Portlet Wrappers
- Dynamic Includes

Read on to learn more.

CHAPTER 3

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 familiar with how things work, proficient with the platform, and start to think in terms of how you'd solve problems most effectively using the platform's tools.

7.0 was designed as an enhancement that builds off of what you already know. Its Upgrade Planner and this 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. Since you already know previous versions of Liferay Portal, you're several steps ahead of everybody else.

Here you'll learn about the benefits of 7.0 for developers compared to previous versions, the architectural improvements, the benefits that modularity brings, 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.

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.

3.1 What Hasn't Changed and What Has

Liferay 7.0 was a new major version of the Liferay platform and as such it included 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 Liferay 7, 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, making it easier to replace 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 Dev Studio DXP 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.

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.

8. The Plugins SDK is no longer available. Visit the Deprecated Apps in 7.1: What To Do article for more information on the Plugins SDK removal. Liferay Workspace, is now Liferay's opinionated development environment.

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.

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.

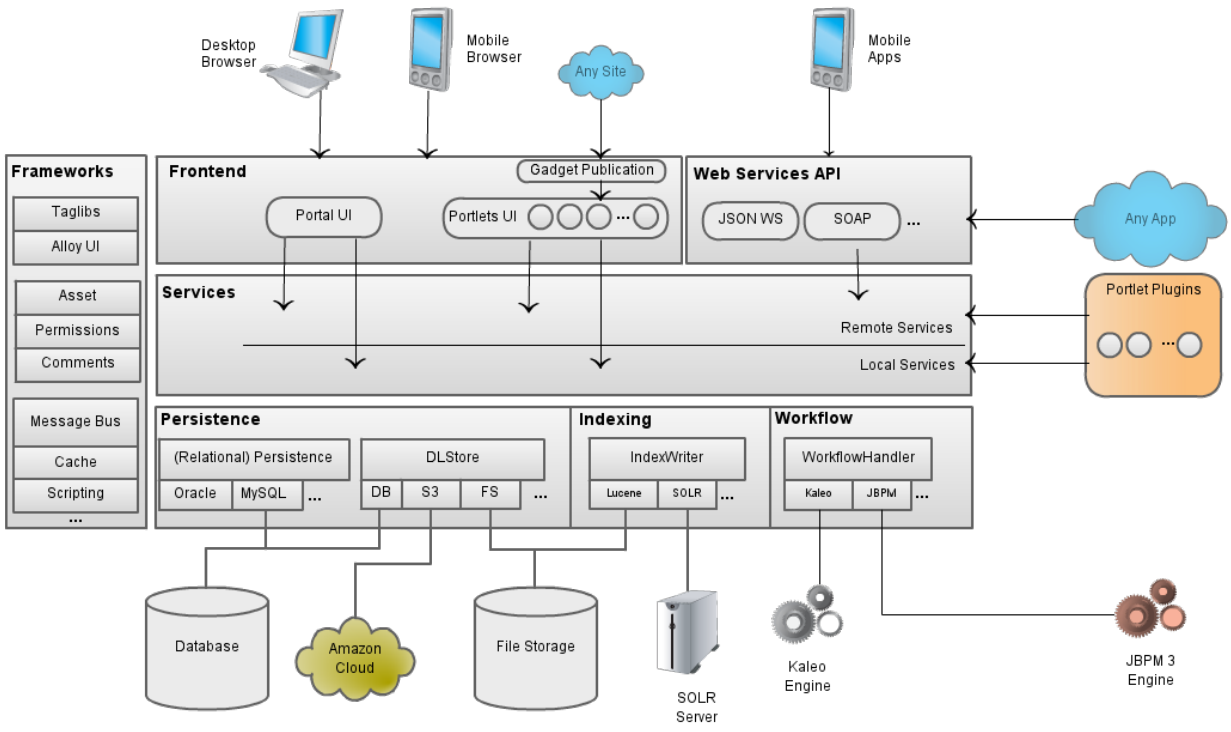


Figure 3.1: Liferay Portal 6's architecture, shown in this figure, is still generally valid in 7.0.

The most significant change (and improvement) over this architecture is that the portal is no longer a single large Java EE Web Application. Liferay Portal 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.

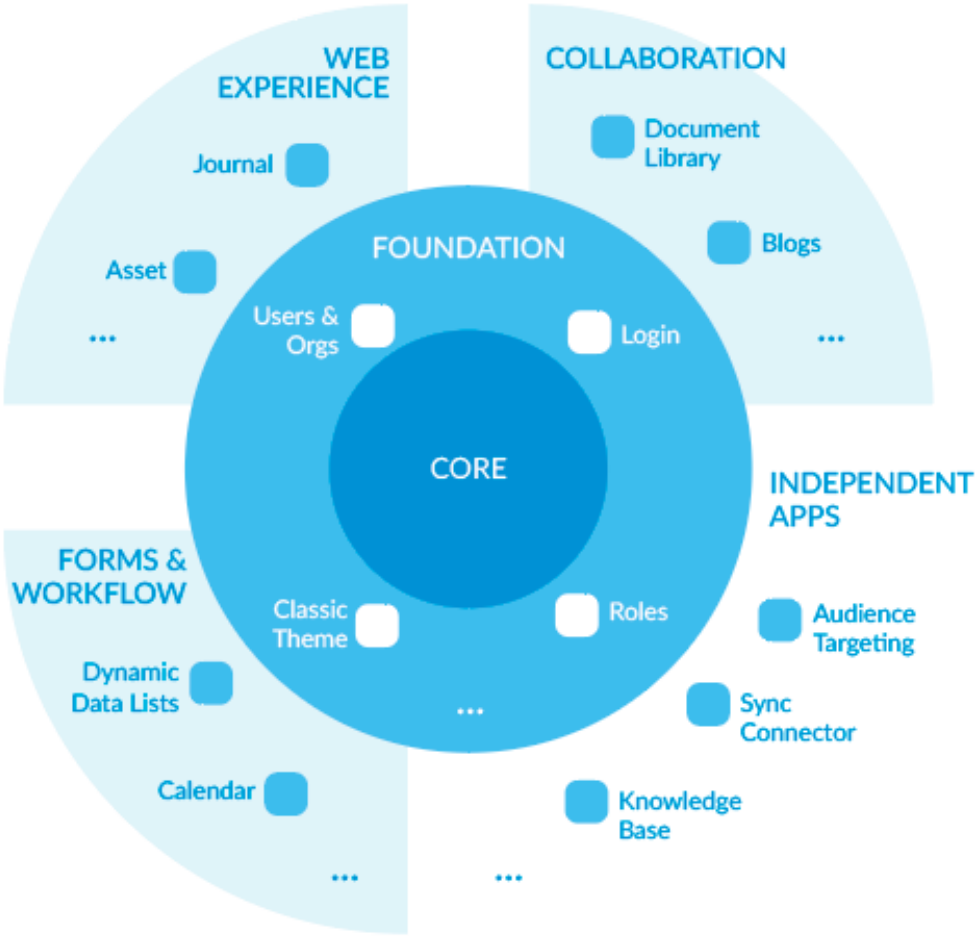


Figure 3.2: 7.0 is composed of the Liferay Core, independent application modules, and App Suites, each with their own set of application and framework modules.

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.

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.

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 Liferay 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 Liferay too. The Portlet Compatibility Layer converts each plugin WAR to a Web Application Bundle (WAB), which is a module.

Let's consider the structure of a Liferay DXP 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 folders 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 Liferay 7 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 Liferay 7.

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

3.2 Benefits of 7.1 for Liferay Portal 6 Developers

More than in any other Liferay release, 7.0 centers on you, the developer. Liferay's platform was rebuilt in Liferay 7.0, 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.

Simpler and Leaner

Liferay has always been simple and lean, compared to the proprietary alternatives; Liferay 7 widens the gap even more.

Liferay 7 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, you can now more easily reuse your existing knowledge and use what you learn developing for Liferay outside of it.

Liferay 7 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.

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.

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).

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.

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 folder structure and build system based on Gradle and bnd. Liferay Workspace can be used standalone through the command line or with Liferay Dev Studio DXP, which runs on Eclipse.

If you have an investment in a specific build tool, such as Maven, developing on Liferay will be easier than ever before. We have built Maven plugins for typical Liferay-specific development tasks (such as using Service Builder) and provide a collection of new archetypes.

The Plugins SDK is no longer available for 7.0. Liferay Workspace, is now Liferay's opinionated development environment. You can transition from a Plugins SDK by adding it to your workspace and migrating projects at your own pace. See the Using a Plugins SDK from Your Workspace article for more information.

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.

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 4

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!

Before setting up your tools and upgrading your plugins, you'll learn how 7.0 uses OSGi to meet the objectives listed above. And equally important, you'll discover how easy and fun modular development can be.

4.1 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 Liferay DXP and OSGi offer, however, you should use OSGi modules.

Modules offer these benefits:

- **Better Encapsulation** - The only classes a module exposes publicly are those in packages 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:

- `[Project root]`
  - `[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 folder 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 Dev Studio DXP. Note also that Liferay Dev Studio DXP 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, you can use bnd from the command line and from Liferay @ide@.

It's time to get hands-on experience creating and deploying an OSGi module. That's next.

4.2 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 create 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 for development in a Liferay Workspace, it can easily be modified to use in other build environments.

Place the project files in a subfolder of your Liferay Workspace's `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.

Now that you've seen an OSGi module in action, you can appreciate more of the ways modularity and OSGi improves development on Liferay. They're explained next.

4.3 More Ways OSGi Improves Development on Liferay

Here are more ways OSGi meets the developer experience goals mentioned earlier. These topics apply to both new and veteran Liferay developers, so you should become familiar with them. After reading each one, make sure to COME BACK HERE to continue with the next tutorial listed or, when you've finished reading all of them, continue this Learning Path.

1. Leveraging Dependencies: In 7.0, you can both declare dependencies among modules and combine modules to create applications. Since leveraging dependencies provides huge benefits, there's an entire tutorial for it.

2. OSGi Services and Dependency Injection: OSGi includes 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. It's 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.

3. 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.

Liferay's developer tooling for 7.0 compliments module development. It supports traditional plugin development and facilitates moving applications to modules. There are improvements for developing themes and using Maven and Gradle build systems, and there's even a Upgrade Planner tool that adapts existing code to 7.0's API and automates much of the plugin upgrade process. Developer tooling improvements are next.

CHAPTER 5

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) has all these things. It's a Gradle-based development environment that integrates with Liferay Dev Studio DXP and can be used in conjunction with other IDEs, such as a "vanilla" Eclipse, IntelliJ, or NetBeans. You can extend Workspace's Gradle environment with additional Gradle plugins for testing, code coverage analysis, and more.

If you prefer Maven over Gradle, you can use the Maven-based Workspace instead. 7.0's' lean artifacts and new project archetypes and Maven plugins make Liferay DXP development with Maven easier than ever.

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

Liferay's tools also streamline the application upgrade process. Liferay Dev Studio DXP's Upgrade Planner adapts traditional plugins to 7.0 APIs. The Liferay Theme Generator migrates themes and layout templates to use the new Liferay JS Theme Toolkit and adapts them to 7.0.

Here are the tooling improvement topics:

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

5.1 From the Plugins SDK to Liferay Workspace

The Liferay Plugins SDK is not available for 7.0. Visit the Deprecated Apps in 7.1: What To Do article for more information on the Plugins SDK removal. 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
- Project validation
- Integration testing
- 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 Dev Studio automatically adapts existing plugins to 7.0. There's also a tutorial that demonstrates how you can optionally migrate traditional plugins to Workspace.

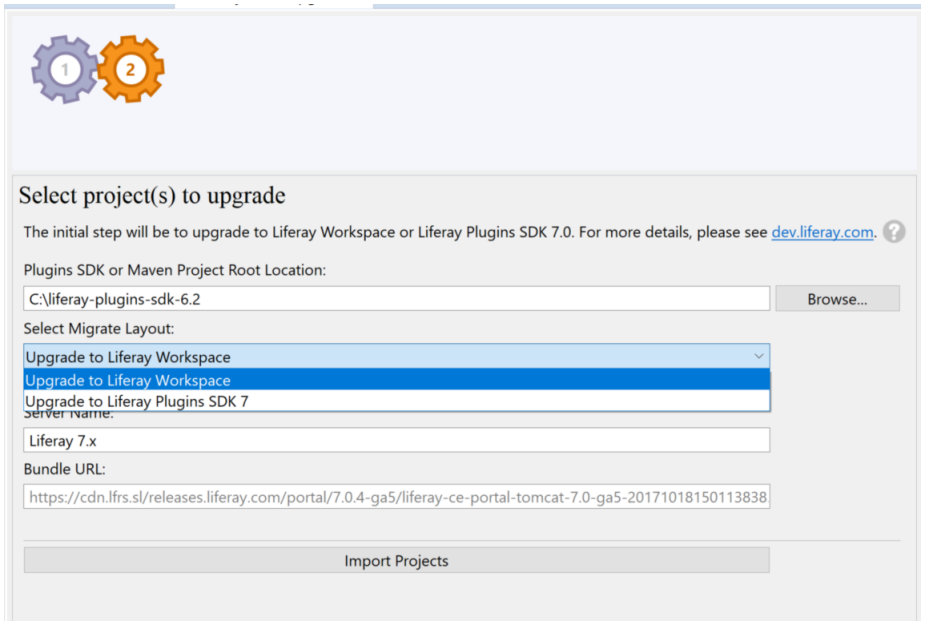


Figure 5.1: Liferay Dev Studio DXP's Upgrade Planner automates many aspects of the plugin upgrade process.

Here's an example Workspace folder structure:
Here's the Workspace anatomy:

- `bundles/` (generated) → default folder for Liferay DXP bundles
- `configs/` → holds Portal server configurations
- `ext/` → holds Ext modules and Ext WAR files
- `gradle/` → holds the Gradle wrapper files
- `modules/` → holds module projects
- `plugins-sdk/` (generated) → holds plugins from previous releases

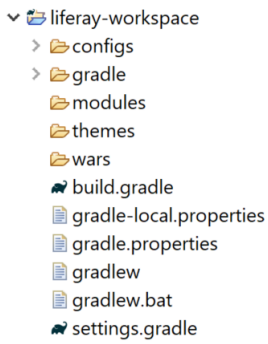


Figure 5.2: Liferay Workspace aggregates projects to use the same server configurations and Gradle build environment.

- `themes/` → holds theme projects created with the Liferay Theme Generator, which use the Liferay JS Theme Toolkit
- `wars/` → holds traditional web application projects
- `build.gradle` → common Gradle build file
- `gradle-local.properties` → sets user-specific properties for your workspace
- `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. The Gradle property `liferay.workspace.environment` in Workspace's `gradle.properties` file specifies the configuration to use. See the Testing Projects section for more details.

Other Gradle properties set root locations for the Liferay DXP bundle, modules, themes, and a Plugins SDK. See the Gradle Workspace Properties section for a list of all available Workspace properties.

Workspace Folder Structure Properties

Property	Description
`liferay.workspace.environment`	Name of a `configs` subfolder holding the Portal server configuration to use
`liferay.workspace.ext.dir`	Ext projects root folder
`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
`liferay.workspace.wars.dir`	WAR-style projects root folder

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

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 project to Liferay DXP's OSGi 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
`buildDB`	Builds database SQL scripts from the generic SQL scripts
`buildSoy`	Compiles Closure Templates in JavaScript functions
`components`	Lists the project's components
`configJSModules`	Generates the config file needed to load AMD files via combo loader in Liferay DXP
`dependencies`	Lists the project's declared dependencies
`formatSource`	Runs Liferay Source Formatter to format project files
`initBundle`	Downloads and installs a Liferay DXP bundle
`model`	Lists the project's configuration model
`projects`	Lists the project's sub-projects
`properties`	Lists the project's
`replaceSoyTranslation`	Replaces `goog.getMsg` definitions
`transpileJS`	Transpiles the project's JavaScript files

This is just a subset of available Gradle commands in a Liferay Workspace. Run `gradlew tasks` from a project in workspace for a full list of Gradle commands.

5.2 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 thirty templates for Gradle and Maven-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
- etc..

Blade creates modules based on these templates. For a full list of these templates, visit the Project Templates reference section.

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 Dev Studio DXP's module project wizard creates Workspace modules from the templates too.

Liferay Dev Studio DXP'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 Dev Studio DXP and Blade. Workspace uses BndTools to generate each module's OSGi headers in a `META-INF/MANIFEST.MF` file. Workspace deploys modules to the OSGi container using Felix File Install commands.

Liferay Dev Studio DXP lets you deploy modules by dragging them onto your Portal server.

To learn more about Workspace and using it in Liferay Dev Studio DXP, see this tutorial.

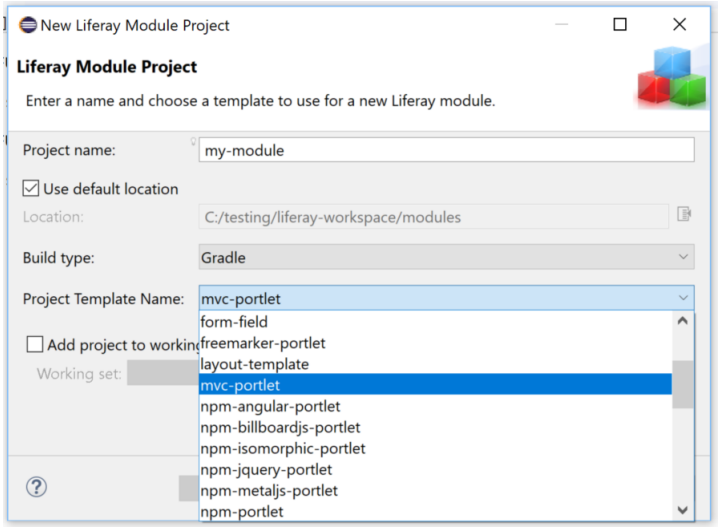


Figure 5.3: Liferay Dev Studio DXP lets developers select templates to stub out modules.

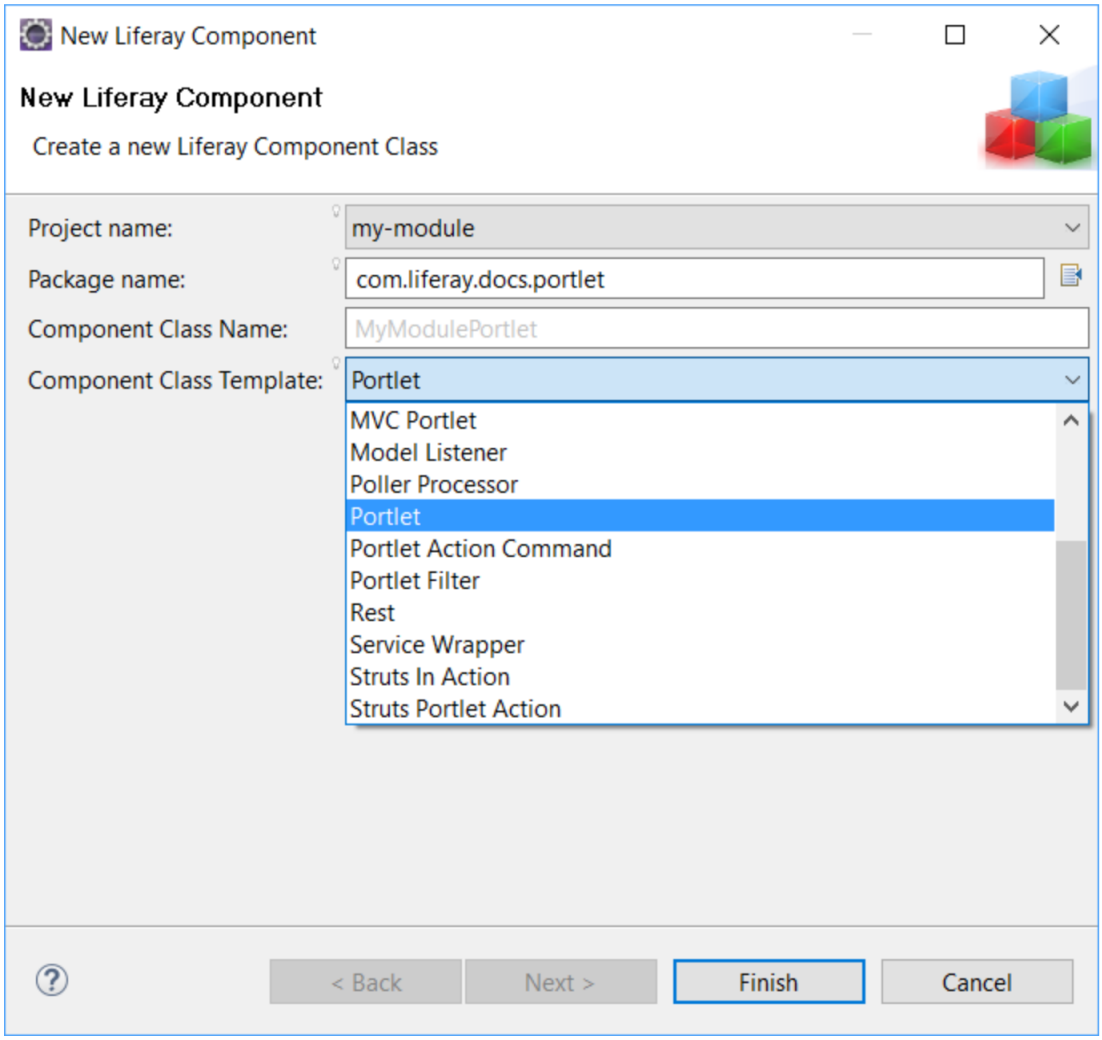


Figure 5.4: Liferay Dev Studio DXP's component wizard facilitates creating component classes.

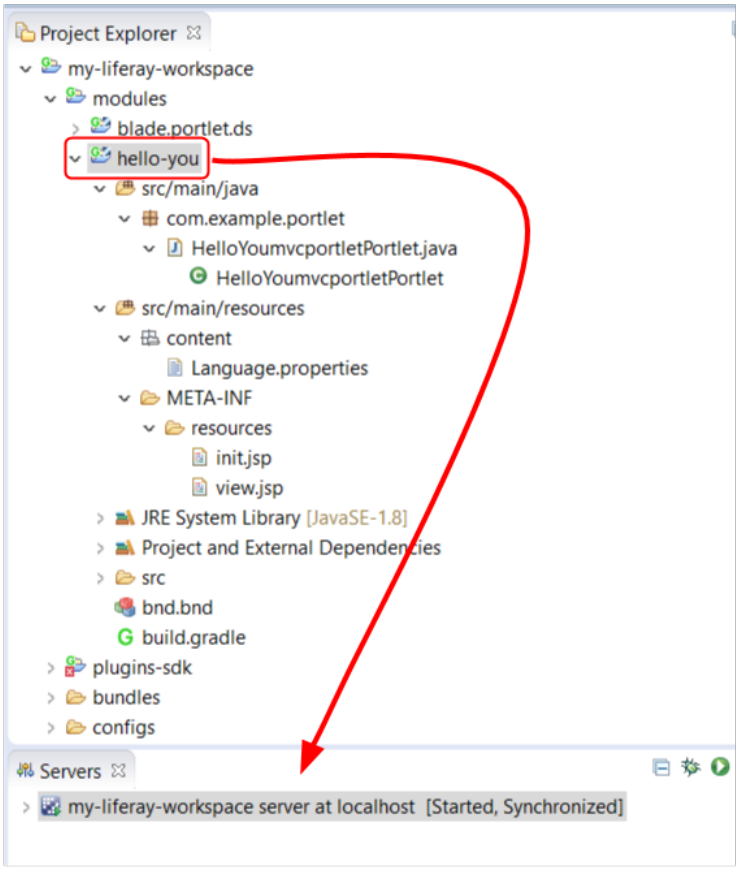


Figure 5.5: Liferay Dev Studio DXP lets you deploy modules using drag-and-drop.

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
```

Make sure to check out the tutorial Starting Module Development for even more information on module development.

5.3 What's New for Maven Users

Liferay Portal 7.0+ and Liferay DXP fully support Maven development and offers several new and improved features:

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

- More granular dependency management

The new archetype `com.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 thirty Maven archetypes for 7.0, and more are in development. Here are some popular ones:

- Configuration Icons
- Fragments
- Menu Buttons
- Portlets
  - MVC
  - npm
  - Soy
  - Spring MVC
- Service Builder
- Themes
- etc...

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.

Liferay also provides several new and updated Maven plugins that simplify the build process. For example, 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 DXP'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.

5.4 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 Dev Studio DXP
- Liferay Dev Studio DXP'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:

- Gradle
- Liferay Workspace
- Maven

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 6

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 Dev Studio DXP'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 Clay, to apply a clean consistent application user experience.

See the optimization tutorials for more options and details.

The Plugins SDK is no longer available to develop plugins for 7.0. Visit the Deprecated Apps in 7.1: What To Do article for more information on the Plugins SDK removal.

In light of the removal, 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 Dev Studio DXP supports developing in Workspaces using Gradle or Maven.

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

6.1 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.

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 → MVCRenderCommand - serveResource → MVCResourceCommand	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 Clay- Convert to OSGi modules
Portlet - JSF	Upgrading a Liferay JSF Portlet	None
Portlet - Liferay MVC	Upgrading a Liferay MVC Portlet	- Migrate to Workspace/Gradle- Apply Clay- Convert to OSGi modules
Portlet - Servlet/JSP	Upgrading a Servlet-based Portlet	None

Plugin	Upgrade path	Optimizations (optional)
Portlet - Spring MVC	Upgrading a Spring MVC Portlet	None
Portlet - Struts 1	Upgrading a Struts Portlet	Converting StrustActionWrappers to MVCCommands
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
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.

6.2 Upgrading Code 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 DXP 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.

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 1 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!

6.3 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.

6.4 Migrating Plugins SDK Projects to Liferay Workspace

The Plugins SDK was deprecated for Liferay DXP 7.0 and removed for Liferay DXP 7.1. Therefore, to upgrade your custom apps to 7.0, you must 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.

6.5 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.

6.6 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.

6.7 Resolving a Plugin's Dependencies

Now that you've imported your plugin project to Liferay Dev Studio DXP, 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 Dev Studio DXP. The Upgrade Planner reports each moved class for you to address one by one. Organizing imports in Dev Studio DXP automatically resolves multiple classes at once.

   It's typically faster to resolve moved classes using the mentioned Eclipse feature. Since Liferay Dev Studio DXP 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, Dev Studio DXP 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.

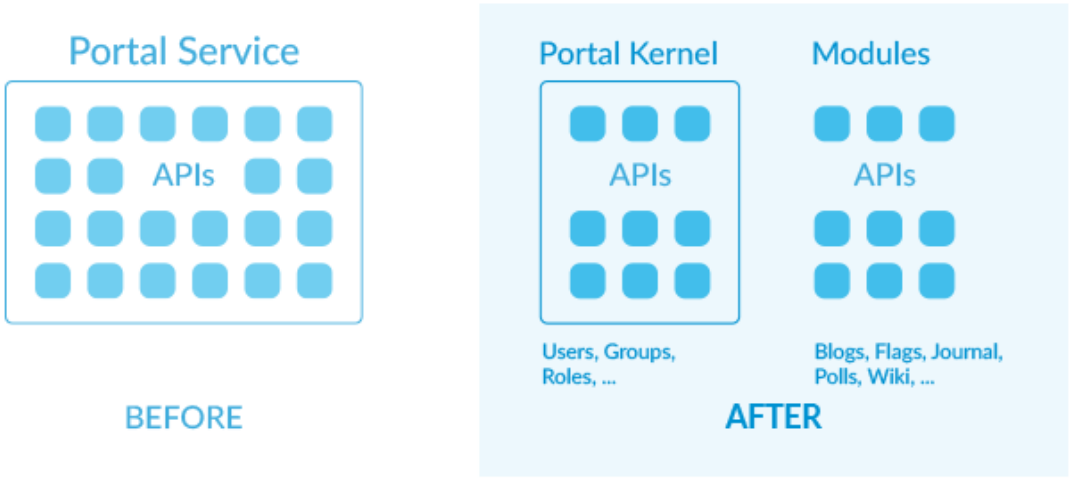


Figure 6.1: Liferay refactored the portal-service JAR for 7.0. Application APIs now exist in their own modules, and the portal-service JAR is now *portal-kernel*.

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 formerly known 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 Liferay Portal 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 Dev Studio DXP

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 Dev Studio DXP 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" revis
    </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 Plugin 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 Portal 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. This means that Liferay DXP can't make available to plugins the same Java classes it did in the past.

These files list the packages that the portal exports:

- `modules/core/portal-bootstrap/system.packages.extra.bnd` file in the GitHub repository. It lists exported packages on separate lines, making them easy to read.
- `META-INF/system.packages.extra.mf` file in `[LIFERAY_HOME]/osgi/core/com.liferay.portal.bootstrap.jar`. The file is available in Liferay DXP bundles. It lists exported packages in a paragraph wrapped at 70 columns–they're harder to read here than in the `system.packages.extra.bnd` file.

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 Portal 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.

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`:

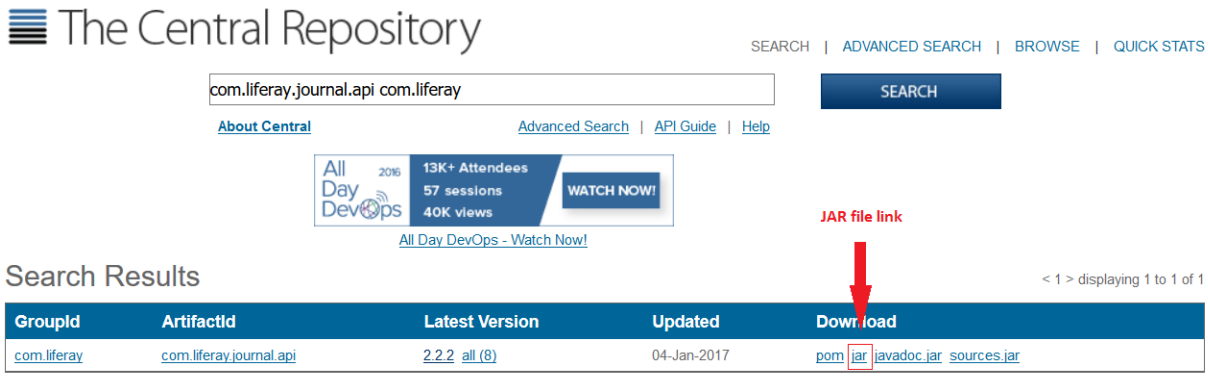


Figure 6.2: After searching Maven Central, download an artifact's JAR file by clicking the *jar* link.

```
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.

6.8 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.

- Liferay DXP 7.0 Breaking Changes

- 7.0 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 7

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!

7.1 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!

7.2 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

Customizing JSPs
Upgrading App JSP Hooks
Resolving a Plugin's Dependencies
Upgrading the Liferay Maven Build

7.3 Upgrading App JSP Hooks

JSPs in OSGi modules can be 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 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]` **7.1** | `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 7.1 JSP with your custom 6.2 JSP.

Make any additional needed changes in your 7.1 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_
19:23:12,910 INFO  [Refresh Thread: Equinox Container: 00ce6547-2355-0017-1884-846599e789c4][BundleStartStopLogger:35] STARTED com.liferay.blogs.web_
```

Your custom JSP is live.

Related Topics

Customizing JSPs

Upgrading Core JSP Hooks

Resolving a Plugin's Dependencies

Upgrading the Liferay Maven Build

7.4 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

7.5 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

7.6 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

7.7 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
Configuring Dependencies
Upgrading the Liferay Maven Build

7.8 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
Configuring Dependencies
Upgrading the Liferay Maven Build

7.9 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 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 Dev Studio DXP
Resolving a Plugin's Dependencies
Configuring Dependencies
Upgrading the Liferay Maven Build

7.10 Converting StrutsAction Wrappers to MVCCommands

In Liferay Portal 6.1 and 6.2, developers could customize the Portal and Portlet Struts Actions using a Hook and StrutsAction wrapper. 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.

Since 7.0, this mechanism no longer applies for most portlets because they no longer use Struts Actions, but instead use Liferay MVCCommands.

This tutorial demonstrates how to convert your existing StrutsAction wrappers to MVCCommands.

Converting Your old wrapper to MVCCommands

Converting StrutsAction wrappers to MVCCommands is easier than you may think.

As a review, legacy StrutsAction wrappers implemented all 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 StrutsAction wrapper, you need to use the matching MVCCommand interface shown below:

- processAction → MVCActionCommand
- render → MVCRenderCommand
- serveResource → MVCResourceCommand

Look at the ExampleStrutsPortletAction class for a StrutsAction wrapper example. Depending on the actions overridden, the user must use different MVCCommands. In this example, the action and render were overridden, so to migrate to the new pattern, you would create two classes: an MVCActionCommand and MVCRenderCommand.

Next you'll determine the mapping the MVCCommand uses.

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 DXP's source code to determine the correct mapping.

Map to your MVCCommand URLs using portlet URL tags:

- MVCRenderCommand URLs go in mvcRenderCommandName parameters. 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>
```

- `MVCActionCommand` URLs go in `actionURL` tag `name` attributes or in a parameter `ActionRequest.ACTION_NAME`. For example:

```
<portlet:actionURL name="/blogs/edit_entry" var="editEntryURL" />
```

- `MVCResourceCommand` URLs go in `resourceURL` tag `id` attributes. 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 `MVCCommand` tutorials:

- Adding Logic to MVCCommands
- Overriding MVCRenderCommands
- Overriding MVCActionCommands
- Overriding MVCResourceCommands

Now you know how to convert your `StrutsActionWrappers` to `MVCCommands`!

Related Topics

Overriding MVC Commands
    Liferay MVC Portlet
    Resolving a Plugin's Dependencies
    Configuring Dependencies
    Upgrading the Liferay Maven Build

CHAPTER 8

UPGRADING 6.2 THEMES

If you've developed themes in Liferay DXP 6.2, as part of your upgrade you'll want to use them in 7.0. The upgrade process requires several modifications. The Liferay Theme Generator helps automate this process.

The following tutorials show you how to upgrade your Liferay Portal 6.2 themes to 7.0:

- Upgrading 6.2 Themes to 7.1

8.1 Upgrading 6.2 Layout Templates

If you've developed layout templates in Liferay DXP 6.2, you can upgrade them for 7.0. The upgrade process requires minimal changes.

The following tutorial shows you how to upgrade your Liferay Portal 6.2 layout templates to 7.0:

- Upgrading 6.2 Layout Templates to 7.1

8.2 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.

8.3 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.

8.4 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.

8.5 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/excep`
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!

8.6 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 9

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
- Liferay MVC Portlet
- Portlet that uses Service Builder
- Liferay JSF Portlet
- Servlet-based 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!

9.1 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 Liferay 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`).

Figure 9.1: 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 Liferay 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:

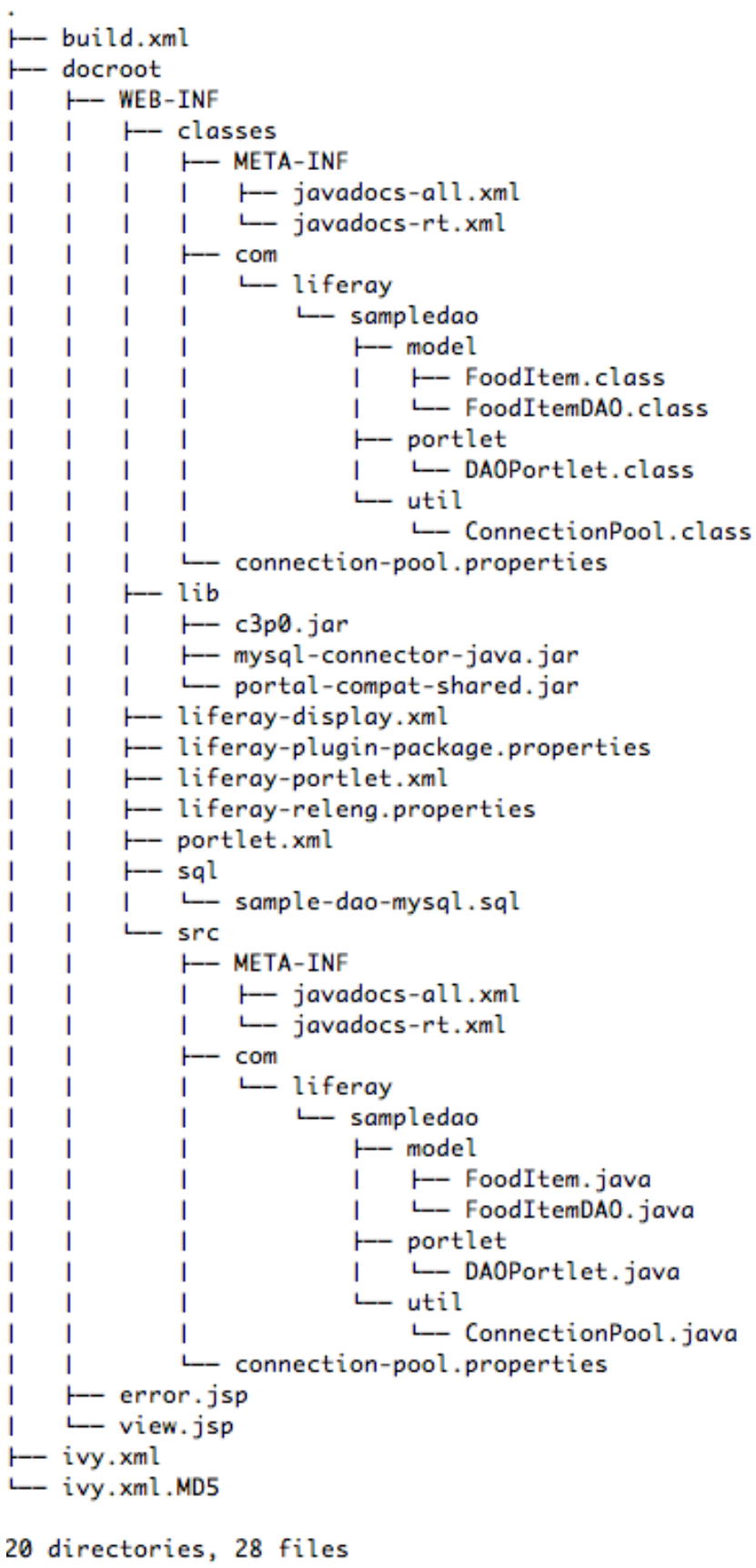


Figure 9.2: The `sample-dao-portlet` project uses a typical Plugins SDK portlet folder structure

- WAR processing
- WAB startup
- Availability to users

Deploying the sample portlet produces messages like these:

```
2018-03-21 17:44:59.179 INFO  [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:262] Processing sample-
dao-portlet-7.1.0.1.war
...
2018-03-21 17:45:09.959 INFO  [Refresh Thread: Equinox Container: 0012cbb0-7e2c-0018-146e-95a4d71cdf95][PortletHotDeployListener:298] 1 portlet for s
dao-portlet is available for use
...
2018-03-21 17:45:10.151 INFO  [Refresh Thread: Equinox Container: 0012cbb0-7e2c-0018-146e-95a4d71cdf95][BundleStartStopLogger:35] STARTED sample-
dao-portlet_7.1.0.1 [655]
```

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

9.2 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.

To upgrade a Liferay MVC Portlet, 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.

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 you upgrade 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

9.3 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.

Figure 9.3: The Guestbook portlet models guestbooks and guestbook entries.

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
- 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 Liferay 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/excep` 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:

```
2018-03-21 18:23:10.032 INFO  [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:262] Processing guestbook-
portlet-7.1.0.1.war
2018-03-21 18:23:15.355 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][BaseAutoDeployListener:43] Copying portlets for C:\portals\l
ce-portal-7.1-m1\tomcat-8.0.32\temp\20180321182315333UGEPAGTR\guestbook-portlet.war
2018-03-21 18:23:15.829 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][BaseDeployer:876] Deploying guestbook-
portlet.war
2018-03-21 18:23:17.797 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][BaseAutoDeployListener:50] Portlets for C:\portals\liferay-
ce-portal-7.1-m1\tomcat-8.0.32\temp\20180321182315333UGEPAGTR\guestbook-portlet.war copied successfully
2018-03-21 18:23:19.621 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][HotDeployImpl:226] Deploying guestbook-
portlet from queue
2018-03-21 18:23:19.621 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][PluginPackageUtil:1003] Reading plugin package for guestbook
portlet
2018-03-21 18:23:19.642 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][ServiceComponentLocalServiceImpl:598] Running GB SQL scripts
21-Mar-2018 18:23:19.669 INFO [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war] org.apache.catalina.core.ApplicationContext.log Initializing
2018-03-21 18:23:20.066 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][PortletHotDeployListener:186] Registering portlets for guest
portlet
2018-03-21 18:23:20.271 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][PortletHotDeployListener:298] 1 portlet for guestbook-
portlet is available for use
2018-03-21 18:23:20.468 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][BundleStartStopLogger:35] STARTED guestbook-
portlet_7.1.0.1 [657]
```

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

9.4 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.

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.

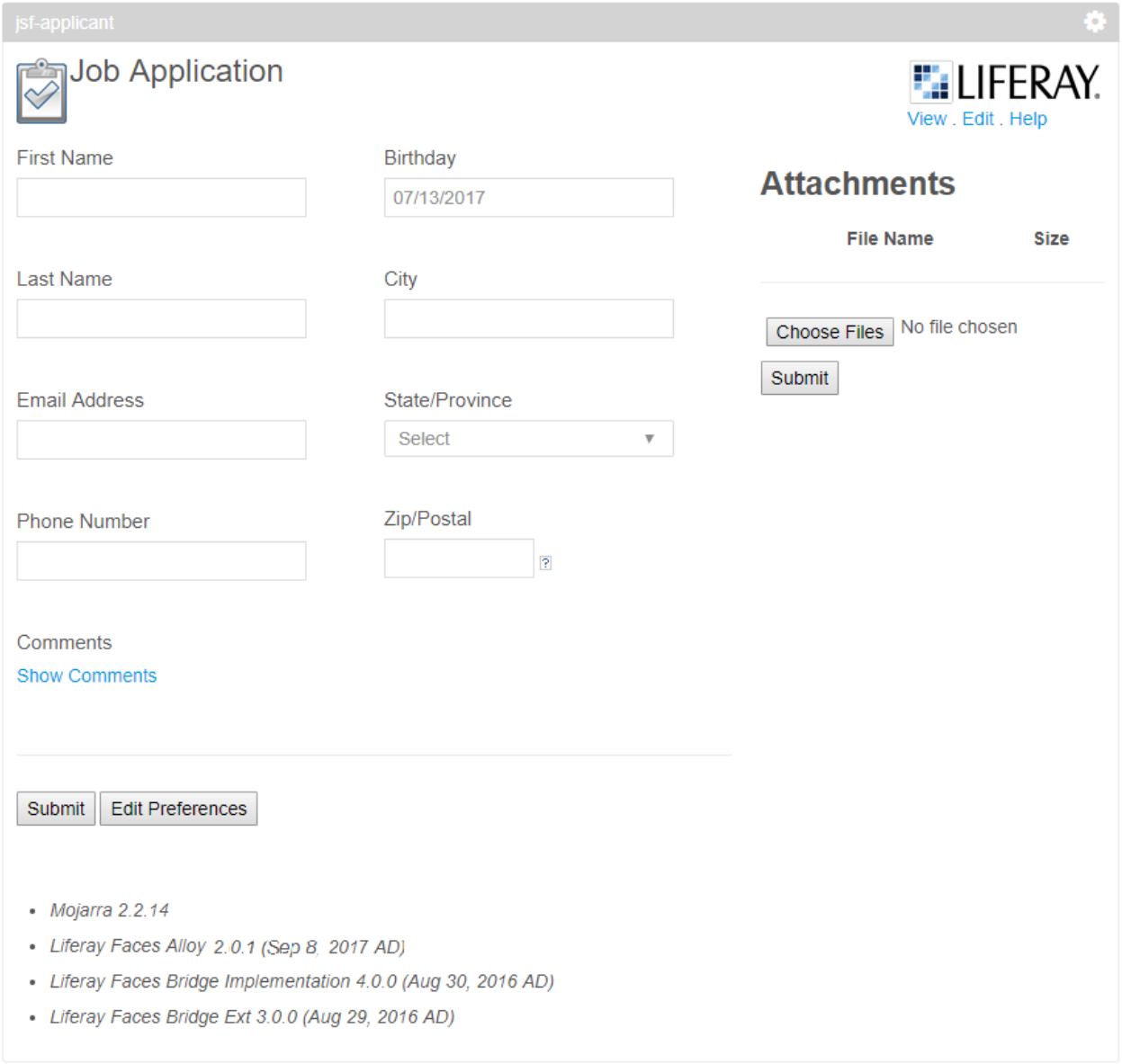


Figure 9.4: The JSF Applicant portlet provides a job application for users to submit.

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.

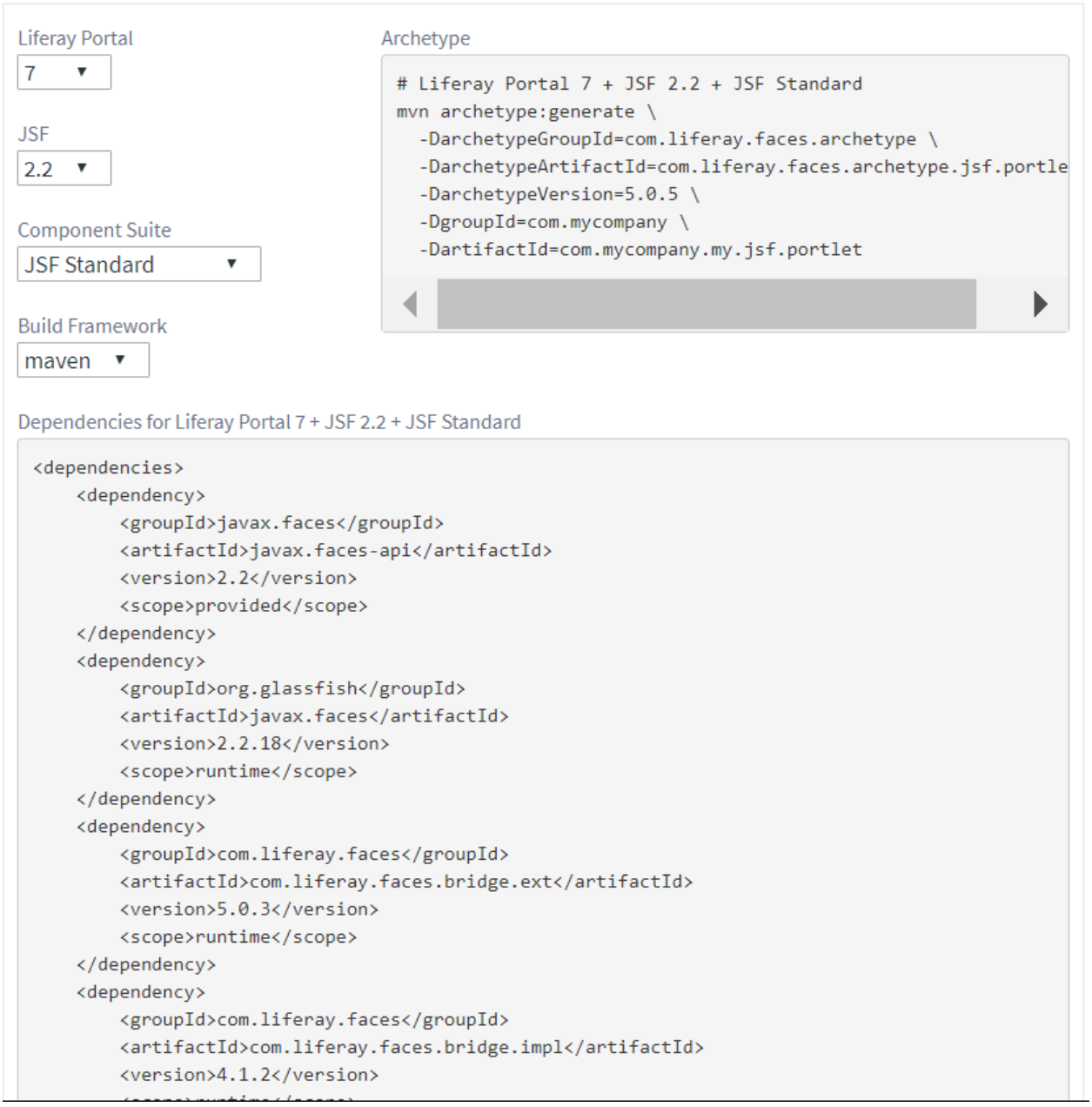


Figure 9.5: The Liferay Faces site gives you options to generate dependencies for many environments.

3. Compare the generated dependencies with your portlet's dependencies and make any necessary updates. For the sample JSF Applicant portlet, the Mojarra dependency and two Liferay Faces dependencies require updating:

```
<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>3.0.0</version>
</dependency>
<dependency>
    <groupId>com.liferay.faces</groupId>
    <artifactId>com.liferay.faces.bridge.impl</artifactId>
    <version>4.0.0</version>
</dependency>
```

Update the dependencies according to the http://liferayfaces.org/ dependency list. For example,

```
<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.faces</artifactId>
    <version>2.2.18</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>com.liferay.faces</groupId>
    <artifactId>com.liferay.faces.bridge.ext</artifactId>
    <version>5.0.3</version>
</dependency>
<dependency>
    <groupId>com.liferay.faces</groupId>
    <artifactId>com.liferay.faces.bridge.impl</artifactId>
    <version>4.1.2</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.po
1.0.war
...
13:42:03,522 INFO  [fileinstall-C:/liferay-ce-portal-7.1-ga1/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-7.1-ga1/osgi/war][PortletHotDeployListener:293] 1 portlet for com.liferay.faces.demo.jsf.applica
1.0 is available for use
```

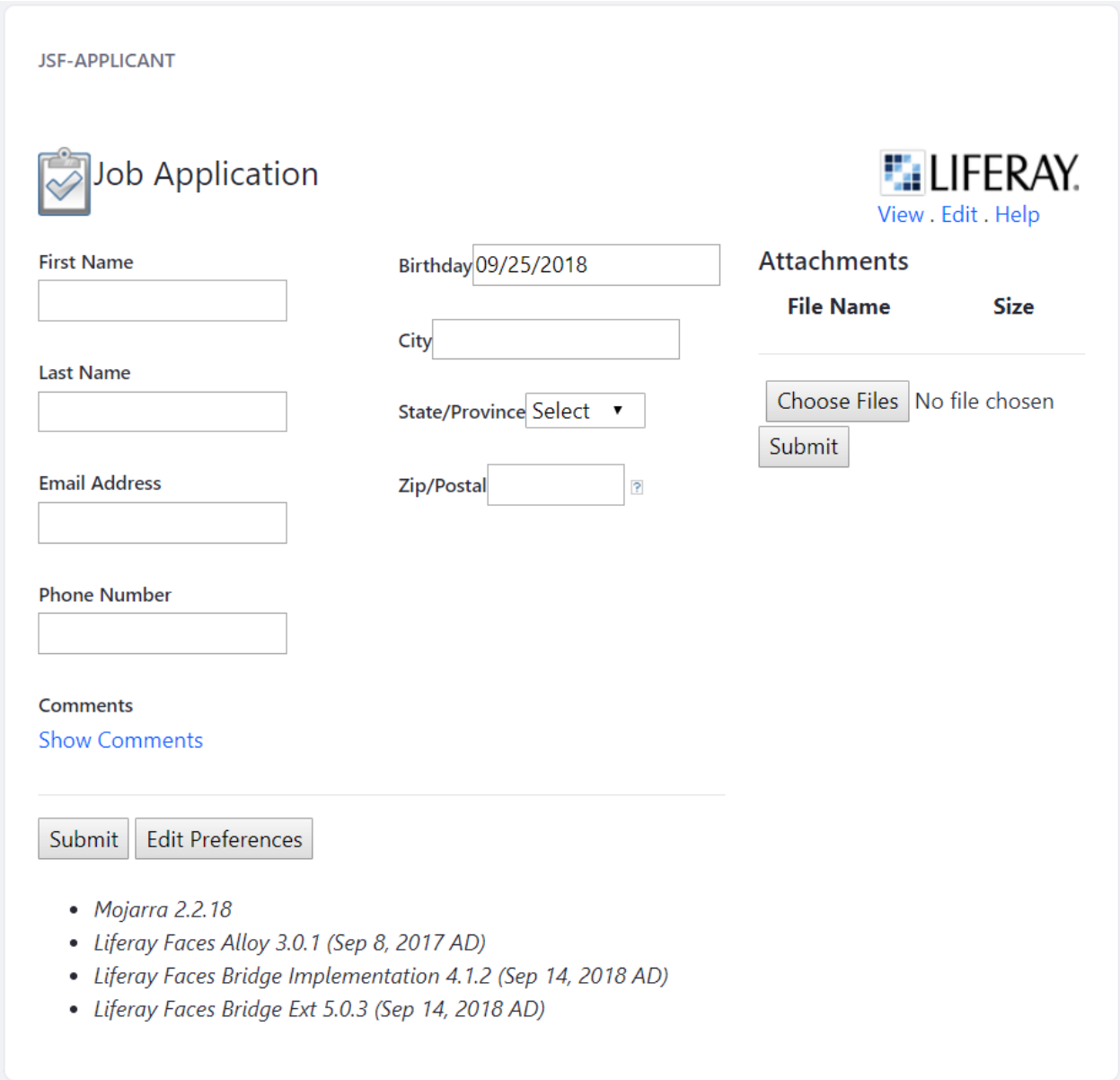


Figure 9.6: You've successfully updated the JSF Applicant portlet for 7.0!

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!

9.5 Upgrading a Servlet-based Portlet

This tutorial shows you how to upgrade servlet-based portlets. It 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.

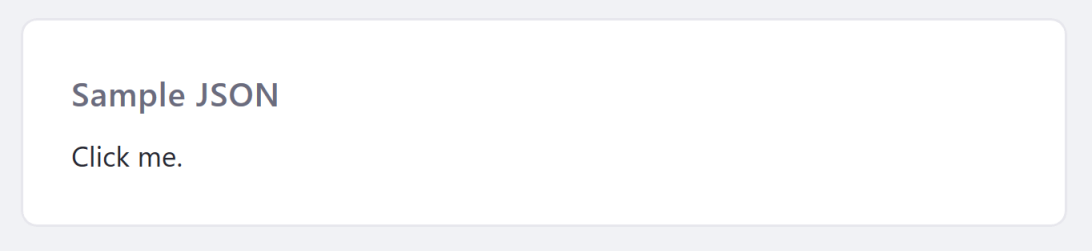


Figure 9.7: The Sample JSON portlet displays text stating *Click me* that you can click to initiate an action.

To get the most from this tutorial, 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

To upgrade a servlet-base Portlet, 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.

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.

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 keep compile time packages from 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 tl
```

For running on 7.0, it's fine to specify the`portal-dependency-jars` property per the instructions for using packages that Liferay DXP 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's OSGi runtime and is available to users.

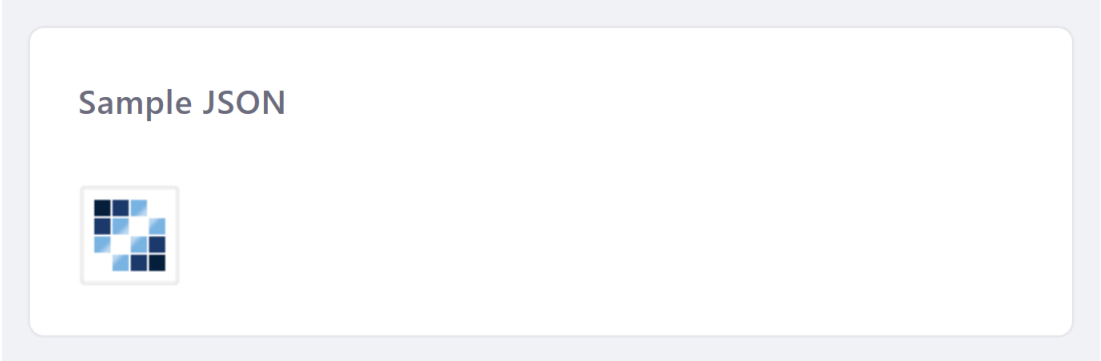


Figure 9.8: Clicking on the sample portlet's *Click me* link shows the Liferay logo.

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

9.6 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.

Figure 9.9: 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.

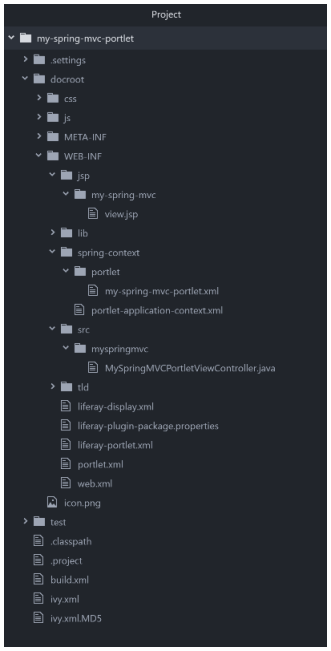


Figure 9.10: The `my-spring-mvc-portlet` project has traditional Liferay plugin files, Spring Portlet MVC application contexts (in `spring-context/`), and a controller class `MySpringMVCPortletviewController`.

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:

1. Adapt the code to 7.0's API

2. Resolve dependencies

Adapt the code to liferay's API

The Liferay 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 Deploying 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.

Depploying 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.

```
2018-04-12 19:28:36.810 INFO  [com.liferay.portal.kernel.deploy.auto.AutoDeployScanner][AutoDeployDir:262] Processing my-spring-
mvc-portlet.war
2018-04-12 19:28:42.182 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][BaseAutoDeployListener:43] Copying portlets for C:\portals\l
ce-portal-7.1-m1\tomcat-8.0.32\temp\20180412192842100ZSINUETA\my-spring-mvc-portlet.war
2018-04-12 19:28:42.706 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][BaseDeployer:876] Deploying my-spring-
mvc-portlet.war
2018-04-12 19:28:47.708 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][BaseAutoDeployListener:50] Portlets for C:\portals\liferay-
ce-portal-7.1-m1\tomcat-8.0.32\temp\20180412192842100ZSINUETA\my-spring-mvc-portlet.war copied successfully
2018-04-12 19:28:56.600 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][HotDeployImpl:226] Deploying my-spring-
mvc-portlet from queue
2018-04-12 19:28:56.601 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][PluginPackageUtil:1003] Reading plugin package for my-
spring-mvc-portlet
2018-04-12 19:28:56.700 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][PortletHotDeployListener:186] Registering portlets for my-
spring-mvc-portlet
2018-04-12 19:28:56.955 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][PortletHotDeployListener:298] 1 portlet for my-
spring-mvc-portlet is available for use
2018-04-12 19:28:57.114 INFO  [fileinstall-C:/portals/liferay-ce-portal-7.1-m1/osgi/war][BundleStartStopLogger:35] STARTED my-
spring-mvc-portlet_7.1.0.1 [658]
```

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

9.7 Upgrading a Struts 1 Portlet

Struts is a stable, widely adopted framework that implements the Model View Controller (MVC) design pattern. If you have a Struts portlet for previous versions of Liferay Portal, 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.

This tutorial demonstrates how to upgrade a portlet that uses the Struts 1 Framework.

Here's a sample Struts portlet's folder structure with file/folder descriptions:

- `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 1 portlet involves these steps:

1. Adapt the portlet 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

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
```

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 server prints messages indicating the following portlet status:

- WAR processing
- WAB startup
- Availability to users

The Struts portlet is now available on your Liferay DXP instance. The Struts portlet behaves just as it did on previous versions on your 7.0 site.

Congratulations on upgrading your Struts portlet to 7.0!

9.8 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!

9.9 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.

9.10 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 Liferay Upgrade Planner.

Liferay also offers a Maven development environment tailored specifically for 7.0 development. Learn more about this in the Maven Workspace tutorial.

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
Bundle Support	com.liferay.portal.tools.bundle.support	Coming Soon
CSS Builder	com.liferay.css.builder	Compiling SASS Files in a Maven Project
DB Support	com.liferay.portal.tools.db.support	DB Support Plugin
Deployment Helper	com.liferay.deployment.helper	Coming Soon
Javadoc Formatter	com.liferay.javadoc.formatter	Coming Soon
Lang Builder	com.liferay.lang.builder	Coming Soon
Service Builder	com.liferay.portal.tools.service.builder	Service Builder Plugin
Source Formatter	com.liferay.source.formatter	Source Formatter Plugin
Theme Builder	com.liferay.portal.tools.theme.builder	Theme Builder Plugin
TLD Formatter	com.liferay.tld.formatter	Coming Soon
WSDD Builder	com.liferay.portal.tools.wsdd.builder	Coming Soon
XML Formatter	com.liferay.xml.formatter	Coming Soon

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 10

OPTIMIZING PLUGINS FOR 7.0

Once you've upgraded your plugin, you can optimize it to take advantage of all 7.0 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 these 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 Liferay's modular architecture and updated frameworks, such as Service Builder.
- Styling your app consistently using Clay –the web implementation of Lexicon, Liferay's design language.
- Modularizing apps to reap the benefits of modularity.

10.1 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 7.0. The Plugins SDK, however, is no longer available for 7.0. Visit the Deprecated Apps in 7.1: What To Do article for more information on the Plugins SDK removal. 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 Liferay DXP 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

From the command line, 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: Executing `blade convert -l` lists Plugins SDK projects that can be migrated to Workspace web apps. Run `blade convert --all` to migrate all the plugin projects.

The image below shows the plugin files before and after they're migrated to Workspace web apps.

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!

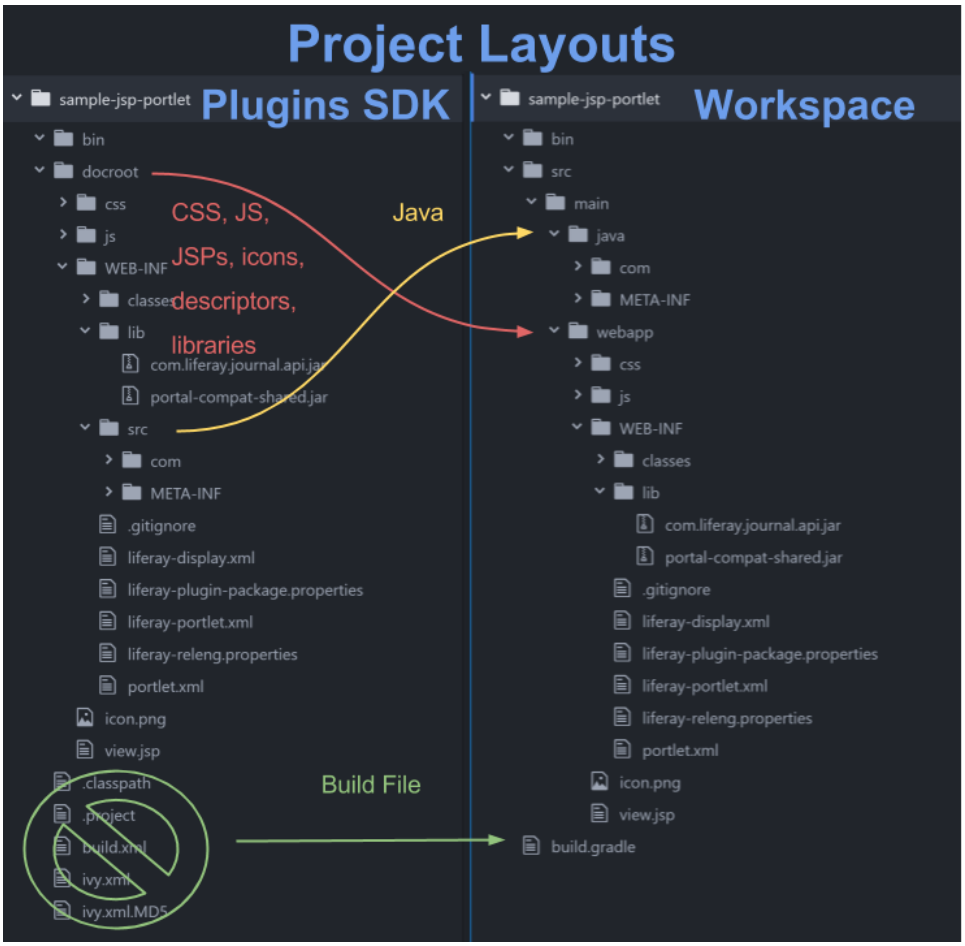


Figure 10.1: The `convert` command migrates a Plugins SDK project to a Workspace web application project. It moves Java source files to `src/main/java` and all other files/folders to `src/main/webapp`.

Related Topics

Workspace Development lifecycle
    Workspace Gradle Tasks

CHAPTER 11

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, Java package dependencies are explicitly listed within a module. Modules run only when all their dependencies are met. This eliminates 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 working on 7.0.

When to convert?

- You have a very large application with many lines of code. If you have 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, your application has 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 modularity.

CHAPTER 12

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 these tutorials, 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 these tutorials is just one of many ways for structuring your application's modules. Also applications come in all different shapes and sizes. There may be special actions that some applications require. These tutorials provide 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 apply to portlets for both Liferay DXP Digital Enterprise and Liferay Portal CE.

The first thing you'll do is create your application's web (client) module.

12.1 Converting Your Application's Portlet Classes and UI

The first thing you'll do is create your application's root folder and the folder structure for its *web* client module. This module holds portlet classes and the web UI. Before you start creating a skeleton structure for the modules, determine the modules that comprise this version of your application. If your application provides service API and implementation classes (which is the case for all Liferay Service Builder applications), you'll create separate modules for them. This tutorial assumes the Maven project model, although any build tools or folder structure is permissible.

Note: You should 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:3.2.0 **or**
- org.dm.gradle:gradle-bundle-plugin:0.9.0

Maven

- biz.aQute.bnd:bnd-maven-plugin:3.2.0

Here are the steps for creating the folder structure:

1. Create the root folder. It is the new home for your application's independent modules and configuration files. For example, if your application's name is *Tasks*, then your root 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. Blade CLI and Maven generate project folder structures based on project templates.

   For example, navigate to the root folder (e.g., `tasks`) and run the following Blade CLI command to generate a generic web client module structure:

   ```
   blade create -t mvc-portlet [APPLICATION_NAME]-web
   ```

3. In your `*-web` module, replace the `/src/main/java/[APPLICATION_NAME]` folder with your root Java package. For example, if your application's root package name is `com.liferay.tasks.web`, your class's folder should be `/src/main/java/com/liferay/tasks/web`. Also, remove the `init.jsp` and `view.jsp` files from the `src/main/resources/META-INF/resources` folder. You'll use your existing application's JSPs instead of these generated default JSPs.

4. Verify that your `*-web` module folder resides in your application's root folder (marked by `[APPLICATION_NAME]` below)'s and your `*-web` module's folder structure looks like this:

   - `[APPLICATION_NAME]`
     - `[APPLICATION_NAME]-web`
       * `src`
         - `main`
         - `java`
         - `[ROOT_PACKAGE]`
         - `resources`

        - content
        - Language.properties

        - META-INF
        - resources

    * bnd.bnd
    * build.gradle

The remaining steps affect the web client module (`*-web` module).

5. The `bnd.bnd` file is used to generate your module's `MANIFEST.MF` file when you build your project. Open it and change it to fit your application. There's further documentation about configuring your module's `bnd.bnd`. For example, here's the Liferay `dictionary-web` module's `bnd.bnd`:

   ```
   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. Specify all your module's dependencies here. The one generated for you is pre-populated with content and default dependencies. Add your module's dependencies in the `dependencies {...}` block.

   Finding Liferay API Modules lists common Liferay API module's symbolic names. The Javadoc overviews for 7.0 and Liferay apps list each module's symbolic name and version. The Configuring Dependencies tutorial demonstrates finding artifact information and specifying dependencies. Liferay DXP provides many Java packages and entire artifacts at runtime in the OSGi container. Your module is activated after installation once all its dependencies resolve. Unresolved dependencies appear in the log. Here's an example message:

   ```
   ! could not resolve the bundles: ... Unresolved requirement: Import-Package: ... Unresolved requirement: Require-Capability ...
   ```

7. Copy your traditional application's JSP files into the `/src/main/resources/META-INF/resources` folder. In most cases, all of your application's JSP files belong in the web client module.

8. Copy your portlet classes and supporting classes that aren't related to Service Builder into their respective package folders in the web client module. Organizing classes into sub-packages can make them easier to manage.

   For example, here's the `journal-web` module's Java source folder structure:

- 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 API and implementation classes. These classes belong in API and implementation modules. The next tutorial demonstrates copying those classes into modules.

---

9. Now that the necessary classes are in your client module, you must make them comply with OSGi. If you're a beginner, we recommend using the Declarative Services component framework because Liferay DXP uses it. This tutorial assumes that you're using Declarative Services. You can, however, use any other OSGi component framework.

   Review your traditional application's XML files and migrate the configuration and metadata information to the portlet class as component properties. You can do this by adding the `@Component` annotation to your portlet class and adding the necessary properties to that annotation. Examine the mapping of the portlet descriptors to component properties. 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. Migrate your traditional application's resource actions (if it has any), 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, see the Directory application's `default.xml`.

    Note that the permissions API has changed in 7.1; adapt your permissions helpers accordingly.

12. Add your language keys to the `src/main/resources/content/Language.properties` file. Only include the language keys unique to your application. Liferay DXP's language keys are available to all portlet applications.

Awesome! You've created your application's web client module and completed some of the most common tasks for modularizing your portlet classes and UI. To convert other parts of your application this tutorial hasn't covered, examine the Liferay DXP developer tutorials to see how those parts fit into application modules. The tutorials are divided into popular areas so you can easily find the topical information you need.

Lastly, the table below is a quick reference guide that maps files and Java packages from a traditional portlet plugin to a module for a fictitious application called `tasks-portlet`.

Traditional Plugin	Module
`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.

12.2 Converting Your Application's Service Builder API and Implementation

In this tutorial, you'll learn about converting a Liferay Portal 6 Service Builder application to a 7.0 style application. In the previous tutorial, 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 tutorial, 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 must 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. 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, copy the module into the project generated by the `service-builder` template. 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 folder should now be in good shape. Next, use Service Builder to generate your application's service API and service implementation code.

1. Copy your traditional application's `service.xml` file into the implementation module's root folder (e.g., `tasks/tasks-service`).

2. Blade CLI generated a `bnd.bnd` file for your service implementation module. 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 generates your API module successfully, but extra configuration might be necessary in certain cases.

5. Navigate to your root project folder. Then run `gradlew buildService`.

   Your service API, implementation classes, and configuration (SQL, Hibernate, Spring, etc.) are generated from your `service.xml` file in their respective modules. The Service Builder Gradle Plugin has multiple options.

6. Now that you've run Service Builder, copy your business logic classes into your implementation module. The table below highlights popular Liferay Portal 6 classes and packages and where to place them in your application. This table suggests how to organize 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' folder 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
 `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 your business logic, run `gradlew buildService` again to generate the remaining services.

Now that your services are generated, you must 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 root project's `settings.gradle` file, 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.

12.3 Building Your Application's Module JARs for Deployment

Now it's time to build and deploy your modules. To build your project, run `gradlew build` from your application project's root folder.

Now check all of your modules' `/build/libs` folders. There should be a newly generated JAR file in each, which is the file to deploy. You can deploy each JAR by running `blade deploy` from each module's root folder.

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, errors occur if it relies on the service implementation and service API modules. Once each module's dependencies are met, they deploy in Liferay. Felix Gogo Shell commands let you check module dependencies.

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 12.1: Once you've connected to your Liferay instance in your Gogo shell prompt, run *lb* to list your new converted modules.

These tutorials explained how to convert your traditional application into the modular format of a 7.0 style application. 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

12.4 Migrating Data Upgrade Processes to the New Framework for Modules

When you make database changes to your application, you must use a *data upgrade process* to migrate users' existing data to the new database schema. While the old framework required several classes, the new framework can 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. This tutorial shows you how to migrate to the new 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.

Before beginning, make sure you know how to create an upgrade process that uses the new framework.

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.

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, 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? First, the upgrade step classes are the same, so you can leave them unchanged. Here are the big changes in the new upgrade processes:

- A single registrator class replaces upgrade process classes.
- Service Builder services require a Bundle Activator.

Start your conversion by creating a registrator class.

Create a Registrator Class

The new data upgrade framework requires using registrator class instead of upgrade process classes. 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.

Next, create a Bundle Activator if your modularized plugin uses Service Builder.

Create a Bundle Activator

If your module implements Service Builder services, it needs a Bundle Activator to initialize a record in the release table. Creating a Bundle Activator is straightforward.

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

Chapter 13

From Liferay DXP 7.0 to 7.1

7.0 offers new and improved frameworks and APIs that make your plugins faster, more secure, and easier to maintain. Upgrading plugins from Liferay DXP 7.0 to 7.1 is easier than ever too. These tutorials show you how:

- Upgrading Plugins
- Upgrading Themes
- Upgrading Layout Templates

Upgrading plugins (portlets, customizations, and extensions) is up first.

CHAPTER 14

UPGRADING PLUGINS FROM LIFERAY DXP 7.0 TO 7.1

Liferay Workspace's Target Platform feature and Upgrade Planner take a lot of manual intervention out of upgrading to 7.0. Target Platform facilitates updating dependencies to 7.0, and the Upgrade Planner helps you adapt plugins to 7.0's API. This tutorial explains the plugin upgrade steps.

Note: Blade CLI's `convert` command migrates traditional plugins to Liferay Workspace, so you can leverage Workspace's upgrade features.

Here are the plugin upgrade steps:

1. Update your Liferay Workspace to that latest version.
2. Update your Liferay DXP-related dependencies by setting your Target Platform to the latest version of 7.0. (Optional)
3. Update your plugin's remaining dependencies.
4. Adapt your code to 7.0's API using the Upgrade Planner. The Upgrade Tool shows you where breaking changes affect your code and addresses many of them automatically.

Congratulations! Your upgraded plugin is ready to deploy to 7.0.

Note: If your plugin resides outside of a Workspace, apply the Target Platform Gradle plugin to your project so you can set that project's Target Platform.

14.1 Related Topics

Liferay Workspace
    Managing Target Platforms for Workspace
    Configuring Dependencies
    Liferay Dev Studio

CHAPTER 15

UPGRADING 7.0 THEMES

If you've developed themes in Liferay DXP 7.0, as part of your upgrade you'll want to use them in 7.0. The upgrade process requires several modifications. The Liferay Theme Generator helps automate this process.

The following tutorials show you how to upgrade your Liferay Portal 7.0 themes to 7.0:

- Upgrading 7.0 Themes to 7.1

15.1 Upgrading 7.0 Layout Templates

If you've developed layout templates in Liferay DXP 7.0, you can upgrade them for 7.0. The upgrade process requires minimal changes.

The following tutorial shows you how to upgrade your Liferay Portal 7.0 layout templates to 7.0:

- Upgrading 7.0 Layout Templates to 7.1

CHAPTER 16

DEVELOPING A WEB APPLICATION

In this Learning Path, you'll create the Liferay Guestbook Web Application from scratch using tools like Liferay Dev Studio DXP 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. Completing this Learning Path prepares 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 Dev Studio DXP optimizes development on Liferay's platform. It integrates Liferay's Blade tools for modular development.

Once you set up your development environment, you can create 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!

16.1 Development Setup Overview

Liferay's development tools help you get started fast. The basic steps for installing Liferay Dev Studio DXP are

- Download a Liferay Dev Studio DXP bundle.
- Unzip the downloaded package to a location on your system.
- Start Dev Studio DXP.

You'll follow these steps and then generate an environment for developing your first Liferay DXP application.

Installing a Liferay Dev Studio DXP Bundle

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 Dev Studio DXP. The JDK is an enhanced version of the Java Environment used for developing new Java technology. Use JDK 8 or JDK 11.

2. Download and install Liferay Dev Studio DXP Installing it is easy: unzip it to a convenient location on your system.

3. To run Liferay Dev Studio DXP, run the `LiferayDeveloperStudio` executable.

The first time you start Liferay Dev Studio DXP, it prompts you to select an Eclipse workspace. If you specify an empty folder, Liferay Dev Studio DXP 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 Dev Studio DXP 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 Dev Studio DXP:

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. For *Liferay Version* select *7.1*.

5. Check the *Download Liferay bundle* checkbox to download and unzip a Liferay DXP instance in your workspace automatically. When prompted, name the server `liferay-tomcat-bundle`.

6. 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.

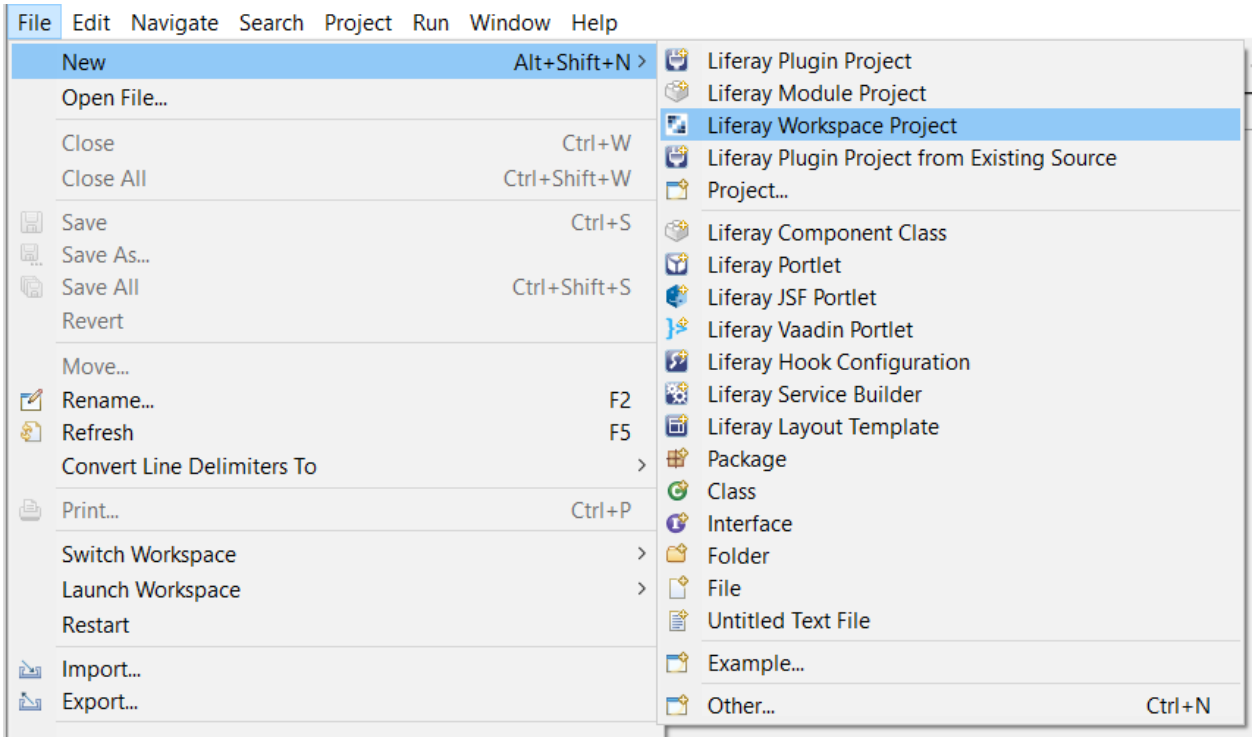


Figure 16.1: By selecting *Liferay Workspace*, you begin the process of creating a new workspace for your Liferay DXP projects.

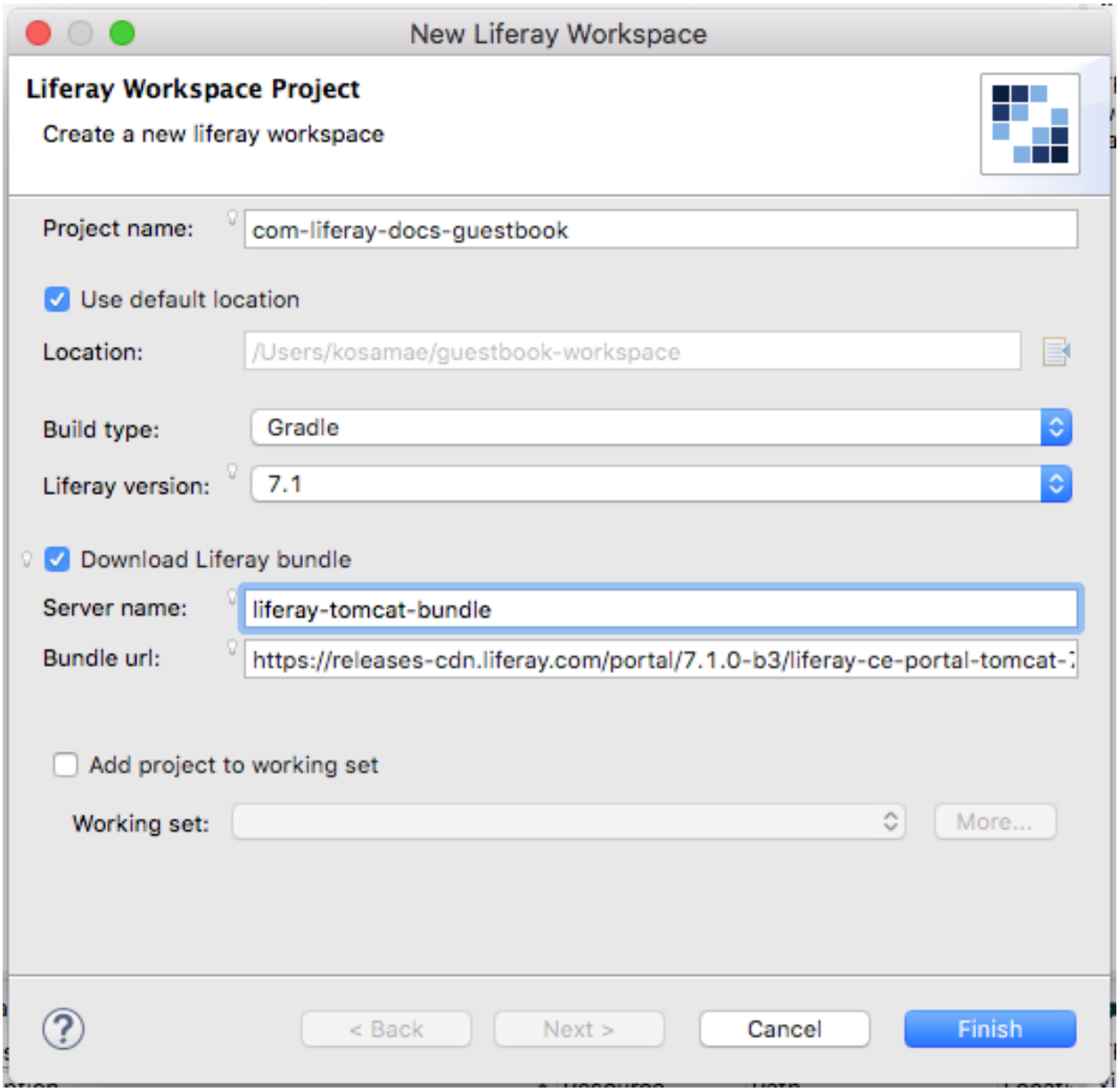


Figure 16.2: Liferay Dev Studio DXP provides an easy-to-follow menu to create your Liferay Workspace.

CHAPTER 17

CREATING A WORKING PROTOTYPE

So far, you've installed Liferay Dev Studio DXP and created a Liferay Workspace. Next, you'll create your application and start adding 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!

17.1 Writing Your First Liferay DXP Application

```
<p id="stepTitle">Developing Your First Portlet</p><p>Step 1 of 8</p>
```

Now 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 users leave on your website. To add a message, you click the *Add Entry* button to show a form for entering and saving 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. Multiple modules form a complete 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

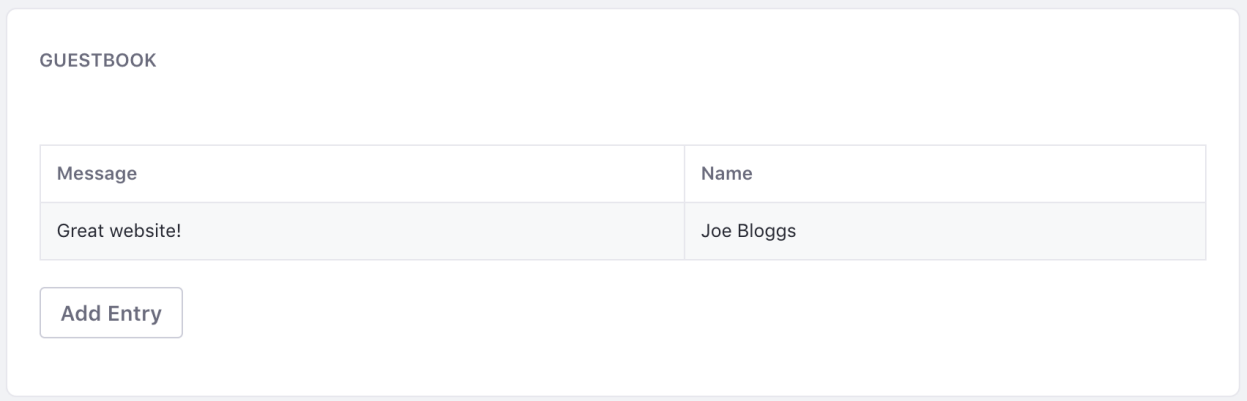


Figure 17.1: You'll create this simple application.

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 Dev Studio DXP, 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 Dev Studio DXP 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.

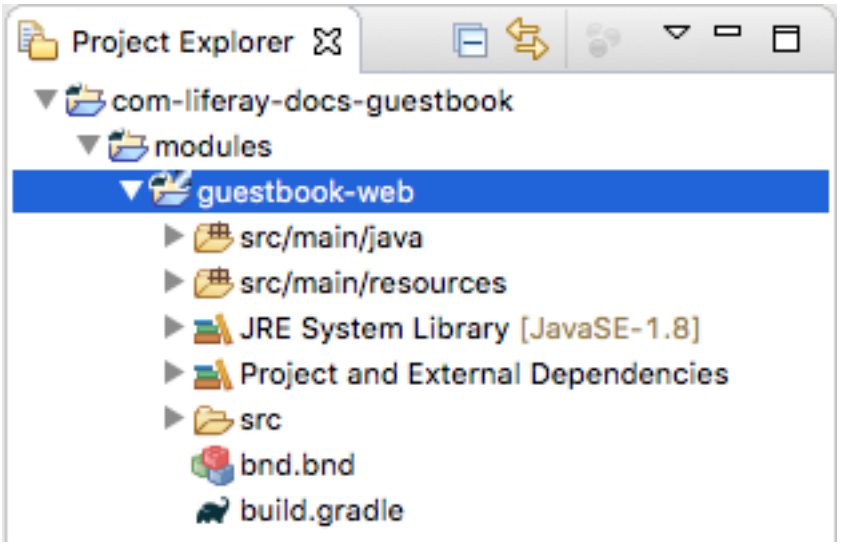


Figure 17.2: Your new module project appears in your Liferay Workspace's `modules` folder.

What is a Portlet?

Web applications can be simple: they might show you one piece of information, such as an article. A complex application might track your taxes as you enter 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, let you 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 a module (sometimes also called a *bundle*) encapsulates pieces of your application, a component is the object that contains the core functionality. A Component is managed by a component framework or container. 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 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 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.

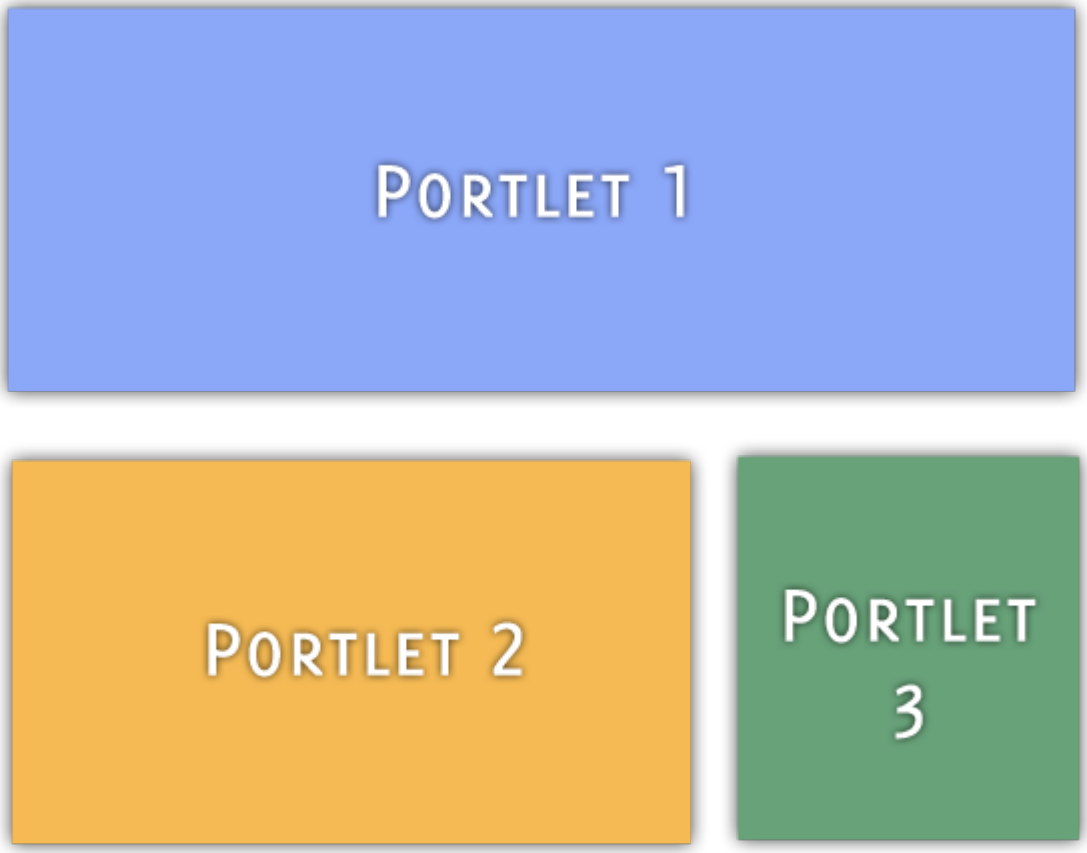


Figure 17.3: Many Liferay applications can run at the same time on the same page.

1. Make sure that your server is running, and if it isn't, select it in Dev Studio DXP's Servers pane and click the start button ( ).

2. After it starts, drag and drop the `guestbook-web` project from the Project Explorer to the server.

Figure 17.4: Drag and drop the module.

3. Open a browser and navigate to Liferay DXP (http://localhost:8080 by default).

   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.

4. To add an application to a page, click *Add* ( ) in the upper right hand corner.

5. Select *Widgets*. In the Applications list, your application should appear in the Sample category. Its name is `Guestbook`.

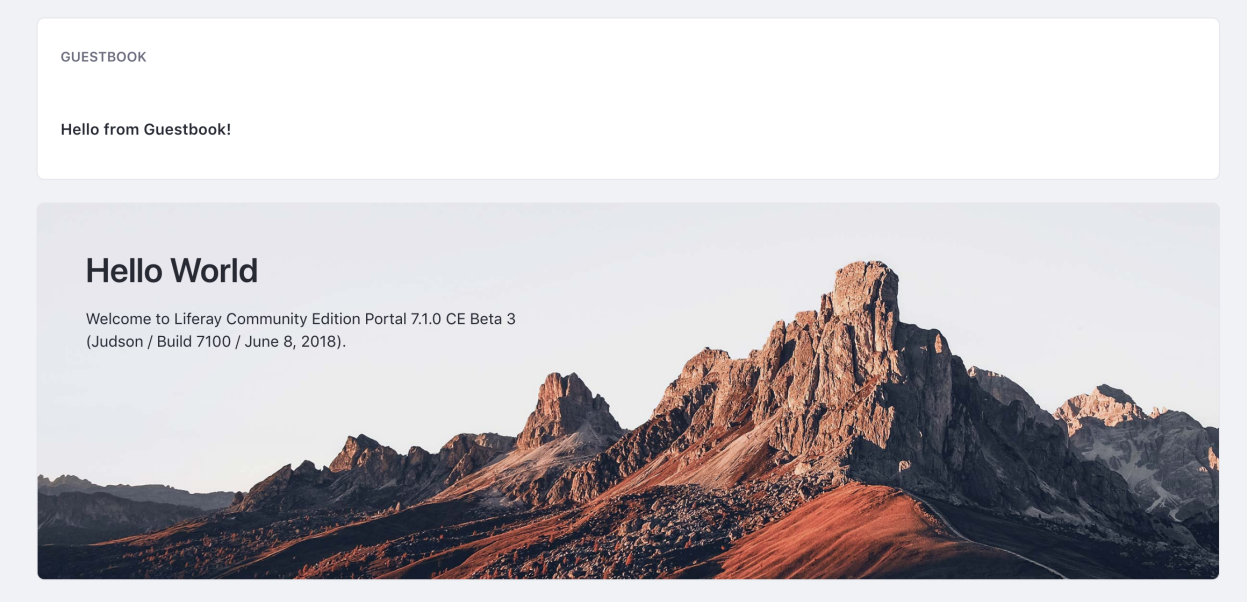


Figure 17.5: This is the default Liferay home page. It contains the Hello World widget and the initial version of the Guestbook application that you created.

Now you're ready to jump in and start developing your Guestbook portlet.

17.2 Creating an Add Entry Button

```
<p id="stepTitle">Developing Your First Portlet</p><p>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 contains the default view for users when the portlet is added to the page. Right now it contains 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 are 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.

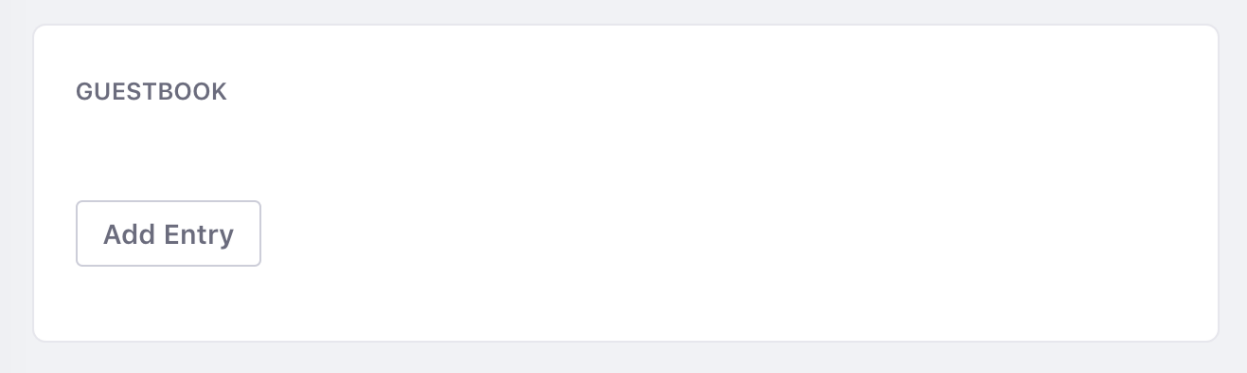


Figure 17.6: Your new button is awesome, but it doesn't work yet.

17.3 Generating Portlet URLs

```
<p id="stepTitle">Developing Your First Portlet</p><p>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 may share a page with yours. This means that you can't define URLs for various functions in your application like you otherwise would.

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 prevent this. If the system generates a unique URL for each piece of functionality, multiple applications can coexist in perfect harmony.

In `view.jsp`, follow these steps to create 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.jsp` is appended to the URL.

Note that your `GuestbookPortlet` class (located in your `guestbook-web` module's `com.liferay.docs.guestbook.portle` 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.

17.4 Linking to Another Page

```
<p id="stepTitle">Developing Your First Portlet</p><p>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, you must learn about portlet actions.

17.5 Triggering Portlet Actions

```
<p id="stepTitle">Developing Your First Portlet</p><p>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 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.

17.6 Creating a Form

```
<p id="stepTitle">Developing Your First Portlet</p><p>Step 6 of 8</p>
```

The form for creating guestbook entries has 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, you go back to `view.jsp`, but 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.

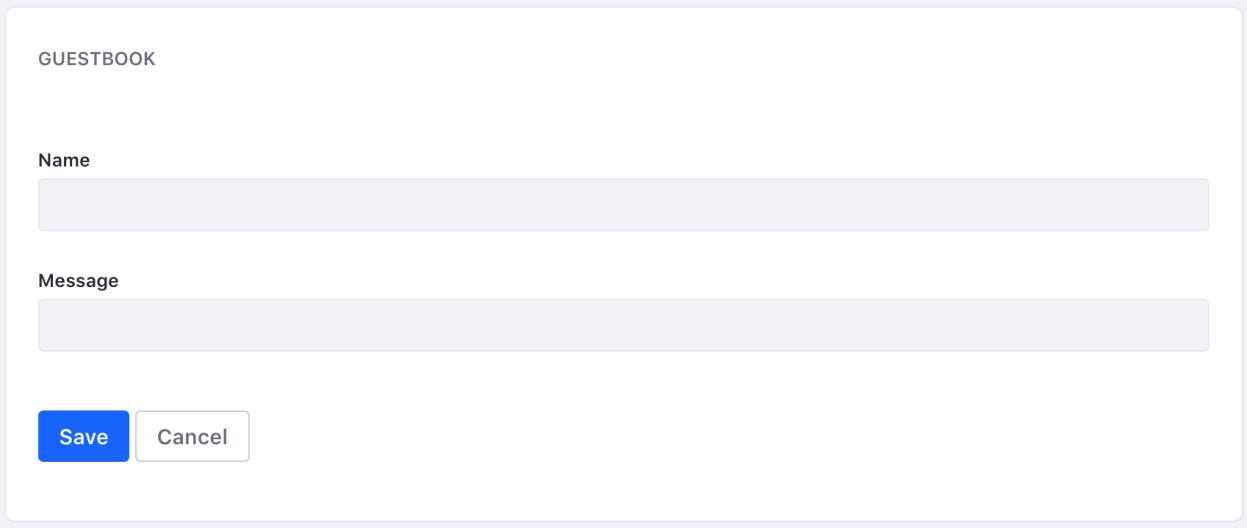


Figure 17.7: This is the Guestbook application's form for adding entries.

Implementing the portlet action (what happens when the user clicks *Save*) is your next task.

17.7 Implementing Portlet Actions

```
<p id="stepTitle">Developing Your First Portlet</p><p>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, and you'll refactor this into a database later. 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`, `java.util.logging.Logger`, and `java.util.logging.Level` 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. Notice how Liferay's `ParamUtil` class makes it 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.

17.8 Displaying Guestbook Entries

```
<p id="stepTitle">Developing Your First Portlet</p><p>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 class 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 *Generate*.

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 uses two portlet phases: render and action. To make the guestbook show the saved guestbook entries when users view the application, you must customize your portlet's render functionality, which it's currently inheriting from its parent class, `MVCPortlet`.

1. 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;
}
```

   As you can see, this method splits the entries in the `String` array into two parts based on the caret (^) character.

2. 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, calls `parsEntries` to convert it to a `List` of `Entry` objects, and places that `List` into the request object. It then calls the parent class's `render` method.

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.

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 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.

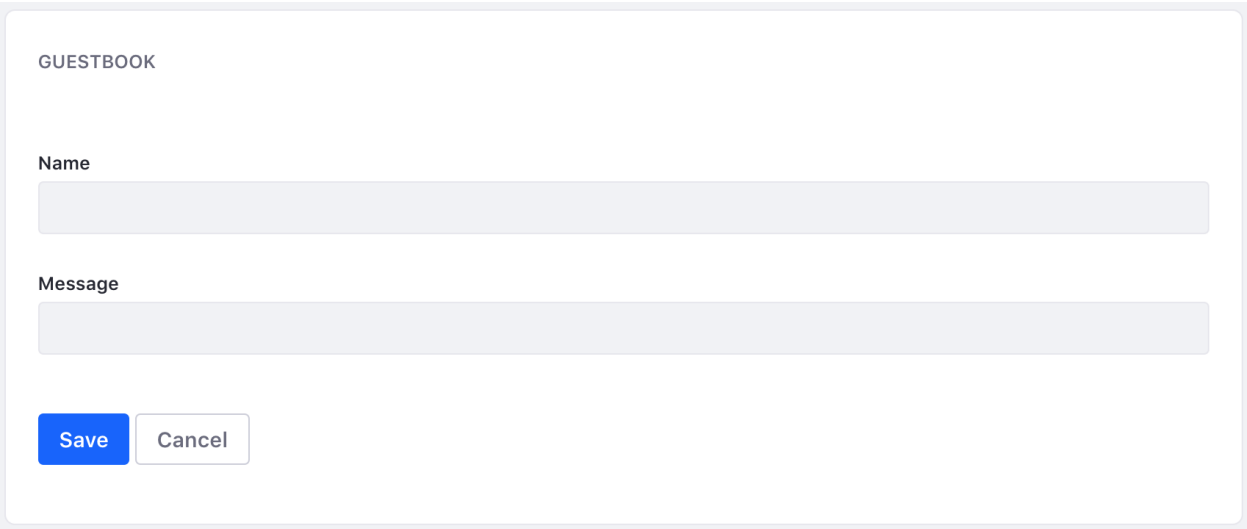


Figure 17.8: You have a form to enter information.

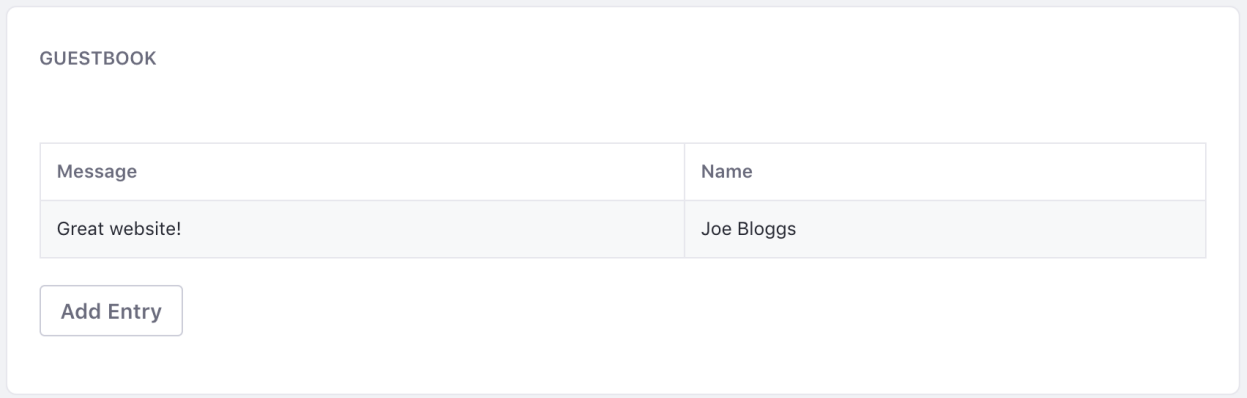


Figure 17.9: Submitted entries are displayed here..

CHAPTER 18

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.

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!

18.1 What is Service Builder?

```
<p id="stepTitle">Generating the Back-end</p><p>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.

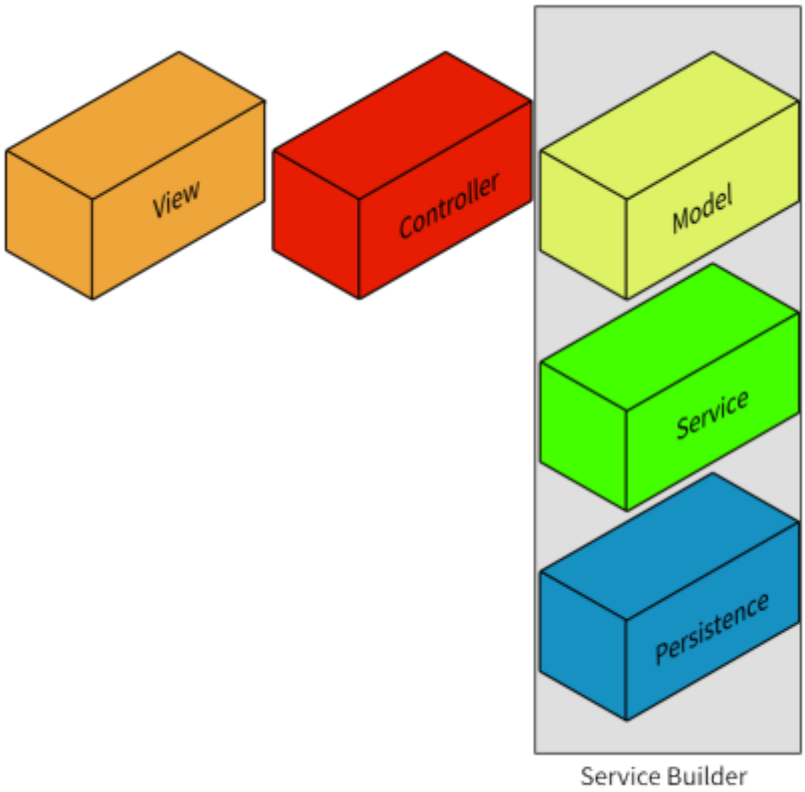


Figure 18.1: Service Builder generates the shaded layers of your application.

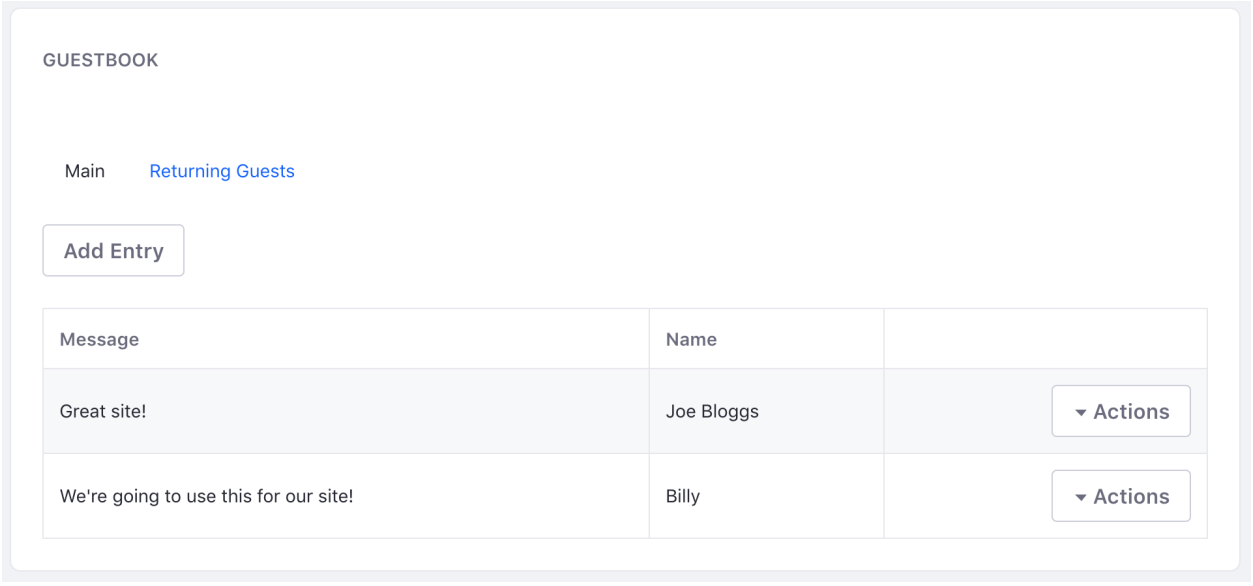


Figure 18.2: When you're done, the Guestbook supports multiple guestbooks and makes use of many Liferay features.

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 from the persistence layer. 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 which calls the persistence layer to store 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 Dev Studio DXP, 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.

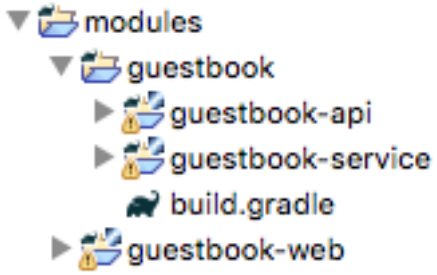


Figure 18.3: Your current project structure.

18.2 Generating Model, Service, and Persistence Layers

```
<p id="stepTitle">Generating the Back-end</p><p>Step 2 of 3</p>
```

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 allows for 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`. Make sure the *Source* tab is selected.

2. When Liferay Dev Studio DXP 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, replace the generated fields with this one:

```
 <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>
```

A finder 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 audit fields to match the fields in 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, an 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 Dev Studio DXP, 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 Dev Studio DXP.

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.

Figure 18.4: The Model, Service, and Persistence Layer comprise a loose coupling design.

18.3 Implementing Service Methods

```
<p id="stepTitle">Generating the Back-end</p><p>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:

```
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 classes 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 classes 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 Dev Studio DXP.

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 19

REFACTORING THE PROTOTYPE

Earlier, 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 must 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!

19.1 Organizing Folders for Larger Applications

```
<p id="stepTitle">Refactoring the Prototype</p><p>Step 1 of 6</p>
```

In larger projects, it is important to have all of your files and modules well organized. You'll make two changes to help better organize your project:

1. Move the `guestbook-web` module into the `guestbook` folder so that it's in the same place as the `guestbook-service` and `guestbook-api` modules that you created.
2. Since you'll now have two portlets, reorganize your JSPs to group them by portlet.

Moving guestbook-web

The best way to move modules around is to use Dev Studio DXP's *Refactor* function. The refactor function scans for and updates project dependencies and links.

1. In the *Project Explorer*, right-click on `guestbook-web` and select *Refactor* → *Move*.
2. In the window that appears, click *Browse*, navigate to the `guestbook` folder and then click *New Folder*.
3. Name the new folder `guestbook-web`.
4. Click *Open* and then *OK* to confirm.

Your `guestbook-web` folder now appears in the structure with the other modules.

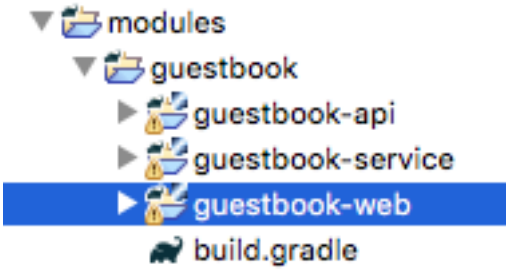


Figure 19.1: After you move it using the Refactor function, all of your modules are in the same folder..

Reorganizing JSPs

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.

```
<p id="stepTitle">Refactoring the Prototype</p><p>Step 2 of 6</p>
```

When users add applications to a page, they pick them from a list of *display categories*.

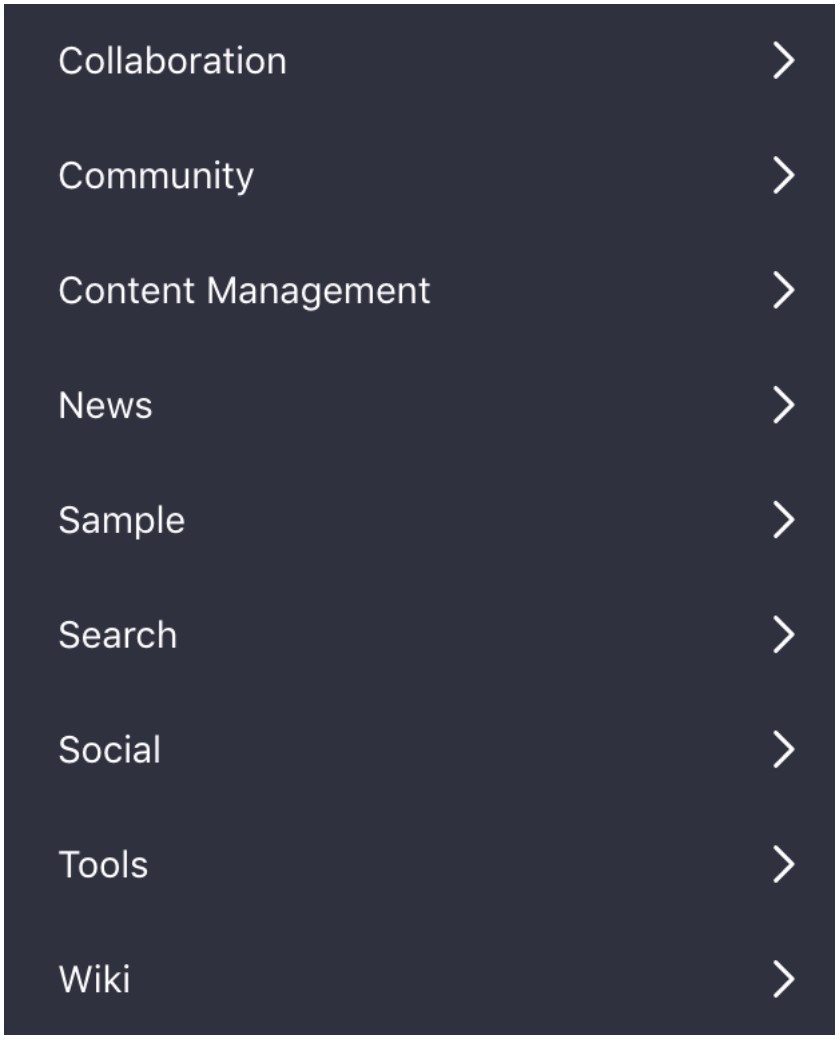


Figure 19.2: Users choose applications from a list of display categories.

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.

1. 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.

19.3 Creating Portlet Keys

```
<p id="stepTitle">Refactoring the Prototype</p><p>Step 3 of 6</p>
```

`PortletKeys` 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 (but add a trailing comma to the prior property!).

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.

19.4 Integrating the New Back-end

```
<p id="stepTitle">Refactoring the Prototype</p><p>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);

            response.setRenderParameter(
                "guestbookId", Long.toString(guestbookId));

        }
        catch (Exception e) {
            System.out.println(e);

            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.

19.5 Updating the View

```
<p id="stepTitle">Refactoring the Prototype</p><p>Step 5 of 6</p>
```

You updated more than just the mechanism behind creating entries: you completely changed the 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.petra.string.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.

19.6 Fitting it All Together

```
<p id="stepTitle">Refactoring the Prototype</p><p>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 features.

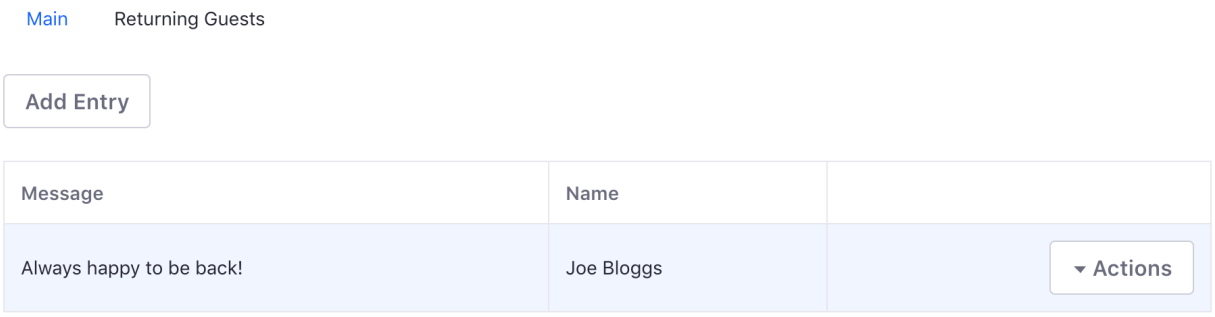


Figure 19.3: Your first guestbook and entry appears. Nice job!

CHAPTER 20

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:

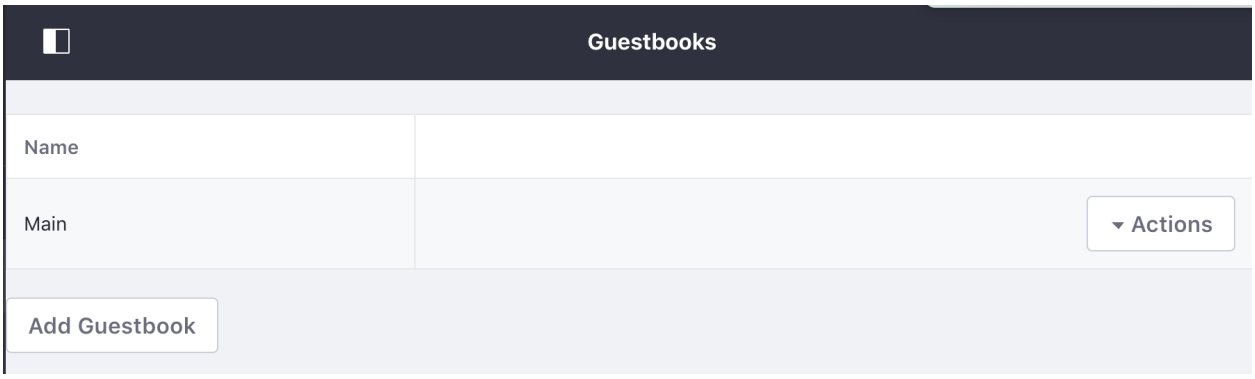


Figure 20.1: The Guestbook Admin portlet lets administrators manage Guestbooks.

Are you ready to begin?

Let's Go!

20.1 Creating the Classes

```
<p id="stepTitle">Writing the Guestbook Admin App</p><p>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.

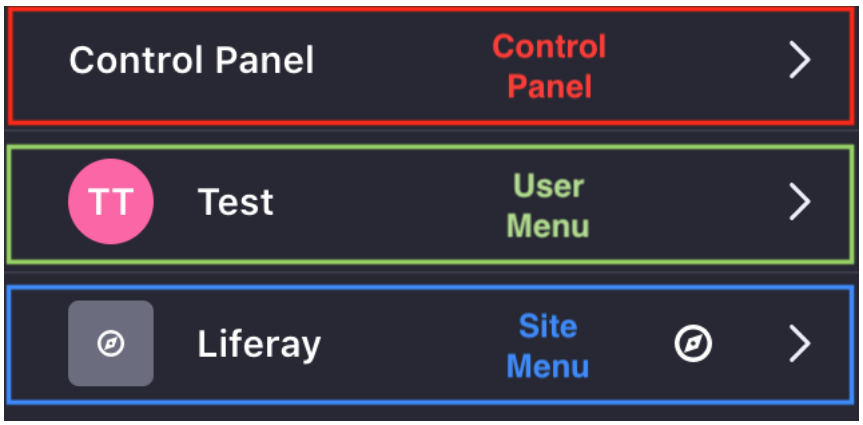


Figure 20.2: The product menu is split into three sections: the Control Panel, the User menu, and the Sites menu.

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`.
5. 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.

20.2 Adding Metadata

```
<p id="stepTitle">Writing the Guestbook Admin App</p><p>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 life cycles.

Follow these steps:

1. Add the following portlet key to the `GuestbookPortletKeys` class:

```
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.

20.3 Updating Your Service Layer

```
<p id="stepTitle">Writing the Guestbook Admin App</p><p>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, remove a method, 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.service.impl` 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 Dev Studio DXP, 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.

20.4 Defining Portlet Actions

```
<p id="stepTitle">Writing the Guestbook Admin App</p><p>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.

20.5 Creating a User Interface

```
<p id="stepTitle">Writing the Guestbook Admin App</p><p>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 displays 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.

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.

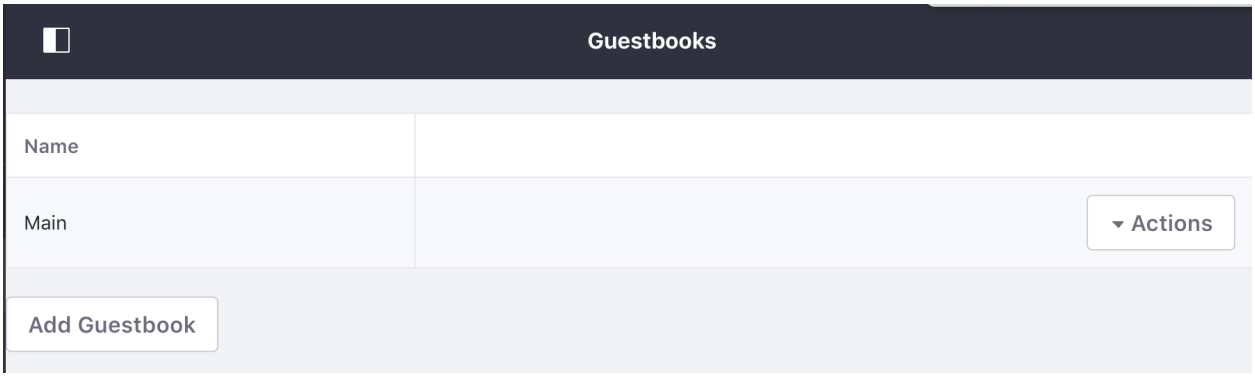


Figure 20.3: The Guestbook Admin portlet lets administrators add or edit guestbooks, configure their permissions, or delete them.

CHAPTER 21

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?

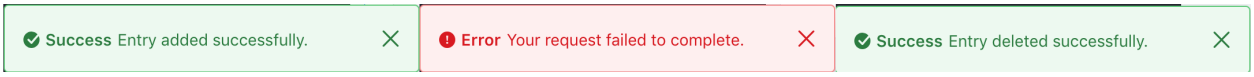


Figure 21.1: You can use Liferay's APIs to display helpful messages.

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!

21.1 Creating Language Keys

```
<p id="stepTitle">Displaying Messages and Errors</p><p>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_es.properties` for Spanish 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.

21.2 Adding Failure and Success Messages

```
<p id="stepTitle">Displaying Messages and Errors</p><p>Step 2 of 3</p>
```

To display 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 through a 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, and add the 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 from 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.

21.3 Adding Messages to JSPs

```
<p id="stepTitle">Displaying Messages and Errors</p><p>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="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" />
```

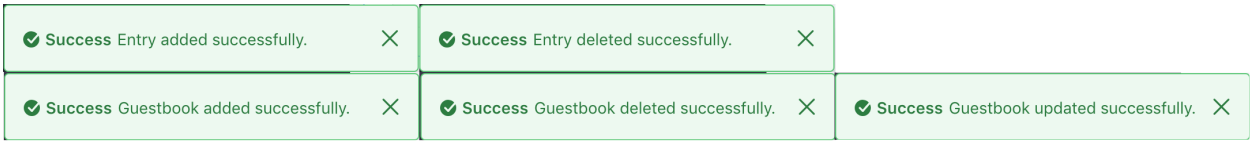


Figure 21.2: Now the message displays the value you specified in `Language.properties`.

Congratulations! You've added useful feedback for operations in your application.

Your application is shaping up, but it is missing another important feature: permissions. Next, you'll add permission checking for your guestbooks and entries.

CHAPTER 22

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. You'll follow Liferay's well-defined process for implementing permissions, called *DRAC*:

- **Define** all resources and permissions
- **Register** all defined resources in the permissions system
- **Associate** permissions with resources
- **Check** for permission before returning resources

Ready to start?
Let's Go!

22.1 Defining Permissions

```
<p id="stepTitle">Implementing Permissions</p><p>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 remote 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.

First, create the permissions file in the `guestbook-service` project:

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 7.1.0//EN" "http://www.liferay.com/dtd/liferay-resource-action-mapping_7_1_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>
```

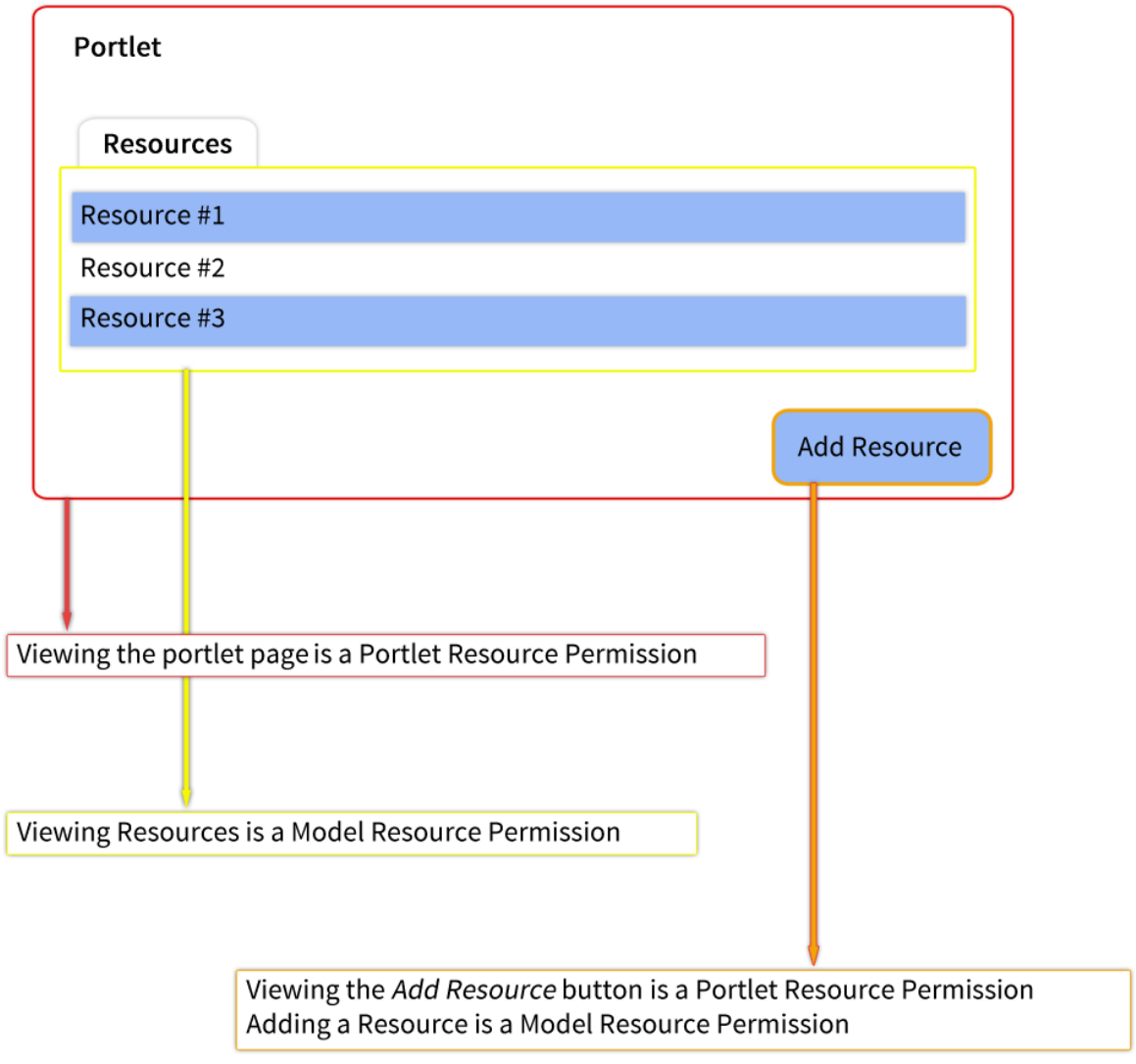


Figure 22.1: Portlet permissions and resource permissions cover different parts of the application.

```
            <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` and `ADD_ENTRY`. Site members can `ADD_ENTRY` by default, while guests can't perform either 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.

Next, you must 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.

You now have permissions defined 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. Create a subfolder called resource-actions in the src/main/resources/META-INF folder.

2. Create a new file in this folder called default.xml.

3. 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 7.1.0//EN" "http://www.liferay.com/dtd/liferay-resource-action-mapping_7_1_0.dtd">
```

4. 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.

5. 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 can access it.

6. 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.

7. Save the file.
8. In `guestbook-web`'s `src/main/resources` folder, create a file called `portlet.properties`.
9. In this file, place the following property:

```
resource.actions.configs=META-INF/resource-actions/default.xml
```

10. 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.

22.2 Registering Your Defined Permissions

```
<p id="stepTitle">Implementing Permissions</p><p>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.

To use these resources, Liferay DXP must know about them. To do that you *register* the resources with the system, both in the database and with the running permissions system in the OSGi container.

Registering Permissions in the Database

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, you need only 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.service.impl` 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(gues` 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 to the database. 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. Now you must register permissions with the permissions system, so it knows how to check for them.

Registering Your Entities with the Permissions Service

A running service checks permissions, but since the Guestbook portlet, Guestbooks, and Guestbook Entries are new to the system, it must be taught about them. You do this by creating permissions registrar classes. These follow what you did in `default.xml`: you need one for your portlet permissions and one for each of your entities. First, you must do a little reorganization.

1. In your API module, create a `GuestbookConstants` class in a new package called `com.liferay.docs.guestbook.cons`

```
package com.liferay.docs.guestbook.constants;

public class GuestbookConstants {

    public static final String RESOURCE_NAME = "com.liferay.docs.guestbook";

}
```

   The `RESOURCE_NAME` string must match exactly your resource name from `default.xml`. You'll see why in a moment.

2. You have a `GuestbookPortletKeys` class in your web module. These keys must now be accessible to all modules, so drag this class from the web module and drop it into the new `com.liferay.docs.guestbook.constants` package in your API module.

Now you're ready to create your permissions registrar classes.

3. In your service bundle, create a package that by convention ends in `internal.security.permission.resource`.
4. Create a class in this package called `GuestbookModelResourcePermissionRegistrar` with the contents below.

```
package com.liferay.docs.guestbook.internal.security.permission.resource;

import java.util.Dictionary;

import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;
import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Deactivate;
import org.osgi.service.component.annotations.Reference;

import com.liferay.docs.guestbook.constants.GuestbookConstants;
import com.liferay.docs.guestbook.constants.GuestbookPortletKeys;
import com.liferay.docs.guestbook.model.Guestbook;
import com.liferay.docs.guestbook.service.GuestbookLocalService;
import com.liferay.exportimport.kernel.staging.permission.StagingPermission;
import com.liferay.portal.kernel.security.permission.resource.ModelResourcePermission;
import com.liferay.portal.kernel.security.permission.resource.ModelResourcePermissionFactory;
import com.liferay.portal.kernel.security.permission.resource.PortletResourcePermission;
import com.liferay.portal.kernel.security.permission.resource.StagedModelPermissionLogic;
import com.liferay.portal.kernel.security.permission.resource.WorkflowedModelPermissionLogic;
import com.liferay.portal.kernel.service.GroupLocalService;
import com.liferay.portal.kernel.util.HashMapDictionary;
import com.liferay.portal.kernel.workflow.permission.WorkflowPermission;

@Component (immediate=true)
public class GuestbookModelResourcePermissionRegistrar {

 @Activate
    public void activate(BundleContext bundleContext) {
        Dictionary<String, Object> properties = new HashMapDictionary<>();

        properties.put("model.class.name", Guestbook.class.getName());

        _serviceRegistration = bundleContext.registerService(
            ModelResourcePermission.class,
            ModelResourcePermissionFactory.create(
                Guestbook.class, Guestbook::getGuestbookId,
                _guestbookLocalService::getGuestbook, _portletResourcePermission,
                (modelResourcePermission, consumer) -> {
                    consumer.accept(
                        new StagedModelPermissionLogic<>(
                            _stagingPermission, GuestbookPortletKeys.GUESTBOOK,
                            Guestbook::getGuestbookId));
                    consumer.accept(
                        new WorkflowedModelPermissionLogic<>(
                                _workflowPermission, modelResourcePermission,
                                _groupLocalService, Guestbook::getGuestbookId));
                }),
            properties);
    }

    @Deactivate
```

```
    public void deactivate() {
        _serviceRegistration.unregister();
    }

    @Reference
    private GuestbookLocalService _guestbookLocalService;

    @Reference(target = "(resource.name=" + GuestbookConstants.RESOURCE_NAME + ")")
    private PortletResourcePermission _portletResourcePermission;

    private ServiceRegistration<ModelResourcePermission> _serviceRegistration;

    @Reference
    private StagingPermission _stagingPermission;

    @Reference
    private WorkflowPermission _workflowPermission;

    @Reference
    private GroupLocalService _groupLocalService;
}
```

This class registers a chain of permission logic classes for checking permissions for Guestbook entities. Since this functionality is the same for all entities, all that's necessary is to specify yours in addition to the standard Liferay ones for staging and workflow. Introspection is done on your entity by the factory to create the necessary permissions service. You implemented the constants class so you can specify the resource model name you defined in `default.xml`. The `model.class.name` is set so that any module needing this service can find this model resource permission by its type.

Now create the registrar for the `Entry` entity:

1. Create a class in the same package called `GuestbookEntryModelResourcePermissionRegistrar`.
2. The only difference between this class and the one above is that it operates on `Entry` entities instead of `Guestbook` entities (the imports have been left off in the snippet below):

```
@Component(immediate = true)
public class GuestbookEntryModelResourcePermissionRegistrar {

 @Activate
    public void activate(BundleContext bundleContext) {
        Dictionary<String, Object> properties = new HashMapDictionary<>();

        properties.put("model.class.name", Entry.class.getName());

        _serviceRegistration = bundleContext.registerService(
            ModelResourcePermission.class,
            ModelResourcePermissionFactory.create(
                Entry.class, Entry::getEntryId,
                _entryLocalService::getEntry, _portletResourcePermission,
                (modelResourcePermission, consumer) -> {
                    consumer.accept(
                        new StagedModelPermissionLogic<>(
                            _stagingPermission, GuestbookPortletKeys.GUESTBOOK,
                            Entry::getEntryId));
                    consumer.accept(
                        new WorkflowedModelPermissionLogic<>(
                                _workflowPermission, modelResourcePermission,
                                _groupLocalService, Entry::getEntryId));
                }),
            properties);
    }
```

```
    @Deactivate
    public void deactivate() {
        _serviceRegistration.unregister();
    }

    @Reference
    private EntryLocalService _entryLocalService;

    @Reference(target = "(resource.name=" + GuestbookConstants.RESOURCE_NAME + ")")
    private PortletResourcePermission _portletResourcePermission;

    private ServiceRegistration<ModelResourcePermission> _serviceRegistration;

    @Reference
    private StagingPermission _stagingPermission;

    @Reference
    private WorkflowPermission _workflowPermission;

    @Reference
    private GroupLocalService _groupLocalService;
}
```

Finally, create the registrar for the portlet permissions:

1. Create a class in the same package called `GuestbookPortletResourcePermissionRegistrar`.
2. This class is simpler because you don't have to tell it how to retrieve primary keys from any entity:

```
@Component (immediate = true)
public class GuestbookPortletResourcePermissionRegistrar {

        @Activate
    public void activate(BundleContext bundleContext) {
        Dictionary<String, Object> properties = new HashMapDictionary<>();

        properties.put("resource.name", GuestbookConstants.RESOURCE_NAME);

        _serviceRegistration = bundleContext.registerService(
            PortletResourcePermission.class,
            PortletResourcePermissionFactory.create(
                GuestbookConstants.RESOURCE_NAME,
                new StagedPortletPermissionLogic(
                    _stagingPermission, GuestbookPortletKeys.Guestbook)),
            properties);
    }

    @Deactivate
    public void deactivate() {
        _serviceRegistration.unregister();
    }

    private ServiceRegistration<PortletResourcePermission> _serviceRegistration;

    @Reference
    private StagingPermission _stagingPermission;

}
```

You've now completed step two: the R in DRAC: registering permissions. Next, you'll enable users to associate permissions with resources.

22.3 Assigning Permissions to Resources

```
<p id="stepTitle">Implementing Permissions</p><p>Step 3 of 4</p>
```

You've now defined your permissions and registered them in the container and in the database so permissions can be checked. Now you'll create a UI for users to assign permissions along with helper classes to make it easy to check permissions in the next step.

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 if that user has the `ADD_ENTRY` permission for that guestbook. Helper classes make it easier to check permissions:

1. Right-click the `guestbook-web` module and select *New → Package*. To follow Liferay's practice, name the package `com.liferay.docs.guestbook.web.security.permission.resource`. This is where you'll place your helper classes.
2. Right-click the new package and select *New → Class*. Name the class `GuestbookPermission`.
3. Replace this class's contents with the following code:

```
package com.liferay.docs.guestbook.web.security.permission.resource;

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

import com.liferay.docs.guestbook.constants.GuestbookConstants;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import com.liferay.portal.kernel.security.permission.resource.PortletResourcePermission;

@Component(immediate=true)
public class GuestbookPermission {

    public static boolean contains(PermissionChecker permissionChecker, long groupId, String actionId) {

        return _portletResourcePermission.contains(permissionChecker, groupId, actionId);

    }

    @Reference(
            target="(resource.name=" + GuestbookConstants.RESOURCE_NAME + ")",
            unbind="-"
            )
    protected void setPortletResourcePermission(PortletResourcePermission portletResourcePermission) {

        _portletResourcePermission = portletResourcePermission;
    }

    private static PortletResourcePermission _portletResourcePermission;

}
```

This class is a component defining one static method (so you don't have to instantiate the class) that encapsulates the model you're checking permissions for. Liferay's `PermissionChecker` class does most of the work: give it the proper resource and action, such as `ADD_ENTRY`, and it returns whether the permission exists or not.

There's only one method: a `check` method that throws an exception if the user doesn't have permission.

Next, you'll create helpers for your two entities:

1. Create a class in the same package called `GuestbookModelPermission.java`.
2. Replace this class's contents with the following code:

```
package com.liferay.docs.guestbook.web.security.permission.resource;

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

import com.liferay.docs.guestbook.model.Guestbook;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import com.liferay.portal.kernel.security.permission.resource.ModelResourcePermission;

@Component(immediate = true)
public class GuestbookModelPermission {

    public static boolean contains(
            PermissionChecker permissionChecker, Guestbook guestbook, String actionId) throws PortalException {

        return _guestbookModelResourcePermission.contains(permissionChecker, guestbook, actionId);
    }

    public static boolean contains(
            PermissionChecker permissionChecker, long guestbookId, String actionId) throws PortalException {

        return _guestbookModelResourcePermission.contains(permissionChecker, guestbookId, actionId);
    }

    @Reference(
            target = "(model.class.name=com.liferay.docs.guestbook.model.Guestbook)",
            unbind = "-")
    protected void setEntryModelPermission(ModelResourcePermission<Guestbook> modelResourcePermission) {

        _guestbookModelResourcePermission = modelResourcePermission;
    }

    private static ModelResourcePermission<Guestbook>_guestbookModelResourcePermission;

}
```

As you can see, this class is similar to `GuestbookPermission`. The difference is that `GuestbookModelPermission` is for the model/resource permission, so you supply the entity or its primary key (`guestbookId`).

Your final class is almost identical to `GuestbookModelPermission`, but it's for the `Entry` entity. Follow these steps to create it:

1. Create a class in the same package called `GuestbookEntryPermission.java`.
2. Replace this class's contents with the following code:

```
package com.liferay.docs.guestbook.web.security.permission.resource;

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

import com.liferay.docs.guestbook.model.Entry;
import com.liferay.portal.kernel.exception.PortalException;
import com.liferay.portal.kernel.security.permission.PermissionChecker;
import com.liferay.portal.kernel.security.permission.resource.ModelResourcePermission;

@Component(immediate = true)
```

```
public class GuestbookEntryPermission {

    public static boolean contains(
            PermissionChecker permissionChecker, Entry entry, String actionId) throws PortalException {

        return _guestbookEntryModelResourcePermission.contains(permissionChecker, entry, actionId);
    }

    public static boolean contains(
            PermissionChecker permissionChecker, long entryId, String actionId) throws PortalException {

        return _guestbookEntryModelResourcePermission.contains(permissionChecker, entryId, actionId);
    }

    @Reference(
            target = "(model.class.name=com.liferay.docs.guestbook.model.Entry)",
            unbind = "-")
    protected void setEntryModelPermission(ModelResourcePermission<Entry> modelResourcePermission) {

        _guestbookEntryModelResourcePermission = modelResourcePermission;
    }

    private static ModelResourcePermission<Entry>_guestbookEntryModelResourcePermission;

}
```

This class is almost identical to `GuestbookModelPermission`. The only difference is that `GuestbookEntryPermission` is for the `Entry` entity.

Now you can expose the permissions UI to your users so they can assign permissions:

1. Go to the `init.jsp` in your `guestbook-web` project. Add the following imports to the file:

```
<%@ page import="com.liferay.docs.guestbook.web.security.permission.resource.GuestbookModelPermission" %>
<%@ page import="com.liferay.docs.guestbook.web.security.permission.resource.GuestbookPermission" %>
<%@ page import="com.liferay.docs.guestbook.web.security.permission.resource.GuestbookEntryPermission" %>
<%@ 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.

2. Open `guestbook_actions.jsp`. Add this code just after the `<liferay-ui:icon-delete>` tag:

```
<c:if
test="<%=GuestbookModelPermission.contains(permissionChecker, guestbook.getGuestbookId(), ActionKeys.PERMISSIONS) %>">

    <liferay-security:permissionsURL
        modelResource="<%= Guestbook.class.getName() %>"
        modelResourceDescription="<%= guestbook.getName() %>"
        resourcePrimKey="<%= String.valueOf(guestbook.getGuestbookId()) %>"
        var="permissionsURL" />

    <liferay-ui:icon image="permissions" url="<%= permissionsURL %>" />

</c:if>
```

3. Save the file.

You just added an action button that displays Liferay's permissions UI for Guestbooks. On top of that, you used the permissions helper you just created to test whether users can even see the action button. It only appears if users have the *permissions* permission.

You'll implement this for Guestbook entries in the next step.

Congratulations! You've now created helper classes for your permissions, and you've enabled users to associate permissions with their resources. The only thing left is to implement permission checks in the application's view layer. You'll do this next.

22.4 Checking for Permission in JSPs

```
<p id="stepTitle">Implementing Permissions</p><p>Step 4 of 4</p>
```

You've already seen how user interface components can be wrapped in permission checks pretty easily. In this step, you'll implement the rest.

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:

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 = guestbooks.get(i);
                String cssClass = StringPool.BLANK;

                if (curGuestbook.getGuestbookId() == guestbookId) {
                    cssClass = "active";
                }

                if (GuestbookModelPermission.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, scopeGroupId, "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 determines 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>

        <c:if
            test="<%= GuestbookEntryPermission.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="<%=GuestbookEntryPermission.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="<%=GuestbookEntryPermission.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 action buttons 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! 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 23

SEARCH AND INDEXING

The Guestbook and Guestbook Admin portlets are up and running. 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.

Note: In previous versions of Liferay DXP, search was only *permissions aware* (indexed with the entity's permissions and searched with those permissions intact) if the application developer specified this line in the `Indexer` class's constructor:

```
setPermissionAware(true);
```

Now, search is permissions aware by default *if the new permissions approach*, as described in the previous step of this Learning Path and in these tutorials, is implemented for an application.

To enable search, you must index Guestbooks and their Entries. Although you probably won't have enough Guestbooks in a Site to warrant searching the Guestbook Admin portlet, indexing Guestbooks has other benefits. In a later section, you'll asset-enable Guestbooks and Guestbook Entries so the 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 index those Guestbooks. Ready?

Let's Go!

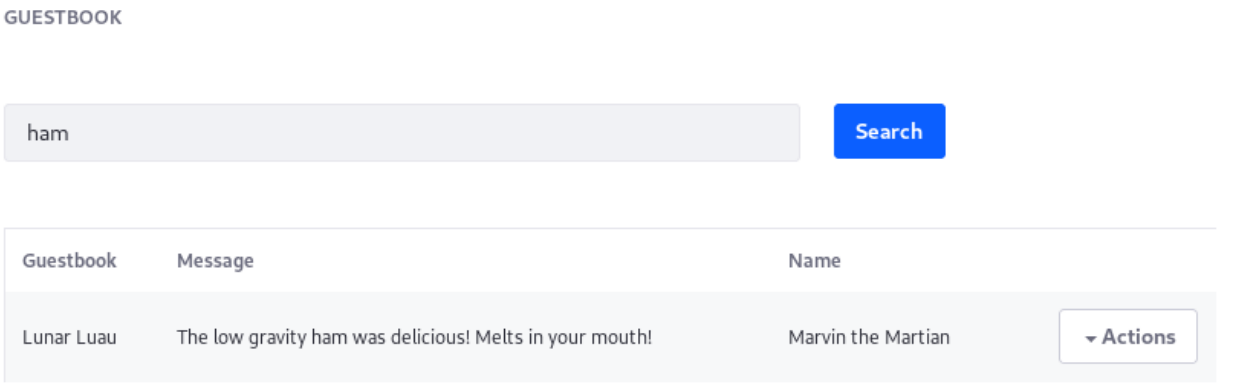


Figure 23.1: Add a search bar so users can search for Guestbook Entries. If a message or name matches the search query, the Entry is displayed in the search results.

CHAPTER 24

ENABLING SEARCH AND INDEXING FOR GUESTBOOKS

In this section, you'll create the classes that control these aspects of the search functionality:

- Registration:
  - `GuestbookSearchRegistrar` registers the search services to the search framework for the Guestbook entity.
- Indexing:
  - `GuestbookModelDocumentContributor` controls which Guestbook fields are indexed in the search engine.
  - `GuestbookModelIndexerWriterContributor` configures the re-indexing and batch re-indexing behavior for Guestbooks.
- Querying:
  - `GuestbookKeywordQueryContributor` contributes clauses to the ongoing search query.
  - `GuestbookModelPreFilterContributor` controls how search results are filtered before they're returned from the search engine.
- Generating Result Summaries:
  - `GuestbookModelSummaryContributor` constructs the result summary for Guestbooks, including specifying which fields to use.

After creating the search classes, you'll modify the service layer to update the search index when a guestbook is persisted. Specifically, `GuestbookLocalServiceImpl`'s `addGuestbook`, `updateGuestbook`, and `deleteGuestbook` methods are updated to invoke the guestbook indexer.

In prior versions of Liferay DXP, search and indexing was accomplished with one `*Indexer` class that extended `BaseIndexer`. In 7.0 is a new pattern that relies on composition instead of inheritance. If you want to use the old approach, feel free to extend `BaseIndexer`. It's still supported.

Since there's no reason to search for guestbooks in the UI, only back-end work is necessary.

Let's Go!

24.1 Understanding Search and Indexing

```
<p id="stepTitle">Enabling Search and Indexing for Guestbooks</p><p>Step 1 of 6</p>
```

By default, Liferay DXP uses Elasticsearch, a search engine backed by the popular Lucene search library, to implement its search and indexing functionality. You could search the database, but that requires resource-hogging table merges. Instead, a search engine like Elasticsearch converts searchable entities into *documents*. In Elasticsearch, documents are searchable database entities converted into JSON objects. After you implement indexing for guestbook entries, Liferay DXP creates a document for each entry. The indexing code specifies which guestbook entry fields to add to each guestbook entry document, and it adds all the guestbook entry documents to an index. A search returns a *hits* object containing pointers to 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 the following features to the existing Elasticsearch API:

- Indexed documents include the fields needed by Liferay DXP (e.g., `entryClassName`, `entryClassPK`, `assetTagNames`, `assetCategories`, `companyId`, `groupId`, staging status).
- It ensures the scope of returned search results is appropriate by applying the right filters to search requests.
- It provides permission checking and hit summaries to display in the search portlet.

To understand how the search and indexing code presented here makes your custom models seamlessly searchable, you must know how to influence each portion of the search and indexing cycle:

Indexing: Model entities store data fields in the database. For example, Guestbooks store the *name* field. During the cycle's Indexing step, you prepare the model entity to be searchable by defining the model's fields that are sent to the search engine, later used during a search.

To influence the way model entity fields are indexed in search engine documents,

`ModelDocumentContributor` classes specify which database fields are indexed in the model entity's search engine documents. This class's `contribute` method is called each time the `add` and `update` methods in the entity's service layer are called.

`ModelIndexerWriterContributor` classes configure the re-indexing and batch re-indexing behavior for the model entity. This class's method is called when a re-index is triggered from the Search administrative application found in Control Panel → Configuration → Search.

Searching: Most searches start with a user entering keywords into a search bar. The entered keywords are processed by the back-end search infrastructure, transformed into a *query* the search engine can understand, and used to match against each search document's fields.

To exert control over the way your model entity documents are searched,

`KeywordQueryContributor` classes contribute clauses to the ongoing search query. This is called at search time, and ensures that all the fields you indexed are also the ones searched. For example, if you index fields with the Site locale appended to them (`title_en_us`, for example), you want the search query to include the same locale when the document is searched. If the search query contain another locale (like `title_es_ES`) or searches the plain field (`title`), inaccurate results are returned.

`ModelPreFilterContributors` control how search results are filtered before they're returned from the search engine. For example, adding the workflow status to the query ensures that an entity in the trash isn't returned in the search results. For the Guestbook application, a `ModelPrefilterContributor` isn't necessary until you get to the section on workflow-enabling Guestbooks.

Returning Results: When a model entity's indexed search document is obtained during a search request, it's converted into a summary of the model entity.

To influence the result summaries for your model entity documents,

`ModelSummaryContributor` classes get the `Summary` object created for each search document, so you can manipulate it by adding specific fields or setting the length of the displayed content.

`ModelVisibilityContributor` classes control the visibility of model entities that can be attached to other asset types (for example, File Entries can be attached to Wiki Pages), in the search context. Since Guestbooks and Guestbook entries won't be attached to other assets, a model visibility contributor isn't necessary.

One important step must occur to make sure the above classes are discovered by the search framework.

Registration

To register the model entity with Liferay's search framework,

`SearchRegistrars` use the search framework's registry to define certain things about your model entity's `ModelSearchDefinition`: which fields are used by default to retrieve documents from the search engine, and which optional search services are registered for your entity. Registration occurs as soon as the Component is activated (during portal startup).

Let's index some Guestbooks, shall we?

24.2 Registering Guestbooks with the Search Framework

```
<p id="stepTitle">Enabling Search and Indexing for Guestbooks</p><p>Step 2 of 6</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 Search Service Provider Interface and API dependencies to the `build.gradle` file:

```
compileOnly group: "com.liferay", name: "com.liferay.portal.search.spi", version: "2.0.0"
compileOnly group: "com.liferay", name: "com.liferay.portal.search.api", version: "2.0.0"
```

3. Save the file and run `Refresh Gradle Project`.

Once the dependency is configured, register the Search services that build the entity's `ModelSearchDefinition`.

A *SearchRegistrar specifies the classes that the entity uses to contribute to building a ModelSearchDefinition. Activation of the SearchRegistrar component results in a cascade of activity in the search framework, culminating with the building of a DefaultIndexer. The DefaultIndexer is registered under the class name defined in the registrar, and then used for indexing/searching objects of that class.

Create the GuestbookSearchRegistrar:

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 GuestbookSearchRegistrar and populate it with two methods, activate and deactivate.

```
@Component(immediate = true)
public class GuestbookSearchRegistrar {

    @Activate
    protected void activate(BundleContext bundleContext) {

        _serviceRegistration = modelSearchRegistrarHelper.register(
            Guestbook.class, bundleContext, modelSearchDefinition -> {
                modelSearchDefinition.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);

                modelSearchDefinition.setModelIndexWriteContributor(
                    modelIndexWriterContributor);
                modelSearchDefinition.setModelSummaryContributor(
                    modelSummaryContributor);
            });
    }

    @Deactivate
    protected void deactivate() {

        _serviceRegistration.unregister();
    }
```

   The annotations @Activate and Deactivate ensure each method is invoked as soon as the Component is started (activated) or when it's about to be stopped (deactivated). On activation of the Component, a chain of search and indexing classes is registered for the Guestbook entity. Set the default selected field names used to retrieve results documents from the search engine. Then set the contributors used to build a model search definition.

2. Specify the service references for the class:

```
    @Reference(target = "(indexer.class.name=com.liferay.docs.guestbook.model.Guestbook)")
    protected ModelIndexerWriterContributor<Guestbook> modelIndexWriterContributor;

    @Reference
    protected ModelSearchRegistrarHelper modelSearchRegistrarHelper;

    @Reference(target = "(indexer.class.name=com.liferay.docs.guestbook.model.Guestbook)")
    protected ModelSummaryContributor modelSummaryContributor;

    private ServiceRegistration<?> _serviceRegistration;

}
```

Target the `Guestbook` model while looking up a reference to the contributor classes. Later, when you create these contributor classes, you'll specify the model name again to complete the circle.

3. Add the imports by Organizing Imports (Ctrl-Shift-O).

4. 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.search
```

The Guestbook search and indexing class registration is completed. Next write the search and indexing logic.

24.3 Indexing Guestbooks

```
<p id="stepTitle">Enabling Search and Indexing for Guestbooks</p><p>Step 3 of 6</p>
```

To control how Guestbook objects are translated into search engine documents, create two classes in the new search package:

1. Implement a `ModelDocumentContributor` that "contributes" fields to a search document indexed by the search engine. The main searchable field for guestbooks is the guestbook name.

2. `ModelIndexerWriterContributor` configures the batch indexing behavior for Guestbooks. This code is executed when Guestbooks are re-indexed from the Search administration section of the Control Panel.

Implementing `ModelDocumentContributor`

Create `GuestbookModelDocumentContributor` and populate it with this:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.Guestbook",
        service = ModelDocumentContributor.class
)
public class GuestbookModelDocumentContributor
    implements ModelDocumentContributor<Guestbook> {

    @Override
    public void contribute(Document document, Guestbook guestbook) {
        try {
            document.addDate(Field.MODIFIED_DATE, guestbook.getModifiedDate());

            Locale defaultLocale = PortalUtil.getSiteDefaultLocale(
    guestbook.getGroupId());

            String localizedTitle = LocalizationUtil.getLocalizedName(
    Field.TITLE, defaultLocale.toString());

            document.addText(localizedTitle, guestbook.getName());
        } catch (PortalException pe) {
            if (_log.isWarnEnabled()) {
                _log.warn(
    "Unable to index guestbook " + guestbook.getGuestbookId(), pe);
```

```
            }
        }
    }

    private static final Log _log = LogFactoryUtil.getLog(
    GuestbookModelDocumentContributor.class);

}
```

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.

Implementing `ModelIndexerWriterContributor`

Create `GuestbookModelIndexerWriterContributor` and populate it with this:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.Guestbook",
        service = ModelIndexerWriterContributor.class
)
public class GuestbookModelIndexerWriterContributor
    implements ModelIndexerWriterContributor<Guestbook> {

    @Override
    public void customize(
        BatchIndexingActionable batchIndexingActionable,
        ModelIndexerWriterDocumentHelper modelIndexerWriterDocumentHelper) {

        batchIndexingActionable.setPerformActionMethod((Guestbook guestbook) -> {
            Document document = modelIndexerWriterDocumentHelper.getDocument(
    guestbook);

            batchIndexingActionable.addDocuments(document);
        });
    }

    @Override
    public BatchIndexingActionable getBatchIndexingActionable() {
        return dynamicQueryBatchIndexingActionableFactory.getBatchIndexingActionable(
    guestbookLocalService.getIndexableActionableDynamicQuery());
    }

    @Override
    public long getCompanyId(Guestbook guestbook) {
        return guestbook.getCompanyId();
    }

    @Override
    public void modelIndexed(Guestbook guestbook) {
        entryBatchReindexer.reindex(
    guestbook.getGuestbookId(), guestbook.getCompanyId());
    }

    @Reference
    protected DynamicQueryBatchIndexingActionableFactory
    dynamicQueryBatchIndexingActionableFactory;

    @Reference
    protected EntryBatchReindexer entryBatchReindexer;
```

```
    @Reference
    protected GuestbookLocalService guestbookLocalService;

}
```

First look at the `customize` method. Configure the batch indexing behavior for the entity's documents by calling `BatchIndexingActionable` methods. This code uses the Guestbook's actionable dynamic query helper method to retrieve all Guestbooks in the virtual instance (identified by the Company ID). 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.

When you write the indexing classes for Entries, you'll add the Guestbook title to the Entry document. Thus, you must provide a way to update the indexed Entry documents if a Guestbook title is changed. The `modelIndexed` method calls a `reindex` method from an interface that will be created later for Entries.

Once the re-indexing behavior is in place, you can move on to controlling how Guestbook documents are queried from the search engine.

24.4 Querying for Guestbook Documents

```
<p id="stepTitle">Enabling Search and Indexing for Guestbooks</p><p>Step 4 of 6</p>
```

The code is in place for for indexing Guestbooks to the search engine. Next, you'll code the behavior necessary for querying the indexed documents.

Implement two interfaces:

1. `KeywordQueryContributor` contributes clauses to the ongoing search query.
2. `ModelPreFilterContributor` controls how search results are filtered before they're returned from the search engine.

Implementing `KeywordQueryContributor`

Create `GuestbookKeywordQueryContributor`:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.Guestbook",
        service = KeywordQueryContributor.class
)
public class GuestbookKeywordQueryContributor
    implements KeywordQueryContributor {

    @Override
    public void contribute(
        String keywords, BooleanQuery booleanQuery,
        KeywordQueryContributorHelper keywordQueryContributorHelper) {

        SearchContext searchContext =
    keywordQueryContributorHelper.getSearchContext();

        queryHelper.addSearchLocalizedTerm(
    booleanQuery, searchContext, Field.TITLE, false);
    }

    @Reference
    protected QueryHelper queryHelper;
```

```
}
```

This class adds Guestbook fields to the search query constructed by the Search application in Liferay DXP. Later, when you asset enable Guestbooks, this code will allow indexed Guestbooks to be searched from the Search application when a keyword is entered into the search bar. Use the query helper to add search terms to the query that allow Guestbooks to be found. Here it's important to note that adding the localized search term is important. Since the localized Guestbook title was indexed, you must retrieve the localized value from the search engine.

Once the query code is in place, define how returned Guestbook documents are summarized.

24.5 Generating Results Summaries

```
<p id="stepTitle">Enabling Search and Indexing for Guestbooks</p><p>Step 5 of 6</p>
```

The Search application and the Asset Publisher application must display results retrieved from the search engine. Control the summarized content by implementing a `ModelSummaryContributor`.

A summary is a condensed, text-based version of the entity's document that can be displayed generically. You create it by combining key parts of the entity's data so users can browse through search resmlts to find the entity they want.

Create a `GuestbookModelSummaryContributor`:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.Guestbook",
        service = ModelSummaryContributor.class
)
public class GuestbookModelSummaryContributor
    implements ModelSummaryContributor {

    @Override
    public Summary getSummary(
        Document document, Locale locale, String snippet) {

        Summary summary = createSummary(document);

        summary.setMaxContentLength(200);

        return summary;
    }

    private Summary createSummary(Document document) {
        String prefix = Field.SNIPPET + StringPool.UNDERLINE;

        String title = document.get(prefix + Field.TITLE, Field.TITLE);

        return new Summary(title, StringPool.BLANK);
    }

}
```

First override `getSummary` and set the maximum summary length on the summary returned. The value `200` is a Liferay standard. Control the summary creation in a utility method called `createSummary`. Create a `prefix` variable using two constants, `Field.SNIPPET` and `Stringpool.UNDERLINE`. The `snippet_title` field is returned from the `document.get` call, and added to the summary. Using the snippet field provides two benefits:

1. Snippet text can be highlighted so matching keywords are emphasized.
2. Snippet text can be shortened automatically by the Search application so a sensible portion of the field's text is displayed in the search results.

Guestbook titles are likely short, so only the highlighting behavior is useful for the title field of Guestbooks. For longer fields (like some `content` fields), the clipping behavior is more useful. Additional highlighting behavior can be configured via the `index.search.highlight.*` properties in portal.properties.

Create summaries by combining key parts of the entity's data so users can browse through search results to find the entity they want.

Once all the search and indexing logic is in place, update the service layer so `add`, `update`, and `delete` service calls trigger the new logic.

24.6 Handling Indexing in the Guestbook Service Layer

```
<p id="stepTitle">Enabling Search and Indexing for Guestbooks</p><p>Step 6 of 6</p>
```

Whenever a Guestbook database entity is added, updated, or deleted, the search index must be updated accordingly. The Liferay DXP annotation `@Indexable` combines with the `IndexableType` to mark your service methods so documents can be updated or deleted. Annotate `addGuestbook`, `updateGuestbook`, and `deleteGuestbook` service methods.

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 indexing classes control the type of index: 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 Dev Studio DXP, double-click `buildService` in `guestbook-service` → `build`. This re-runs Service Builder to incorporate your changes to `GuestbookLocalServiceImpl`.

Next, you'll enable search and indexing for Guestbook Entries.

CHAPTER 25

ENABLING SEARCH AND INDEXING FOR ENTRIES

In this section, you'll create the classes that control these aspects of the search functionality:

- Registration:
  - `EntrySearchRegistrar` registers the search service for the Entry entity.
- Indexing:
  - `EntryModelDocumentContributor` controls which Entry fields are indexed in the search engine.
  - `EntryModelIndexerWriterContributor` configures the re-indexing and batch re-indexing behavior for Entries.
  - `EntryBatchReindexer`, an interface, and its `EntryBatchReindexerImpl`, for re-indexing Entries when their Guestbook is updated.
- Querying:
  - `EntryKeywordQueryContributor` contributes clauses to the ongoing search query.
  - `EntryModelPreFilterContributor` controls how search results are filtered before they're returned from the search engine.
- Generating Result Summaries:
  - `EntryModelSummaryContributor` constructs the result summary for Entries, including specifying which fields to use.

After creating the search classes, modify the service layer to update the search index when an Entry is persisted:

- Update `EntryLocalServiceImpl`'s `addEntry`, `updateEntry`, and `deleteEntry` methods to update the index so it matches the databse.

Note: In prior versions of Liferay DXP, search and indexing was accomplished with one `*Indexer` class that extended `BaseIndexer`. This Learning Path demonstrates a new pattern that relies on composition instead of inheritance. If you desire to use the old approach, feel free to extend `BaseIndexer`. It's still supported.

Let's Go!

25.1 Registering Entries with the Search Framework

```
<p id="stepTitle">Enabling Search and Indexing for Entries</p><p>Step 1 of 5</p>
```

The search registrar for Entries is very similar to the one created for Guestbooks. You'll even put it in the same package (`com.liferay.docs.guestbook.search`).

Create the `EntrySearchRegistrar`:

1. In `com.liferay.docs.guestbook.search`, create a new class called `EntrySearchRegistrar` and populate it with two methods, `activate` and `deactivate`.

```
@Component(immediate = true)
public class EntrySearchRegistrar {

    @Activate
    protected void activate(BundleContext bundleContext) {

        _serviceRegistration = modelSearchRegistrarHelper.register(
            Entry.class, bundleContext, modelSearchDefinition -> {
                modelSearchDefinition.setDefaultSelectedFieldNames(
                    Field.COMPANY_ID, Field.ENTRY_CLASS_NAME,
                    Field.ENTRY_CLASS_PK, Field.UID,
                    Field.SCOPE_GROUP_ID, Field.GROUP_ID);

                modelSearchDefinition.setDefaultSelectedLocalizedFieldNames(
                    Field.TITLE, Field.CONTENT);

                modelSearchDefinition.setModelIndexWriteContributor(
                    modelIndexWriterContributor);
                modelSearchDefinition.setModelSummaryContributor(
                    modelSummaryContributor);
                modelSearchDefinition.setSelectAllLocales(true);

            });
    }

    @Deactivate
    protected void deactivate() {
        _serviceRegistration.unregister();
    }
```

As you did with Guestbooks, set the default selected field names used to retrieve results documents from the search engine. For Entries, call `setDefaultSelectedLocalizedFieldNames` for the title and content fields. This ensures that the localized version of the field is searched and returned. The only other difference with Entries is the call to `setSelectAllLocales(true)`. It takes the fields

set in `setDefaultSelectedLocalizedFieldNames` and sets those fields for each available locale in the `stored_fields` parameter of the search request. If not set to `true`, only a single locale is searched.

2. Specify the service references for the class:

```
    @Reference(target = "(indexer.class.name=com.liferay.docs.guestbook.model.Entry)")
    protected ModelIndexerWriterContributor<Entry> modelIndexWriterContributor;

    @Reference
    protected ModelSearchRegistrarHelper modelSearchRegistrarHelper;

    @Reference(target = "(indexer.class.name=com.liferay.docs.guestbook.model.Entry)")
    protected ModelSummaryContributor modelSummaryContributor;

    private ServiceRegistration<?> _serviceRegistration;

}
```

   Target the `Entry` model while looking up a reference to the contributor classes. Later, when you create these contributor classes, you'll specify the model name again to complete the circle.

The Entry search and indexing class registration is completed. Next write the search and indexing logic.

25.2 Indexing Entries

```
<p id="stepTitle">Enabling Search and Indexing for Entries</p><p>Step 2 of 5</p>
```

To control how Entry objects are translated into search engine documents, create these classes in the search package:

1. `ModelDocumentContributor`: The main searchable fields for Entries are *Name* and *Message*. The Guestbook name associated with the entry is indexed, too.
2. `ModelIndexerWriterContributor` configures the batch indexing behavior for Entries. This code is executed when Entries are re-indexed from the Search administration section of the Control Panel.
3. A new interface, `EntryBatchReindexer`, with its implementation, `EntryBatchReindexerImpl`. These classes contain code to ensure that Entries are re-indexed when their Guestbook is updated.

Implementing `ModelDocumentContributor`

Create `EntryModelDocumentContributor` and populate it with this:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.Entry",
        service = ModelDocumentContributor.class
)
public class EntryModelDocumentContributor
    implements ModelDocumentContributor<Entry> {
```

```
    @Override
    public void contribute(Document document, Entry entry) {
        try {
            Locale defaultLocale = PortalUtil.getSiteDefaultLocale(
    entry.getGroupId());

            document.addDate(Field.MODIFIED_DATE, entry.getModifiedDate());
            document.addText("entryEmail", entry.getEmail());

            String localizedTitle = LocalizationUtil.getLocalizedName(
    Field.TITLE, defaultLocale.toString());
            String localizedContent = LocalizationUtil.getLocalizedName(
    Field.CONTENT, defaultLocale.toString());

            document.addText(localizedTitle, entry.getName());
            document.addText(localizedContent, entry.getMessage());

            long guestbookId = entry.getGuestbookId();

            Guestbook guestbook = _guestbookLocalService.getGuestbook(
    guestbookId);

            String guestbookName = guestbook.getName();

            String localizedGbName = LocalizationUtil.getLocalizedName(
    Field.NAME, defaultLocale.toString());

            document.addText(localizedGbName, guestbookName);
        } catch (PortalException pe) {
            if (_log.isWarnEnabled()) {
                _log.warn("Unable to index entry " + entry.getEntryId(), pe);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static final Log _log = LogFactoryUtil.getLog(
    EntryModelDocumentContributor.class);

    @Reference
    private GuestbookLocalService _guestbookLocalService;

}
```

As with Guestbooks, add the localized values for fields that might be translated. The 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.

Implementing `ModelIndexerWriterContributor`

Create `EntryModelIndexerWriterContributor` and populate it with this:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.Entry",
        service = ModelIndexerWriterContributor.class
)
public class EntryModelIndexerWriterContributor
    implements ModelIndexerWriterContributor<Entry> {

    @Override
    public void customize(
        BatchIndexingActionable batchIndexingActionable,
```

```
        ModelIndexerWriterDocumentHelper modelIndexerWriterDocumentHelper) {

        batchIndexingActionable.setPerformActionMethod((Entry entry) -> {
            Document document = modelIndexerWriterDocumentHelper.getDocument(
    entry);

            batchIndexingActionable.addDocuments(document);

        });
    }

    @Override
    public BatchIndexingActionable getBatchIndexingActionable() {
        return dynamicQueryBatchIndexingActionableFactory.getBatchIndexingActionable(
    entryLocalService.getIndexableActionableDynamicQuery());
    }

    @Override
    public long getCompanyId(Entry entry) {
        return entry.getCompanyId();
    }

    @Reference
    protected DynamicQueryBatchIndexingActionableFactory
    dynamicQueryBatchIndexingActionableFactory;

    @Reference
    protected EntryLocalService entryLocalService;

}
```

The interesting work is done in the `customize` method, where all Entries are retrieved and added to a collection.

Implementing `EntryBatchReindexer`

Create a new interface class, `EntryBatchReindexer`, with one method called `reindex`:

```
package com.liferay.docs.guestbook.search;

public interface EntryBatchReindexer {

    public void reindex(long guestbookId, long companyId);

}
```

Then create the implementation class, `EntryBatchReindexerImpl`:

```
@Component(immediate = true, service = EntryBatchReindexer.class)
public class EntryBatchReindexerImpl implements EntryBatchReindexer {

    @Override
    public void reindex(long guestbookId, long companyId) {
        BatchIndexingActionable batchIndexingActionable =
    indexerWriter.getBatchIndexingActionable();

        batchIndexingActionable.setAddCriteriaMethod(dynamicQuery -> {
            Property guestbookIdPropery = PropertyFactoryUtil.forName(
    "guestbookId");

            dynamicQuery.add(guestbookIdPropery.eq(guestbookId));
        });

        batchIndexingActionable.setCompanyId(companyId);
```

```
        batchIndexingActionable.setPerformActionMethod((Entry entry) -> {
            Document document = indexerDocumentBuilder.getDocument(entry);

            batchIndexingActionable.addDocuments(document);
        });

        batchIndexingActionable.performActions();

    }

    @Reference(target = "(indexer.class.name=com.liferay.docs.guestbook.model.Entry)")
    protected IndexerDocumentBuilder indexerDocumentBuilder;

    @Reference(target = "(indexer.class.name=com.liferay.docs.guestbook.model.Entry)")
    protected IndexerWriter<Entry> indexerWriter;

}

The `reindex` method of the interface is called when a Guestbook is updated.
The entry documents are re-indexed to include the update Guestbook title.
```

Once the re-indexing behavior is in place, move on to the code for controlling how Entry documents are queried from the search engine.

25.3 Querying for Entry Documents

```
<p id="stepTitle">Enabling Search and Indexing for Entries</p><p>Step 3 of 5</p>
```

The code is in place for indexing Entries to the search engine. Next code the behavior necessary for querying the indexed documents.

Implement two classes:

1. `EntryKeywordQueryContributor` contributes clauses to the ongoing search query.
2. `EntryModelPreFilterContributor` controls how search results are filtered before they're returned from the search engine.

Implementing `KeywordQueryContributor`

Create `EntryKeywordQueryContributor` and populate it with this:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.Entry",
        service = KeywordQueryContributor.class
)
public class EntryKeywordQueryContributor implements KeywordQueryContributor {

    @Override
    public void contribute(
        String keywords, BooleanQuery booleanQuery,
        KeywordQueryContributorHelper keywordQueryContributorHelper) {

        SearchContext searchContext =
    keywordQueryContributorHelper.getSearchContext();

        queryHelper.addSearchLocalizedTerm(
    booleanQuery, searchContext, Field.TITLE, false);
        queryHelper.addSearchLocalizedTerm(
```

```
    booleanQuery, searchContext, Field.CONTENT, false);
        queryHelper.addSearchLocalizedTerm(
    booleanQuery, searchContext, "entryEmail", false);
    }

    @Reference
    protected QueryHelper queryHelper;

}
```

Adding the localized search terms is important. For all localized Entry fields in the index, retrieve the localized value from the search engine.

Now that the query code is in place, you can define how returned Entry documents are summarized.

25.4 Generating Results Summaries

```
<p id="stepTitle">Enabling Search and Indexing for Entries</p><p>Step 3 of 5</p>
```

The Search application and the Asset Publisher application display results retrieved from the search engine. You can control the display by implementing a `ModelSummaryContributor`.

Create a `EntryModelSummaryContributor`:

```
@Component(
        immediate = true,
        property = "indexer.class.name=com.liferay.docs.guestbook.model.Entry",
        service = ModelSummaryContributor.class
)
public class EntryModelSummaryContributor implements ModelSummaryContributor {

    @Override
    public Summary getSummary(
        Document document, Locale locale, String snippet) {

        Summary summary = createSummary(document);

        summary.setMaxContentLength(128);

        return summary;
    }

    private Summary createSummary(Document document) {
        String prefix = Field.SNIPPET + StringPool.UNDERLINE;

        String title = document.get(prefix + Field.TITLE, Field.CONTENT);
        String content = document.get(prefix + Field.CONTENT, Field.CONTENT);

        return new Summary(title, content);
    }

}
```

First override `getSummary`, and set the maximum summary length on the summary returned. The value `200` is a Liferay standard. Control the summary creation in a utility method called `createSummary`. Guestbooks only included the title in the summary, but Entries use the title and the content (the Entry message field) to populate the summary.

Create summaries by combining key parts of the entity's data.

Now that the search and indexing logic is in place, you can update the service layer so `add`, `update`, and `delete` service calls trigger the new logic.

25.5 Handling Indexing in the Entry Service Layer

```
<p id="stepTitle">Enabling Search and Indexing for Entries</p><p>Step 5 of 5</p>
```

Whenever an 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 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 indexing classes control exactly how the indexing happens. Setting the `@Indexable` annotation's type to `IndexableType.REINDEX` updates the indexed document 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 Dev Studio DXP, double-click `buildService` in `guestbook-service` → `build`. This re-runs Service Builder to incorporate your changes to `EntryLocalServiceImpl`.

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 26

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!

26.1 Adding a Search Bar to the Guestbook Portlet

```
<p id="stepTitle">Updating Your UI for Search</p><p>Step 1 of 2</p>
```

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:

```
<portlet:renderURL var="searchURL">
    <portlet:param name="mvcPath"
    value="/guestbookwebportlet/view_search.jsp" />
</portlet:renderURL>
```

   The render URL points to `/guestbookwebportlet/view_search.jsp` (created in the next step). You construct the URL first to specify what happens when the user submits a search query.

2. Right after the render URL, create an AUI form that adds an input field for search keywords and a *Submit* button that executes the form action, which is mapped to the `searchURL`.

```
<aui:form action="${searchURL}" name="fm">

    <div class="row">
        <div class="col-md-8">
            <aui:input inlineLabel="left" label="" name="keywords" placeholder="search-entries" size="256" />
```

```
        </div>

        <div class="col-md-4">
            <aui:button type="submit" value="search" />
        </div>
    </div>
</aui:form>
```

The body of the search form consists of a `<div>` with one row containing two fields: an input field, named `keywords` and a *Submit* button. 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. Next create the `view_search.jsp` file to display the search results.

26.2 Creating a Search Results JSP for the Guestbook Portlet

```
<p id="stepTitle">Updating Your UI for Search</p><p>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.

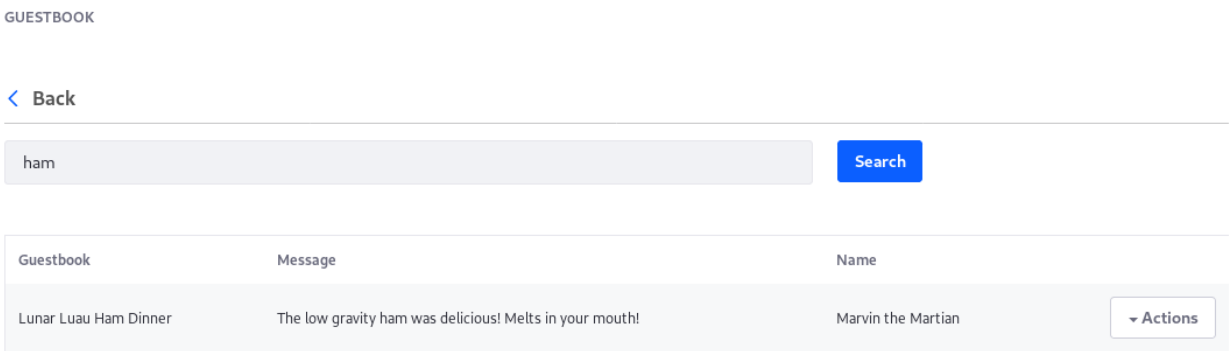


Figure 26.1: The search results should appear in a search container, and the Actions button should appear for each entry. The search bar should also be displayed.

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:

```
<portlet:renderURL var="searchURL">
        <portlet:param name="mvcPath"
        value="/guestbookwebportlet/view_search.jsp" />
</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}" name="fm">

    <liferay-ui:header backURL="${viewURL}" title="back" />

    <div class="row">
        <div class="col-md-8">
            <aui:input inlineLabel="left" label="" name="keywords" placeholder="search-entries" size="256" />
        </div>

        <div class="col-md-4">
            <aui:button type="submit" value="search" />
        </div>
    </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 *Back*. 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` creates 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.

Once indexing is in place, the asset framework can be added to the Guestbook application. It provides functionality that's shared 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.

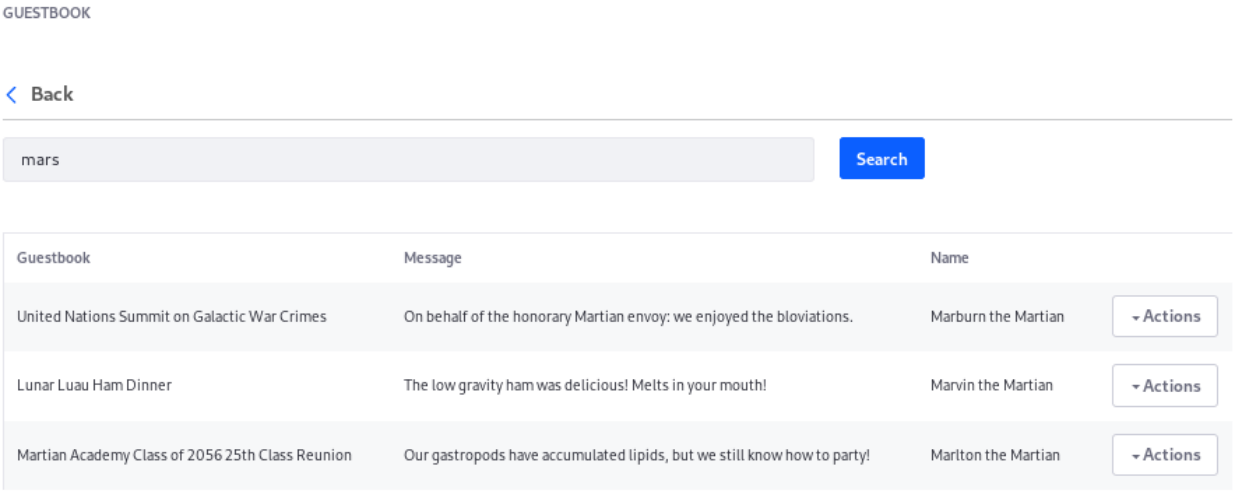


Figure 26.2: The Guestbook Application now supports searching for indexed Guestbook Entries.

CHAPTER 27

ASSETS: INTEGRATING WITH LIFERAY'S FRAMEWORK

The 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.

The asset framework includes the following features:

- Tags and categories
- Comments and ratings
- Related assets (a.k.a. Asset links)
- Faceted search
- Integration with the Asset Publisher portlet
- Integration with the Search portlet
- Integration with the Tags Navigation, Tag Cloud, and Categories Navigation portlets

Now 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 core portlets like the Asset Publisher, Tags Navigation, Tag Cloud, and Categories Navigation portlets. Ready to start?

Let's Go!

27.1 Enabling Assets at the Service Layer

```
<p>Enabling Assets at the Service Layer</p><p>Step 1 of 3</p>
```

Each row in the `AssetEntry` table represents an asset. It has an `entryId` primary key along with `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

(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 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 the AssetEntry service so your application can add asset entries corresponding to guestbooks and guestbook entries. You also use the AssetLink service 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, because it searches for entities by status (i.e., is it Workflow-approved?) and by groupId (i.e., is it in this Site?). Add these below the existing finders for the Guestbook and Entry entities:

   ```
   <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. Run the buildService Gradle task. This task injects the objects referenced above into your services for use.

4. Right-click build.gradle and select *Gradle → Refresh Gradle Project*.

Great! Next, you'll handle assets in your service layer.

27.2 Handling Assets at the Guestbook Service Layer

```
<p id="stepTitle">Enabling Assets at the Service Layer</p><p>Step 2 of 3</p>
```

Before you can update the Service Layer to add the Asset Renderers, you must update your build.gradle to provide the portlet-api and javax.servlet-api libraries that the asset link service needs to function.

1. Open the build.gradle file in your guestbook-service module.

2. Add the following two lines in the dependencies section:

   ```
   compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
   compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
   ```

Now you'll update the guestbook service layer to use assets. You must update the add, update, and delete methods of your project's GuestbookLocalServiceImpl:

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 `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.updateLi` 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.

27.3 Handling Assets for Entry Service Layer

```
<p id="stepTitle">Enabling Assets at the Service Layer</p><p>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, except you're specifying assets for `Entry` entities.

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/resources/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 the asset framework. Every asset needs an asset renderer class so the Asset Publisher portlet can display it.

CHAPTER 28

IMPLEMENTING ASSET RENDERERS

Assets are display 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. 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!

28.1 Implementing a Guestbook Asset Renderer

```
<p id="stepTitle">Implementing Asset Renderers</p><p>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.

You'll create 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.web.internal.asset` in the `guestbook-web` 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:

   ```
   public class GuestbookAssetRenderer extends BaseJSPAssetRenderer<Guestbook> {

   }
   ```

2. Add the constructor, the guestbook class variable, and the permissions model resource. Most of the methods in this class are getters that return fields from the private `_guestbook` object. Methods requiring a permission check use `_guestbookModelResourcePermission`:

```
public GuestbookAssetRenderer(Guestbook guestbook, ModelResourcePermission<Guestbook> modelResourcePermission) {

            _guestbook = guestbook;
            _guestbookModelResourcePermission = modelResourcePermission;
}

private Guestbook _guestbook;
private final ModelResourcePermission<Guestbook> _guestbookModelResourcePermission;
private Logger logger = Logger.getLogger(this.getClass().getName());
```

3. The `BaseJSPAssetRenderer` abstract class that you're extending contains dummy implementations of the `hasEditPermission` and `hasViewPermission` methods that you must override with actual permission checks using the permissions resources that you created earlier:

```
@Override
public boolean hasEditPermission(PermissionChecker permissionChecker)
{
    try {
        return _guestbookModelResourcePermission.contains(
            permissionChecker, _guestbook, ActionKeys.UPDATE);
    }
    catch (Exception e) {
    }

    return false;
}

@Override
public boolean hasViewPermission(PermissionChecker permissionChecker)
{
    try {
        return _guestbookModelResourcePermission.contains(
            permissionChecker, _guestbook, ActionKeys.VIEW);
    }
    catch (Exception e) {
    }

    return true;
}
```

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 and the `Guestbook` entity available in the `HttpServletRequest` object.

5. Override the `getJspPath` method. It returns a string representing 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) {

      logger.log(Level.SEVERE, e.getMessage());

  } catch (SystemException e) {

      logger.log(Level.SEVERE, e.getMessage());

  }

  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. Organize imports (Ctrl-Shift-O) and save the file. Choose `com.liferay.petra.*` libraries when prompted, to avoid the deprecated ones in Liferay's kernel. For logging, choose `java.util.logging.Logger` and `java.util.logging.Level`.

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.web.internal.asset package, create a class called GuestbookAssetRendererFactory that extends Liferay DXP's BaseAssetRendererFactory class, and overwrite the generated constructor and class variables with this:

```
@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);
  }

  private ServletContext _servletContext;
  private GuestbookLocalService _guestbookLocalService;
  private static final boolean _LINKABLE = true;
  public static final String CLASS_NAME = Guestbook.class.getName();
  public static final String TYPE = "guestbook";
  private Logger logger = Logger.getLogger(this.getClass().getName());
  private ModelResourcePermission<Guestbook>
  _guestbookModelResourcePermission;
```

   This code contains the class declaration, the constructor, and the class variables. It sets the class name it creates an AssetRenderer for, a portlet ID, and a true boolean (_LINKABLE). The boolean denotes implemented methods that provide URLs in the generated AssetRenderer.

2. Implement the getAssetRenderer method, which constructs new GuestbookAssetRenderer instances for particular 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 and permissions resource model as arguments:

```
@Override
public AssetRenderer<Guestbook> getAssetRenderer(long classPK, int type)
throws PortalException {

  Guestbook guestbook = _guestbookLocalService.getGuestbook(classPK);

  GuestbookAssetRenderer guestbookAssetRenderer =
  new GuestbookAssetRenderer(guestbook, _guestbookModelResourcePermission);

  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);
  long groupId = guestbook.getGroupId();
  return GuestbookPermission.contains(permissionChecker, groupId,
  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) {

              logger.log(Level.SEVERE, e.getMessage());

        }

      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;
            }

        @Reference(unbind = "-")
            protected void setGuestbookLocalService(GuestbookLocalService guestbookLocalService) {
                _guestbookLocalService = guestbookLocalService;
        }
    }
```

6. Organize imports (Ctrl-Shift-O). Select the `org.osgi` packages (not `a.Qute`) when prompted and save the file.

Great! The guestbook asset renderer is complete. Next, you'll create the entry asset renderer.

28.2 Implementing an Entry Asset Renderer

```
<p id="stepTitle">Implementing Asset Renderers</p><p>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 to learn how this code works.

Creating the EntryAssetRenderer Class

In the `com.liferay.docs.guestbook.web.internal.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.web.internal.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.security.permission.resource.ModelResourcePermission;
import com.liferay.portal.kernel.util.HtmlUtil;
import com.liferay.portal.kernel.util.PortalUtil;
import com.liferay.petra.string.StringUtil;
import com.liferay.docs.guestbook.constants.GuestbookPortletKeys;
import com.liferay.docs.guestbook.model.Entry;
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, ModelResourcePermission<Entry> modelResourcePermission) {
```

```
    _entry = entry;
    _entryModelResourcePermission = modelResourcePermission;
}

@Override
public boolean hasViewPermission(PermissionChecker permissionChecker)
{
    try {
        return _entryModelResourcePermission.contains(
                permissionChecker, _entry, ActionKeys.VIEW);
    }
    catch (Exception e) {
    }

    return true;
}

@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 final ModelResourcePermission<Entry> _entryModelResourcePermission;
    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 other 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.web.inte` package, create a class called `EntryAssetRendererFactory` that extends Liferay DXP's `BaseAssetRendererFactory` class. Replace its content with the following code:

```
package com.liferay.docs.guestbook.web.internal.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.constants.GuestbookPortletKeys;
import com.liferay.docs.guestbook.model.Entry;
import com.liferay.docs.guestbook.service.EntryLocalService;
import com.liferay.docs.guestbook.web.internal.security.permission.resource.GuestbookEntryPermission;
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.security.permission.resource.ModelResourcePermission;
import com.liferay.portal.kernel.theme.ThemeDisplay;
import com.liferay.portal.kernel.util.WebKeys;

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, _entryModelResourcePermission);

    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 GuestbookEntryPermission.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";

    private ModelResourcePermission<Entry>
    _entryModelResourcePermission;


}
```

Exporting the Asset Package

The container makes the asset renderers and their factories available to Liferay DXP when it needs them. You must export the package to the container.

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.web.internal.security.permission.resource,\
                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. Then add and edit guestbooks and guestbook entries. Then check the Asset Publisher portlet. The Asset Publisher dynamically displays assets of any kind from the current Site.

Confirm that the Asset Publisher displays the guestbooks and guestbook entries that you added.

Great! Next, you'll update your portlets' user interfaces to use several asset framework features: comments, ratings, tags, categories, and related assets.

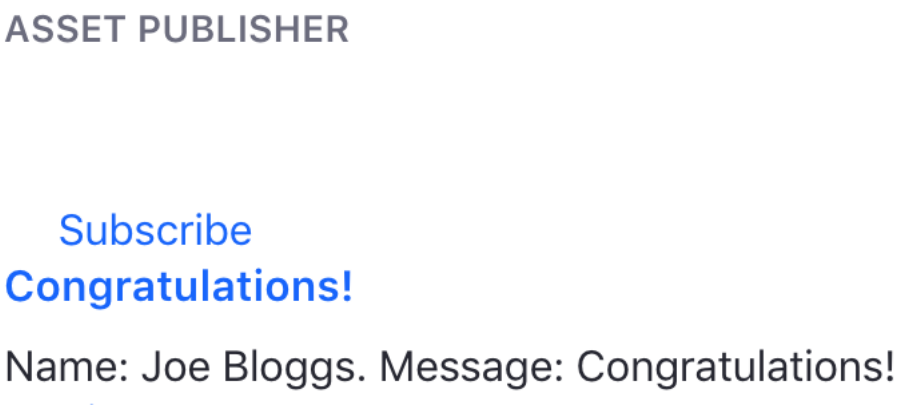


Figure 28.1: After you've implemented and registered your asset renderers for your custom entities, the Asset Publisher can display your entities.

CHAPTER 29

ADDING ASSET FEATURES TO YOUR USER INTERFACE

```
<p id="stepTitle">Adding Asset Features to Your UI</p><p>Step 1 of 5</p>
```

Now that your guestbook and guestbook entry entities are asset-enabled, you can add asset functionality to 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 already implemented. Your only task is to update your applications' user interfaces to use these features.

Now you'll create several new JSPs that need new imports. Add the following imports to the `guestbook-web` module project's `init.jsp` file:

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

<%@ 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" %>
<%@ page import="com.liferay.portal.kernel.comment.Discussion" %>
<%@ page import="com.liferay.portal.kernel.comment.CommentManagerUtil" %>
<%@ page import="com.liferay.portal.kernel.service.ServiceContextFunction" %>
```

Add these imports now so you don't run into errors as you work through this section.

29.1 Creating JSPs for Displaying Custom Assets in the Asset Publisher

```
<p id="stepTitle">Adding Asset Features to Your UI</p><p>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 to JSPs that don't exist yet. These methods return paths to 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 added 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>
```

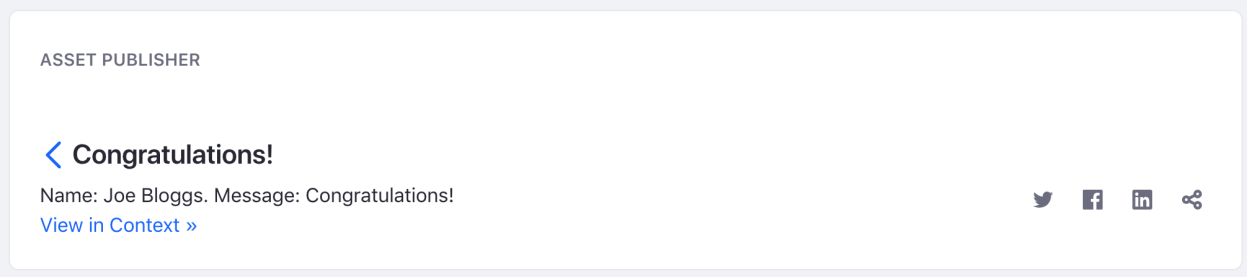


Figure 29.1: When you click the title for a guestbook or guestbook entry in the Asset Publisher, your `full_content.jsp` should be displayed.

This JSP shows a combination of fields from the Guestbook and the selected Entry.

After deploying your changes, 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 social media so you can publicize your asset. The *Back* icon and the *View in Context* link return you to the Asset Publisher's default view.

29.2 Enabling Tags, Categories, and Related Assets for Guestbooks

```
<p id="stepTitle">Adding Asset Features to Your UI</p><p>Step 3 of 5</p>
```

Since you already asset-enabled guestbooks at the service layer, guestbook entities can now support tags and categories. All that's left is to enable them in the UI. In this step, you'll update the Guestbook Admin portlet's `edit_guestbook.jsp` so administrators 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:button-row>` 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. These input tags represent asset categories and asset tags. You can group related input fields together with an <aui:fieldset> tag. The tags generate the appropriate selectors for tags and categories and displays those that have already been added to the guestbook:

```
<aui:fieldset>
    <liferay-asset:asset-categories-selector className="<%= Guestbook.class.getName() %>" classPK="<%= guestbook %>" />
    <liferay-asset:asset-tags-selector className="<%= Guestbook.class.getName() %>" classPK="<%= guestbook %>" />
</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.

29.3 Enabling Tags, Categories, and Related Assets for Guestbook Entries

```
<p id="stepTitle">Adding Asset Features to Your UI</p><p>Step 4 of 5</p>
```

Enabling tags, categories, and related assets for guestbook entries is similar to enabling them for guestbooks. Please 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");

    Entry entry = null;

    if (entryId > 0) {
        entry = EntryLocalServiceUtil.getEntry(entryId);
    }

    long guestbookId = ParamUtil.getLong(renderRequest, "guestbookId");
    %>
```

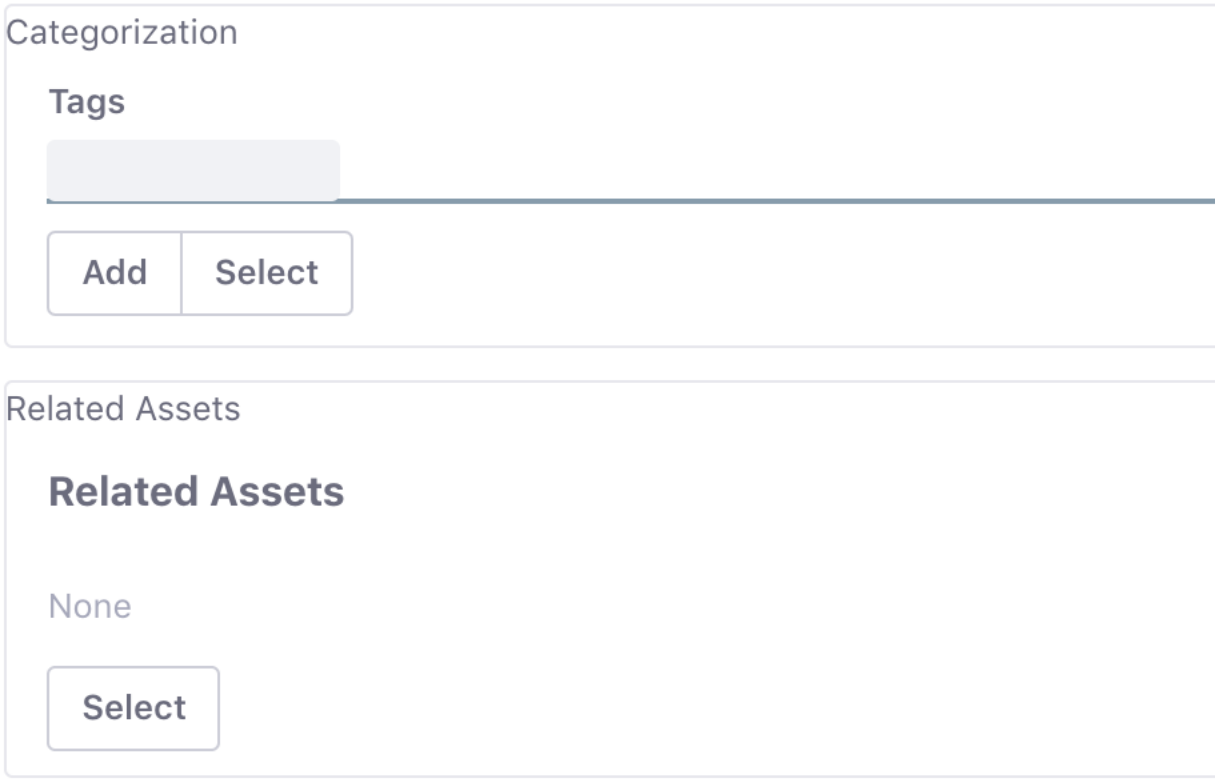


Figure 29.2: Once you've updated your Guestbook Admin portlet's `edit_guestbook.jsp` page, you'll see forms for adding tags and selecting related assets.

```
<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>
                               <liferay-asset:asset-categories-selector className="<%= Entry.class.getName() %>" classPK="<%= entryId %>" />
                               <liferay-asset:asset-tags-selector className="<%= Entry.class.getName() %>" classPK="<%= entryId %>" />
                            </aui:fieldset>
                    </liferay-ui:panel>

                    <liferay-ui:panel defaultState="closed"
                    extended="<%= false %>" id="entryAssetLinksPanel"
                    persistState="<%= true %>" title="related-assets">
                            <aui:fieldset collapsed="<%= true %>" collapsible="<%= true %>" label="related-assets">

                <liferay-asset: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. Add and remove tags, categories, and related assets.

Note: Setting your custom asset as the *Main Asset* of a page is required to display related assets in the Related Assets portlet. This is done when creating Friendly URLs in a later step.

Well done! Next, you'll enable comments and ratings for guestbook entries.

29.4 Enabling Comments and Ratings for Guestbook Entries

```
<p id="stepTitle">Adding Asset Features to Your UI</p><p>Step 5 of 5</p>
```

The 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. Good application design requires that you have a View page where users can rate and comment on assets. 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 Custom 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 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 component with the <liferay-ui:ratings> tag:

```
<liferay-ui:ratings className="<%=Entry.class.getName()%>"
  classPK="<%=entry.getEntryId()%>" type="stars" />

<br />
```

9. Next you need to add a scriptlet to retrieve the comments discussion object:

```
<% Discussion discussion =
CommentManagerUtil.getDiscussion(user.getUserId(),
scopeGroupId, Entry.class.getName(),
entry.getEntryId(), new ServiceContextFunction(request));
%>
```

10. Below that add the tag for tracking the number of comments:

    " key='<%= (discussion.getDiscussionCommentsCount() == 1) ? "x-comment" : "x-comments" %>' />

11. Create the liferay-comment:discussion tag, which creates the comments form, *Reply* button, and retrieves the discussion content. It also handles the form action of posting the comment without requiring you to create a portlet action URL.

```
  <liferay-comment:discussion
    className="<%= Entry.class.getName() %>"
    classPK="<%= entry.getEntryId() %>"
    discussion="<%= discussion %>"
    formName="fm2"
    ratingsEnabled="true"
    redirect="<%= currentURL %>"
    userId="<%= entry.getUserId() %>"
    />

  </liferay-ui:panel>
</liferay-ui:panel-container>
```

12. 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 the `MBMessage` table, there's a `classPK` column. This `classPK` represents the guestbook entry's `entryId`, linking the comment to the guestbook. Ratings are stored in the `RatingsEntry` table. Similarly, the `RatingsEntry` table contains a `classPK` column that links the guestbook entry to the rating. Using a `classPK` foreign key in one table to represent the primary key of another table is a common pattern throughout Liferay DXP.

Next, you'll update the guestbook actions to use the new view.

Updating the Entry Actions JSP

Your `view_entry.jsp` page is 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 your comments and ratings.

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.

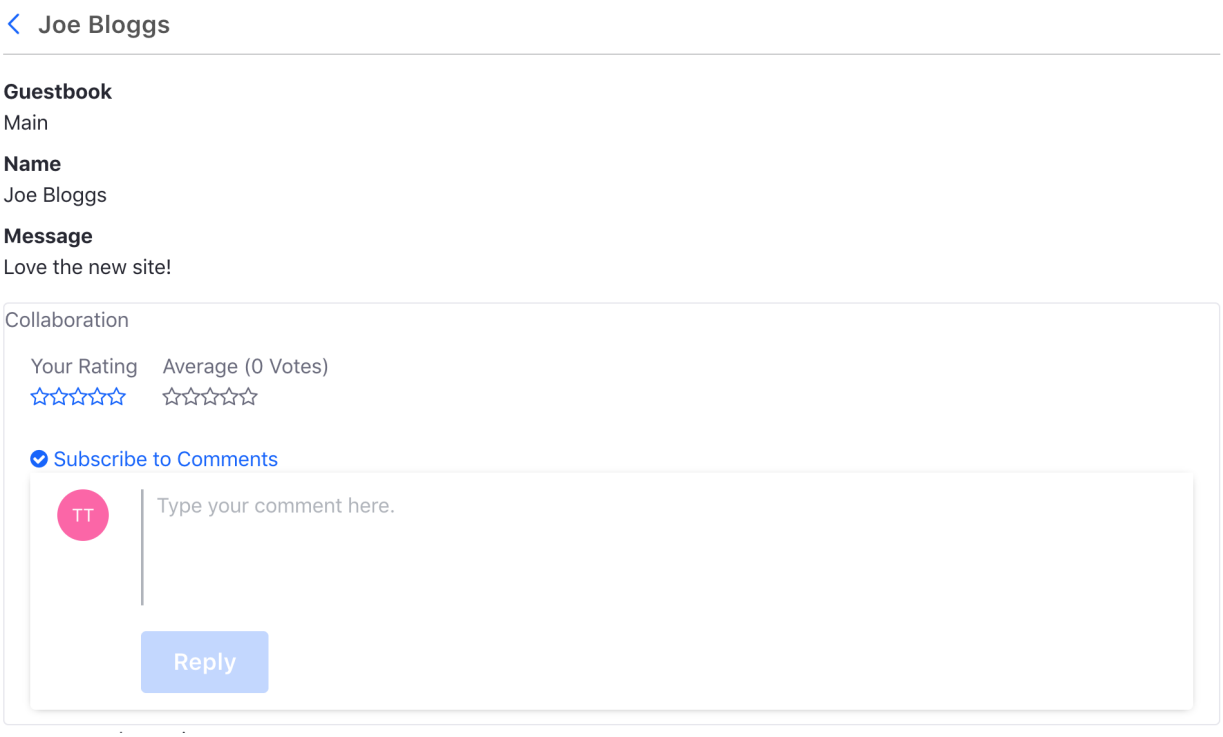


Figure 29.3: Now you can see comments, rating, and the full range of asset features.

CHAPTER 30

TOOLING

You can write code for Liferay using any standard toolset. Liferay is tool-agnostic, rather than pigeonholing you into something specific. This frees you to work with whatever you're already productive using.

Liferay has also created its own tools that streamline Liferay DXP development. These tools integrate with popular build environments (e.g., Gradle and Maven). They include

- Liferay Dev Studio DXP: an Eclipse-based IDE supporting development for Liferay DXP.
- Blade CLI: a command line interface bootstrapped on to a Gradle based environment that is used to build and manage Liferay Workspaces and Liferay DXP projects.
- Liferay Workspace: a generated environment built to hold and manage Liferay DXP projects.
- Liferay IntelliJ plugin: a plugin providing support for Liferay DXP development with IntelliJ IDEA.

Liferay also provides a plethora of Gradle and Maven plugins you can apply to your projects. Many of these are already built into tools such as Liferay Workspace.

Want samples or predefined project templates? Liferay has you covered with 30+ project templates and many more project samples.

If you're a newbie looking for the best development tool for Liferay DXP, 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 31

LIFERAY DEV STUDIO DXP

Liferay Dev Studio DXP provides an all-in-one, integrated development environment based on Eclipse for Liferay DXP. Dev Studio includes Liferay IDE plugins paired with a pre-installed Liferay Digital Enterprise server.

@ide@ works with build tools such as Gradle and Maven and configuration tools like BndTools.

Dev Studio makes Liferay development easier. There are editors for Service Builder files, workflow definitions, POM files, and more. You'll find wizards for creating every kind of Liferay project there is, snippets for tag libraries, and auto-deploy of changes to plugins.

In this section of tutorials, you'll learn how to install Liferay Dev Studio DXP and develop/manage modules using Liferay Workspace and other technologies.

31.1 Installing Liferay Dev Studio DXP

Liferay Dev Studio DXP 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. Before beginning the installation process, view Dev Studio's Compatibility Matrix to get acquainted with its supported Liferay versions and application servers.

In this tutorial, you'll learn the different methods available for installing Liferay Dev Studio:

- install the Dev Studio bundle from scratch
- install Dev Studio into an existing Eclipse instance using an update URL
- install Dev Studio into an existing Eclipse instance using a ZIP file

Important: If you're installing Dev Studio into an existing Eclipse environment, you must be on Eclipse Oxygen or newer. For instructions on upgrading to Oxygen, see Eclipse's upgrade documentation. With this particular upgrade, you should also deactivate the current available update sites in the *Window → Preferences → Install/Update → Available Software Sites* menu to ensure a successful upgrade (e.g., Neon).

Install the Liferay Dev Studio 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 Dev Studio, 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 Liferay's latest 3.2.x Project SDK with Dev Studio DXP executable that correlates to your operating system. The Project SDK includes Dev Studio DXP, Liferay Workspace, and Blade CLI.

   You may be prompted for your liferay.com username and password before downloading the Liferay DXP installer. Since Dev Studio DXP 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 Dev Studio'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. Select the Java Runtime to use for the installation process. Then click *Next*.

5. Click *Next* to begin the installation process. Select the installation folder for your Liferay Dev Studio instance. Then click *Next*.

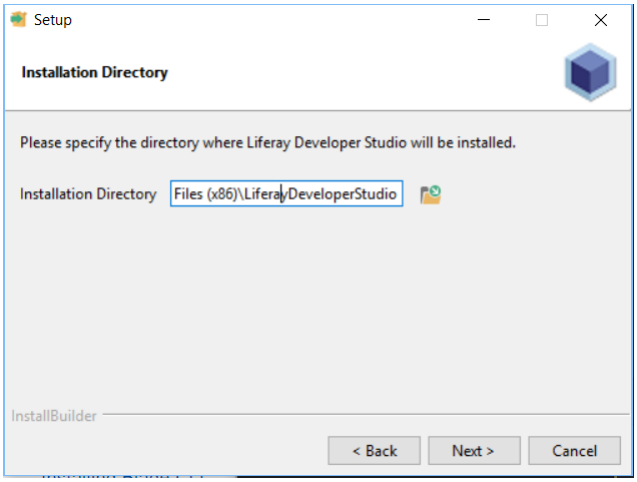


Figure 31.1: Choose the folder your Dev Studio instance should reside.

6. Input the Liferay DXP activation key to set up the Liferay DXP bundle packaged with Dev Studio DXP. Then click *Next*.

   Dev Studio installs 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).

7. Configure proxy settings for your Project SDK. If you must use Dev Studio behind a firewall, you may want to configure the proxy settings. See the Liferay IDE Proxy Settings and Liferay Workspace Proxy Settings tutorials for more information. Skip this step if you don't need this.

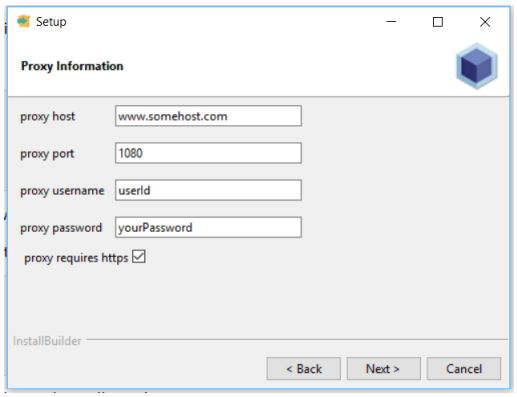


Figure 31.2: Configure your Project SDK's proxy settings, if necessary.

8. Click *Next* to finish the installation process for your Dev Studio instance.

Congratulations! You've installed Liferay Dev Studio DXP! It's now available in the folder you specified. To run Dev Studio, execute the `DeveloperStudio` executable. A Liferay Workspace has also been initialized in that same folder.

Install Liferay Dev Studio into Eclipse Environment

To install Dev Studio 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/milestone/.
3. You'll see the Dev Studio components in the list below. Check them off and click *Next*.
4. Accept the terms of the agreements. Click *Next*, and Dev Studio is installed. Like other Eclipse plugins, you must restart Eclipse to use them.

Liferay Dev Studio is now installed in your existing Eclipse environment.

Install Liferay Dev Studio into Eclipse from a ZIP File

To install Dev Studio using a Zip file, follow these steps:

1. Go to the Liferay Dev Studio DXP downloads page. From the drop-down menu, select *Developer Studio Updatesite 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 Dev Studio Update Site `.zip` file. Then press *OK*.
4. You'll see the Dev Studio components in the list below. Check them off and click *Next*.

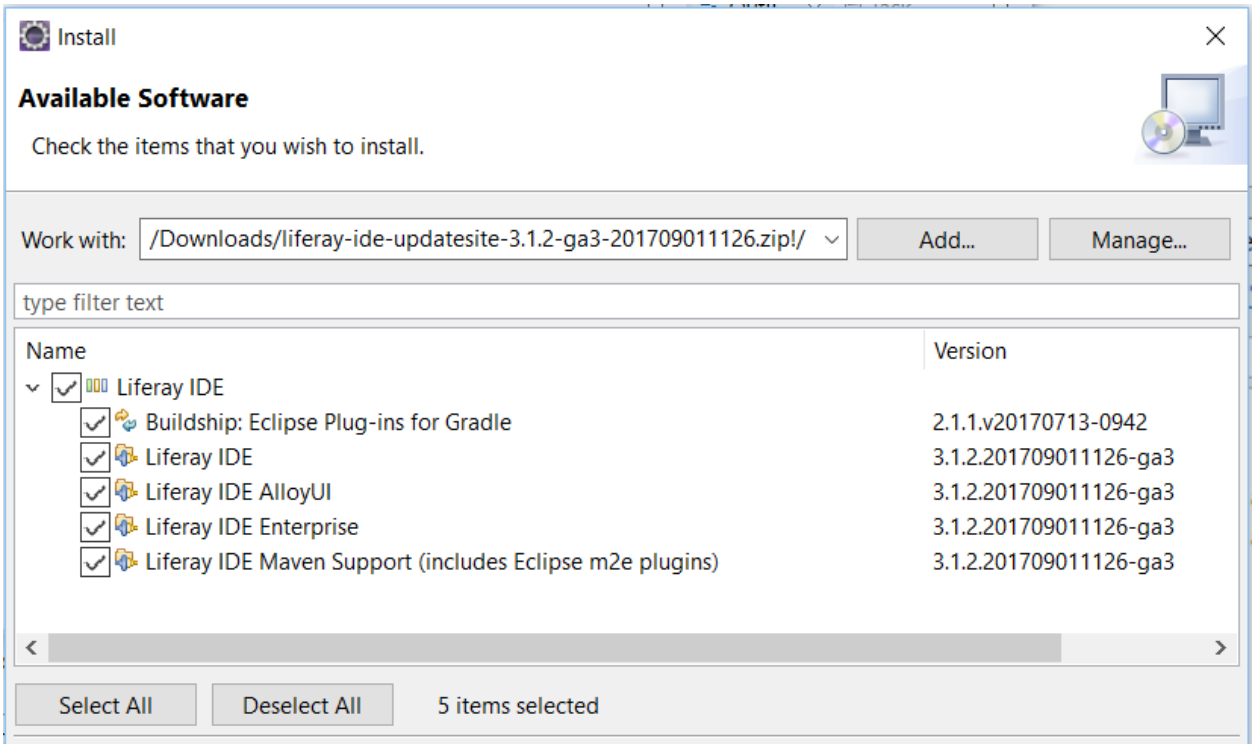


Figure 31.3: Make sure to check all the Dev Studio components you wish to install.

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 Dev Studio in your existing Eclipse environment.

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. Hover over your profile picture in the top-right corner 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.

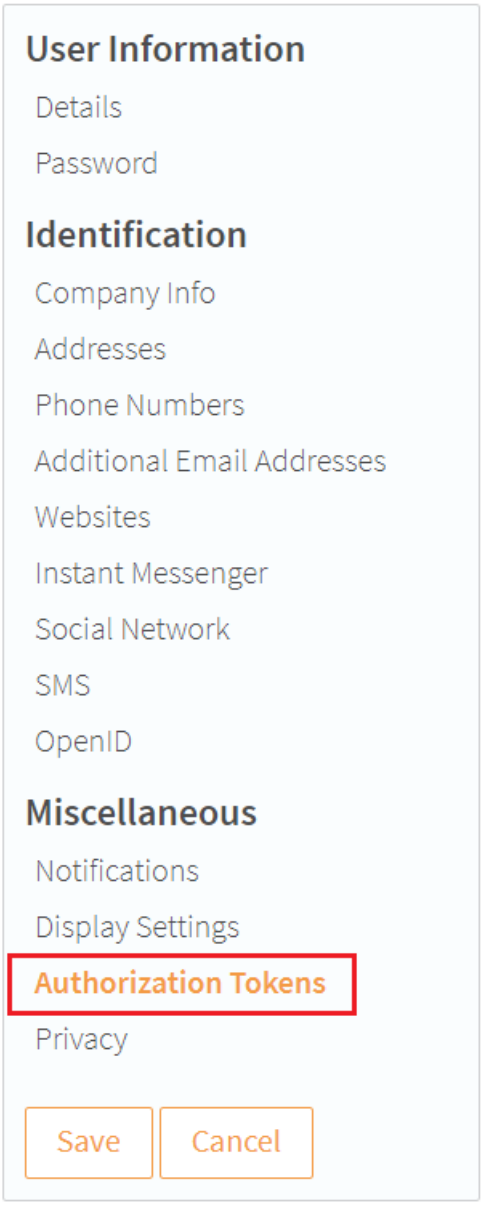


Figure 31.4: You can manually create your workspace token in the Authorization Tokens menu.

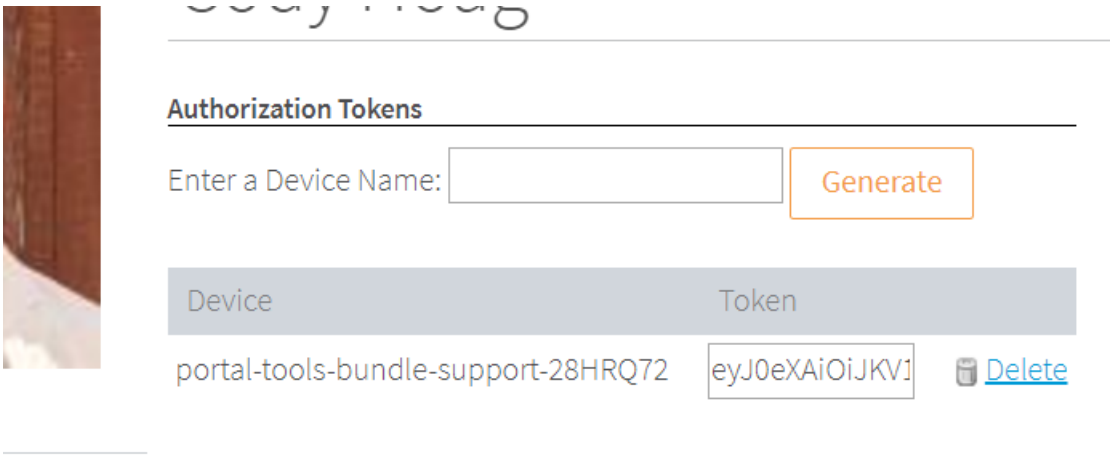


Figure 31.5: The generated token is available to copy.

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.

31.2 Creating a Liferay Workspace with Dev Studio

In this tutorial, you'll learn how to generate a Liferay Workspace using Liferay Dev Studio, which runs on the Blade CLI behind the scenes. Dev Studio 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 new perspectives designed for Liferay DXP development: the *Liferay Workspace* and *Liferay Plugins* perspectives. If you plan on using a Liferay Workspace for your Liferay DXP development, you should select the *Liferay Workspace* perspective (default). This offers development tools that are helpful when using a Liferay Workspace. The *Liferay Plugins* perspective is for developers using Ant-based development tools such as the Plugins SDK. Since the Plugins SDK is only provided for Liferay Portal/DXP 7.0 and older development, this should not be used for 7.0 development.

To create a Liferay Workspace in Dev Studio, select *File → New → Liferay Workspace Project*.

A New Liferay Workspace dialog appears, presenting several configuration options. Follow the instructions below to create your workspace.

1. Give your workspace project 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. Select the build tool you want your workspace to be build with (i.e., Gradle or Maven).

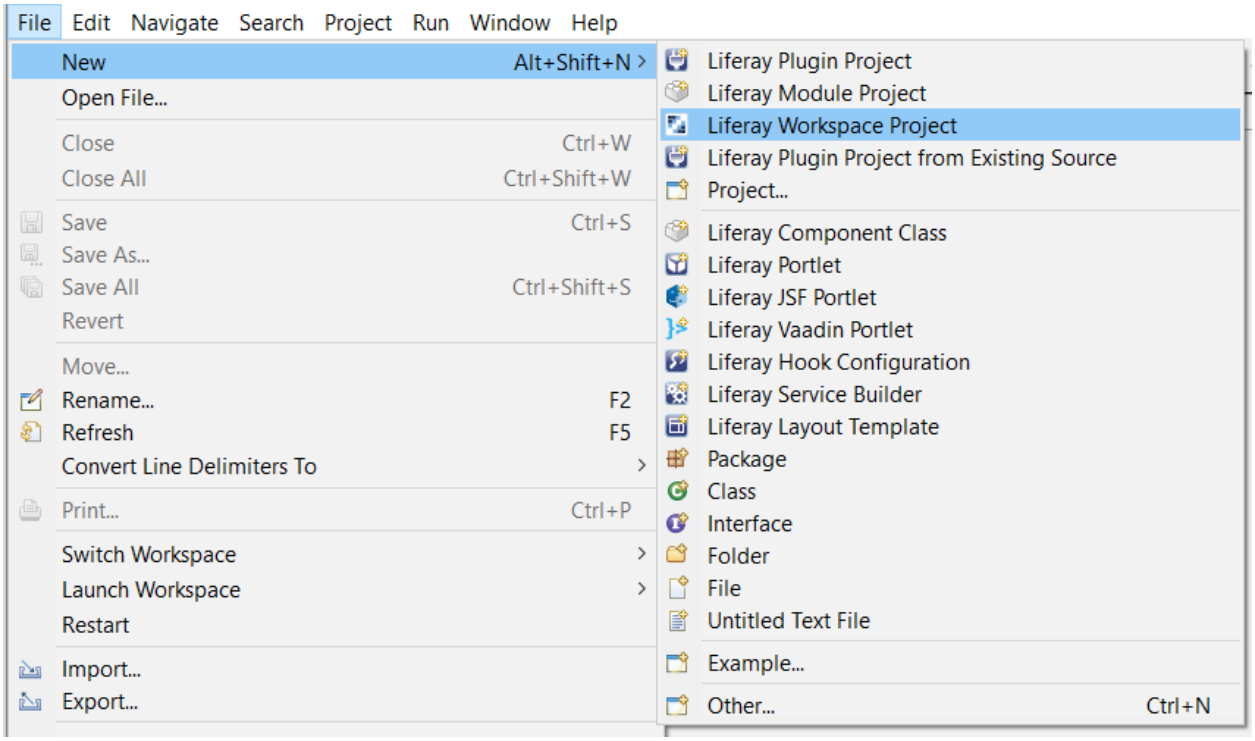


Figure 31.6: By selecting *Liferay Workspace*, you begin the process of creating a new workspace for your Liferay projects.

4. Choose the Liferay DXP version you plan to develop for (i.e., 7.1 or 7.0).

5. Select the specific target platform version corresponding to the GA release you're developing for (e.g., 7.1.0 → 7.1 GA1). For more information on target platform benefits, see the Managing the Target Platform for Liferay Workspace articles.

6. 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 and provide the server's download URL, if selected. This Liferay bundle is generated the same way as described in the previous section.

```
**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.
```

---

7. Check the *Add project to working set* checkbox if you want the workspace to be a part of a larger working set you've already created in Dev Studio. For more information on working sets, visit Eclipse Help.

8. Click *Finish* to create your Liferay Workspace.

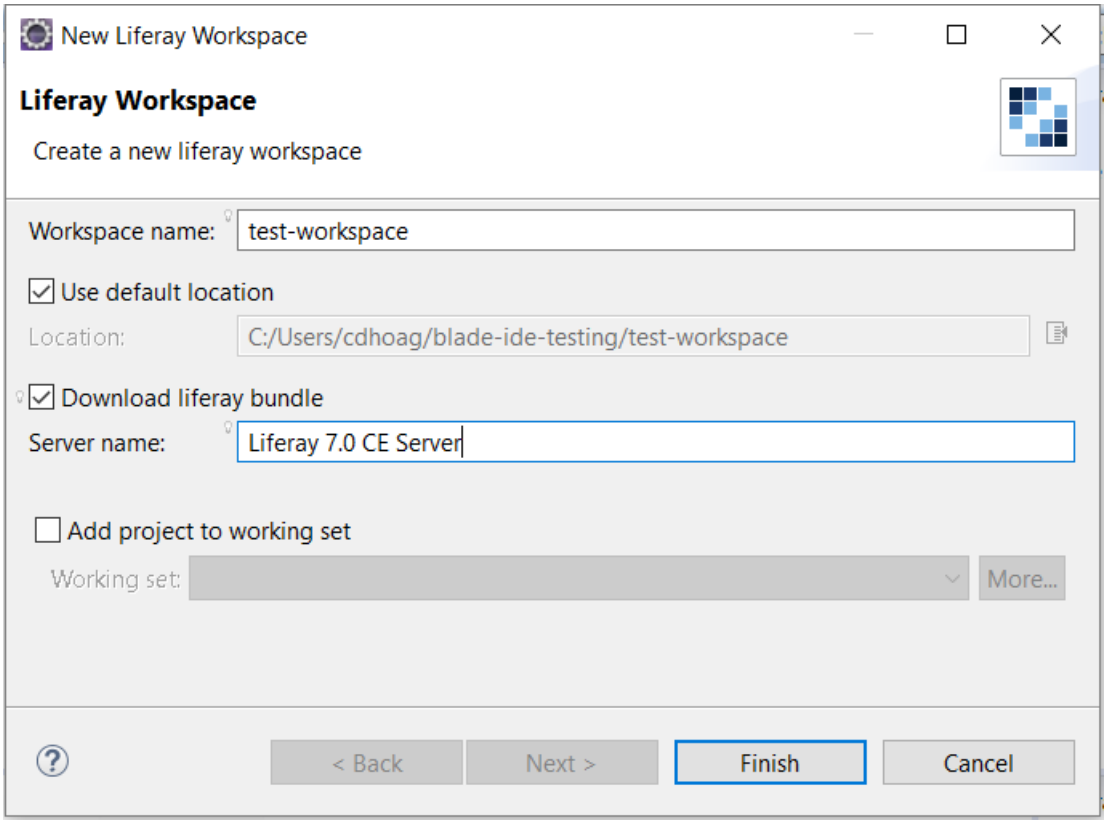


Figure 31.7: Liferay Dev Studio provides an easy-to-follow menu to create your Liferay Workspace.

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 Dev Studio!

Liferay Workspace Settings in Dev Studio

The Liferay Workspace perspective is intended for Gradle or Maven projects for Liferay DXP. Since Liferay Workspaces are used for Gradle/Maven 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.

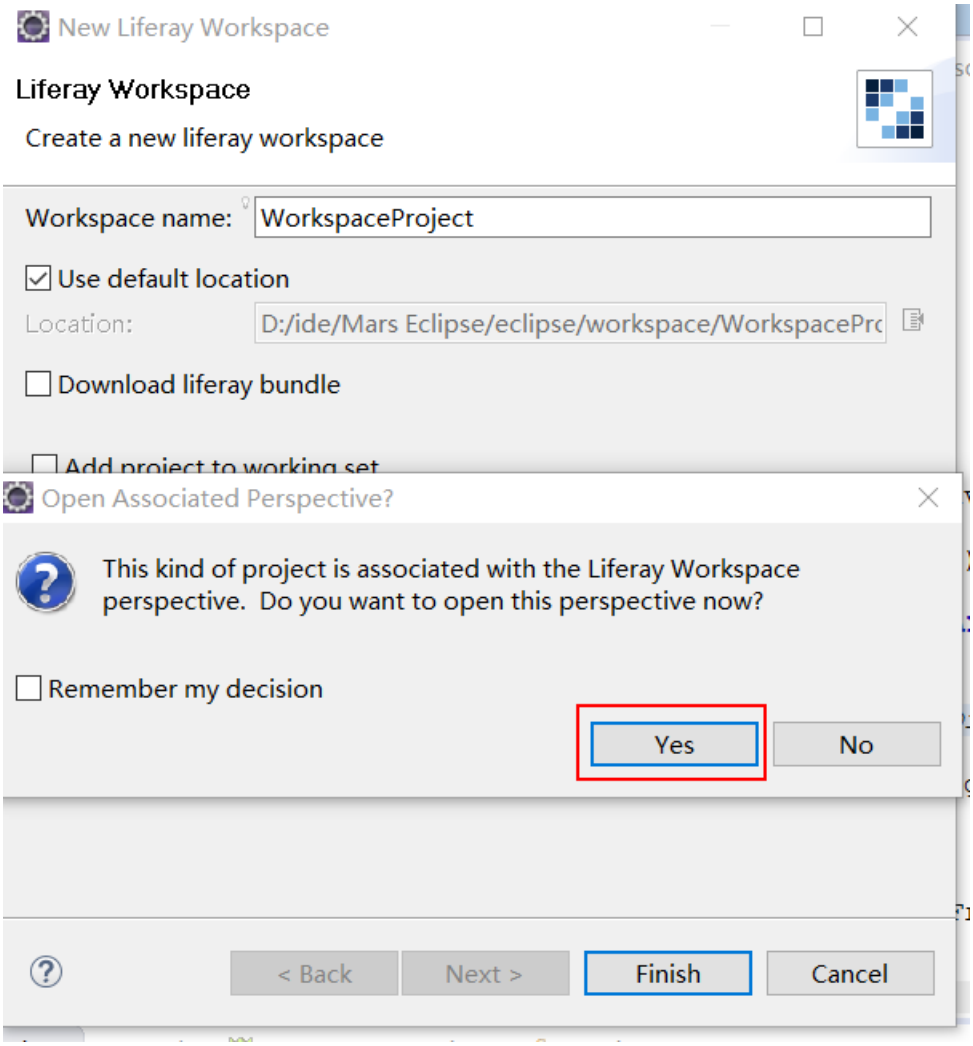


Figure 31.8: The Liferay Workspace perspective is preferred for 7.0 and OSGi module development.

If you've already created a Liferay Workspace and you'd like to import it into your existing Dev Studio, 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 Dev Studio. Now that your workspace is created, you can begin creating Liferay projects.

!VVideo Tutorial

31.3 Setting Proxy Requirements for Liferay Dev Studio

If you have proxy server requirements and want to configure your http(s) proxy to work with Liferay Dev Studio, 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*.

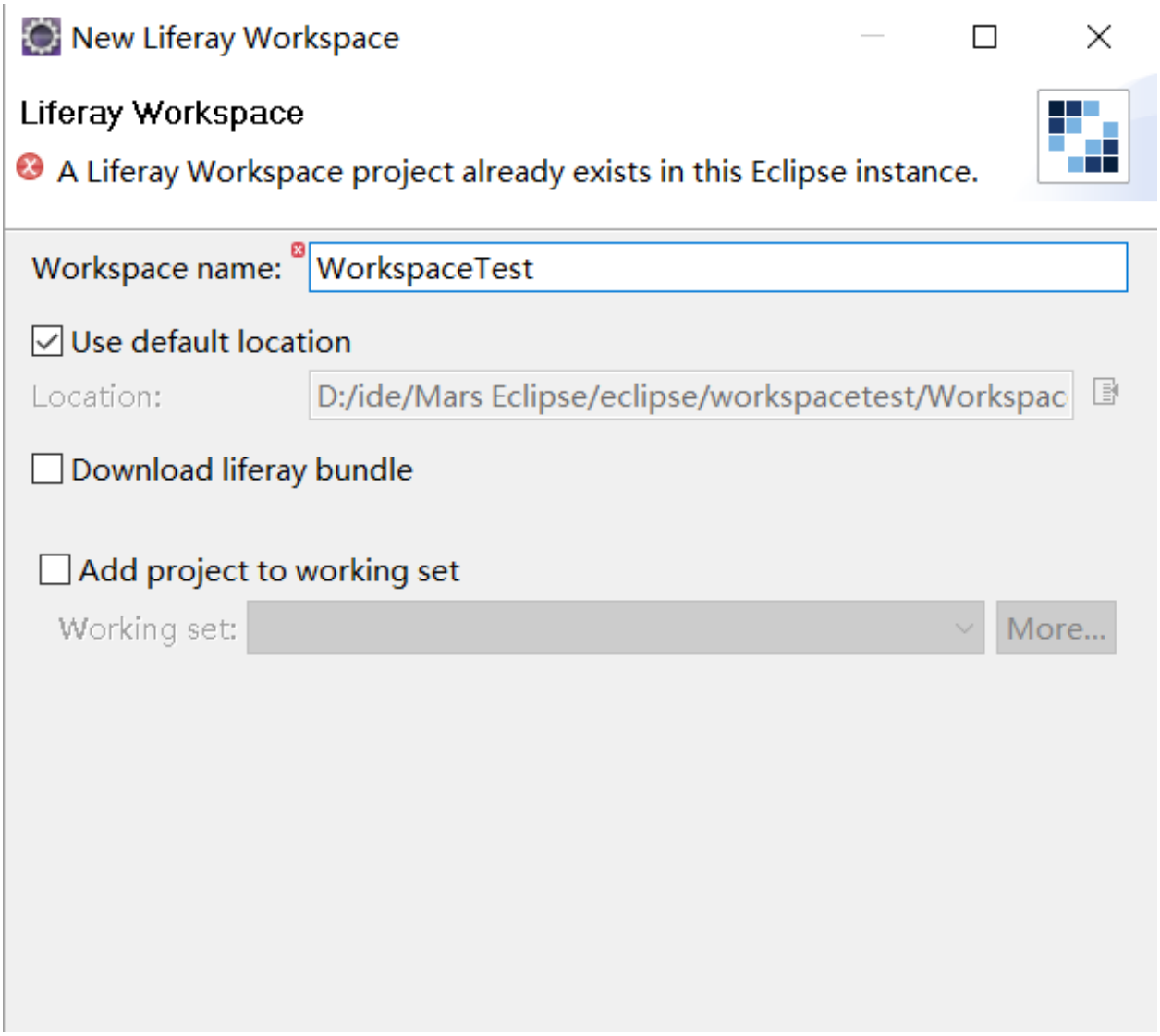


Figure 31.9: A Dev Studio workspace only supports one Liferay Workspace project. If you create another, you'll be given an error message.

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*.

If you're working with a Liferay Workspace in Dev Studio, 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 Dev Studio's proxy settings!

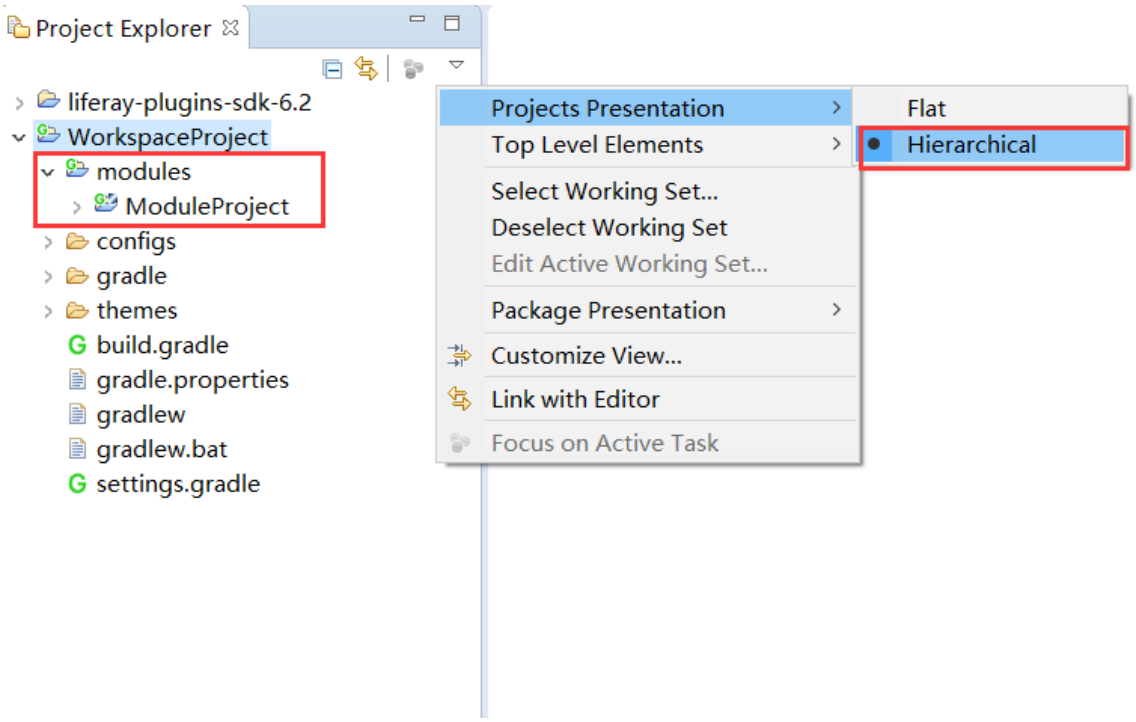


Figure 31.10: The Hierarchical project presentation mode is set, by default.

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!

31.4 Updating Liferay Dev Studio

If you're already using Liferay Dev Studio but need to update your environment, follow the steps below:

1. In Dev Studio, go to *Help → Install New Software...*.

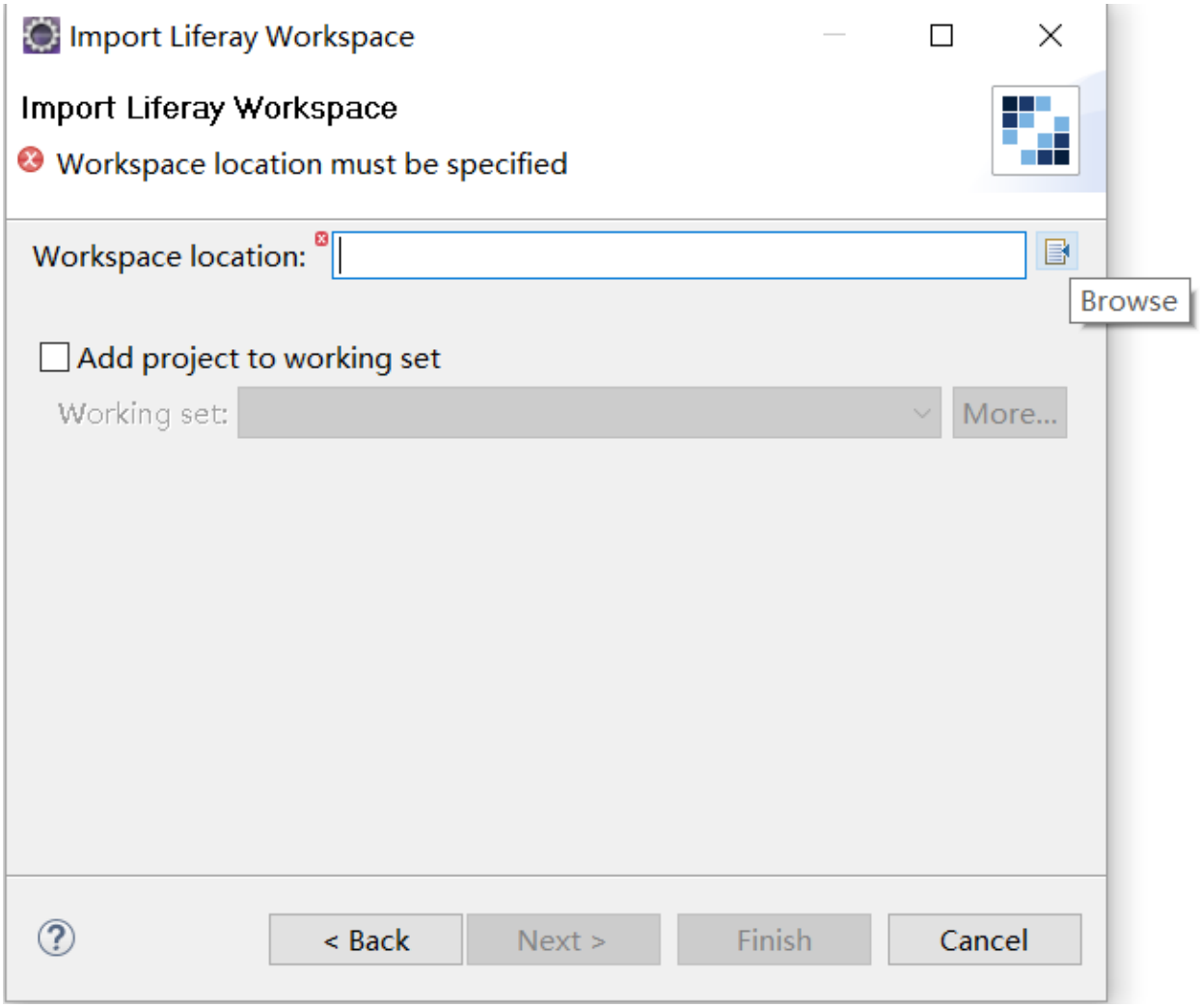


Figure 31.11: You can import an existing Liferay Workspace into your current Dev Studio session.

2. In the *Work with* field, copy in the URL http://releases.liferay.com/tools/ide/latest/stable/.
3. You'll see the Dev Studio components in the list below. Check them off and click *Next*.
4. Accept the terms of the agreements. Click *Next*, and Dev Studio is updated. You must restart Dev Studio for the updates to take effect.

You're now on the latest version of Liferay Dev Studio!

31.5 Creating Modules with Liferay Dev Studio

Dev Studio 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*.

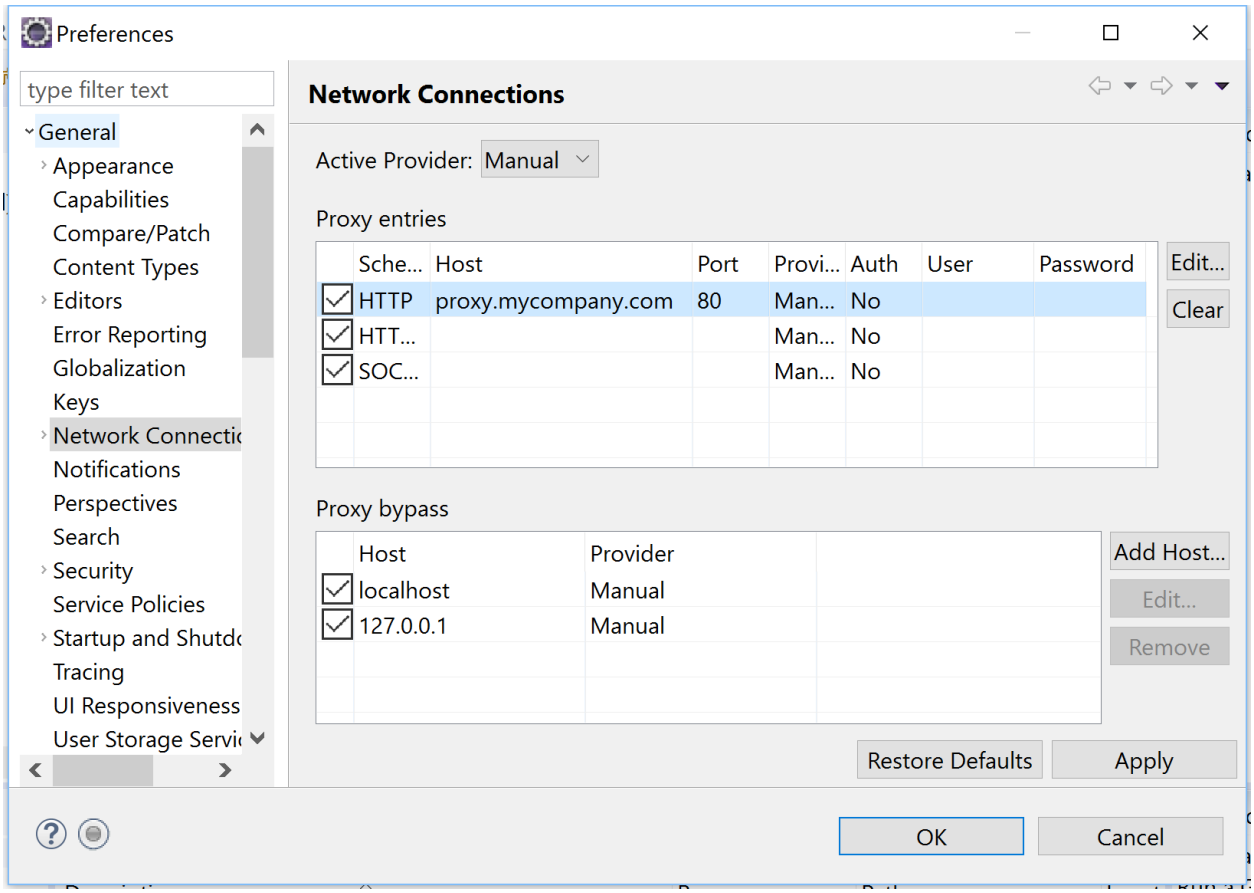


Figure 31.12: You can configure your proxy settings in Dev Studio's Network Connections menu.

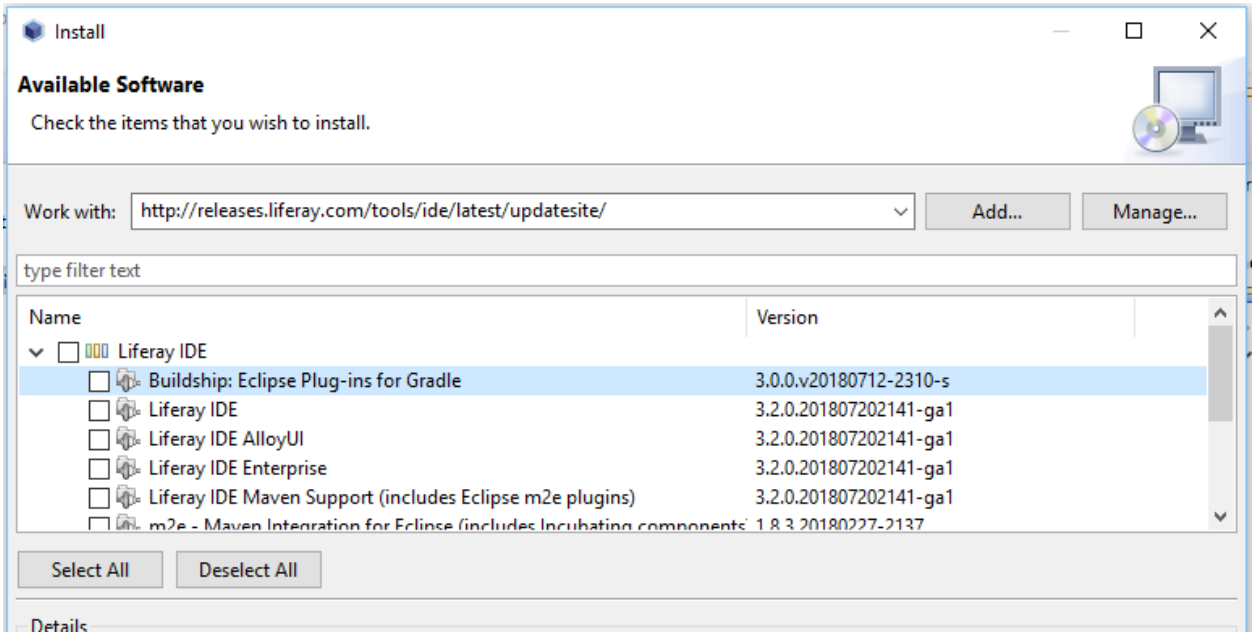


Figure 31.13: Make sure to check all the Dev Studio components you wish to install.

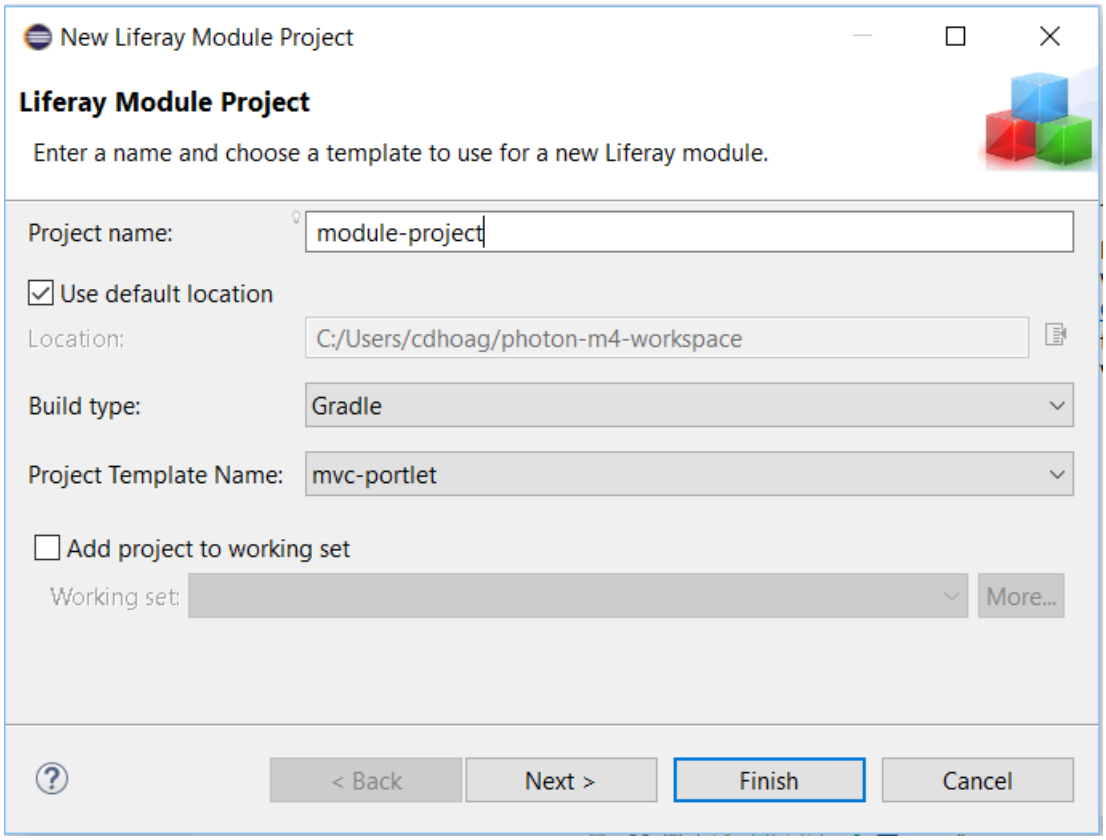


Figure 31.14: When selecting *New → Liferay Module Project*, a Module Project Wizard appears.

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 based on the project template you selected. For example, if you selected a template that requires a component class, you must configure it in the wizard.

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. See more about creating a component class in Liferay Dev Studio in the Creating Component Classes section.

Once you've configured your module, click *Finish* to create your project.

Now that you've created your module project, you can configure your project's presentation in the Dev Studio'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.

You now have the knowledge to create a Liferay module project from Liferay Dev Studio.

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,

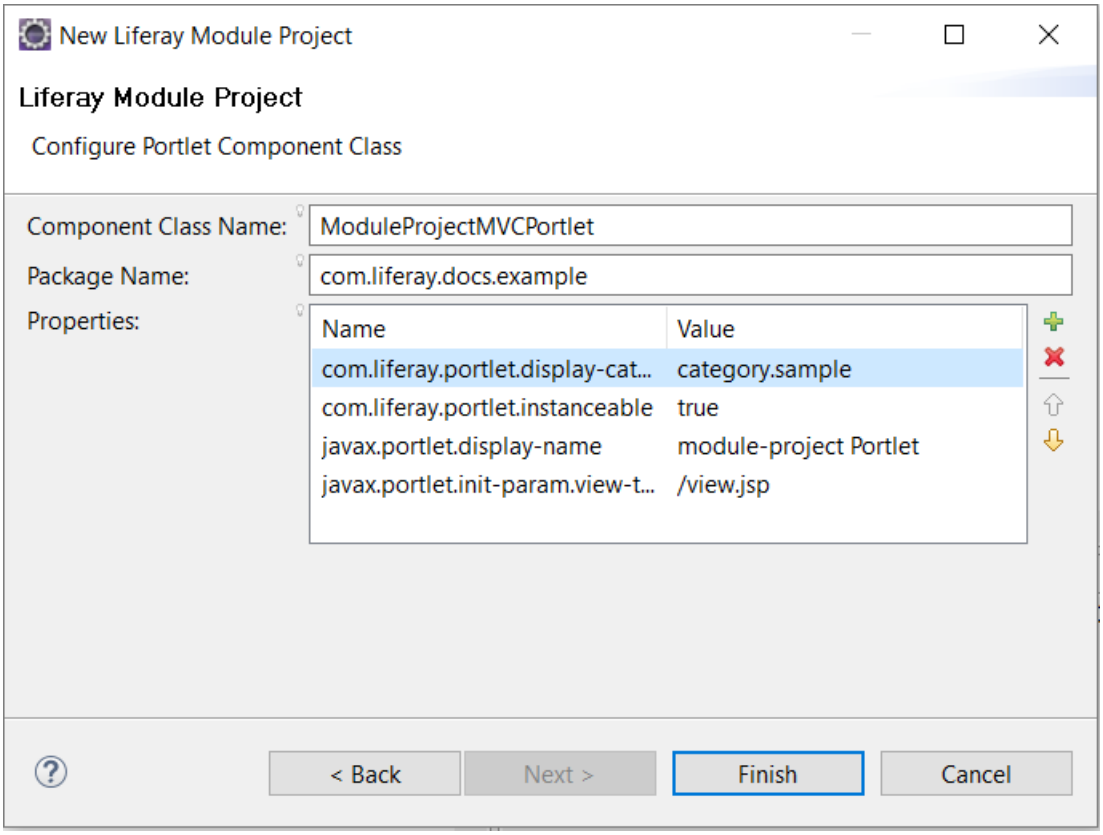


Figure 31.15: Specify your component class's details in the Portlet Component Class Wizard.

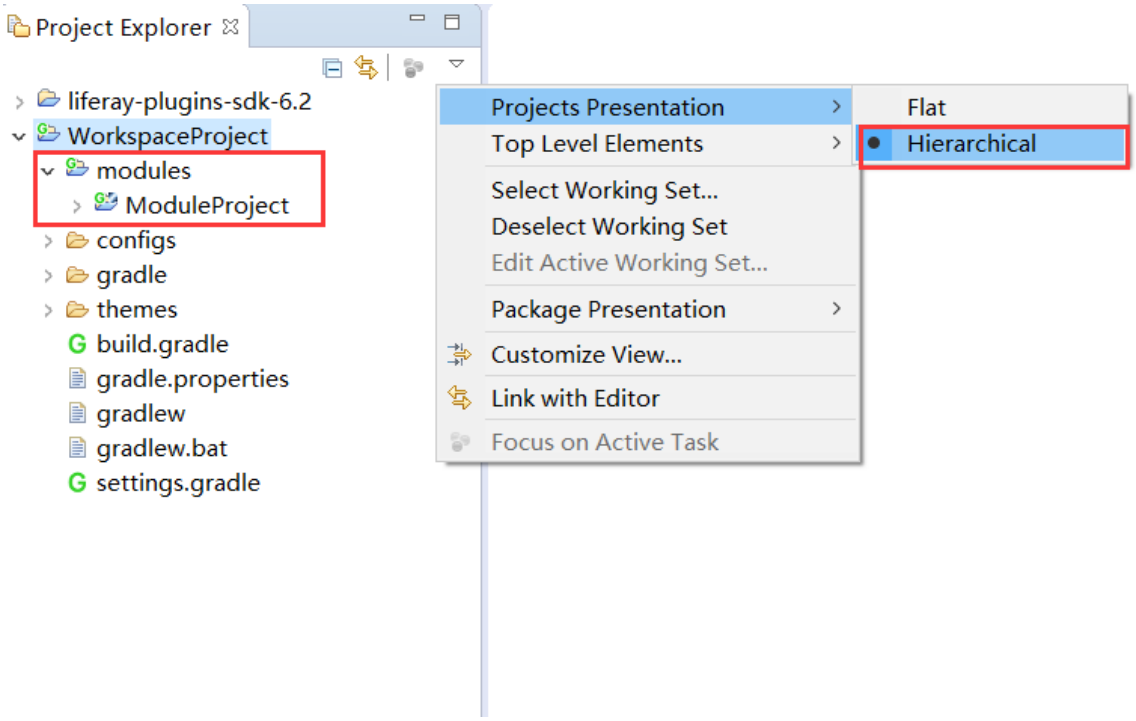


Figure 31.16: The Hierarchical project presentation mode is set, by default.

except you can select a component class template. There are many templates in the `Component Class Template` list:

- *Auth Failures*: 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
- *MVC Portlet*: creates a new MVC portlet
- *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

Next you'll learn how to import existing projects into Dev Studio.

Importing Existing Module Projects

Dev Studio 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're now equipped to import module projects into Liferay Dev Studio. Now go out there and get stuff done!

31.6 Creating Themes with Liferay Dev Studio

Liferay Dev Studio lets you create and configure Liferay theme projects. You can create a theme standalone or in a Liferay Workspace. You can even create a Gradle or Maven based theme! Read on to learn more about creating themes in Dev Studio.

1. In Dev Studio, 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* or *Maven*.

3. Select *Finish*.

That's it! You've created a theme project in Dev Studio! Learn how to deploy it in this tutorial.

If you've configured a Liferay Workspace in your Dev Studio instance, your theme is available in the workspace's `wars` folder by default. If you don't have a workspace configured in Dev Studio, it's available in the root of Dev Studio's Project Explorer.

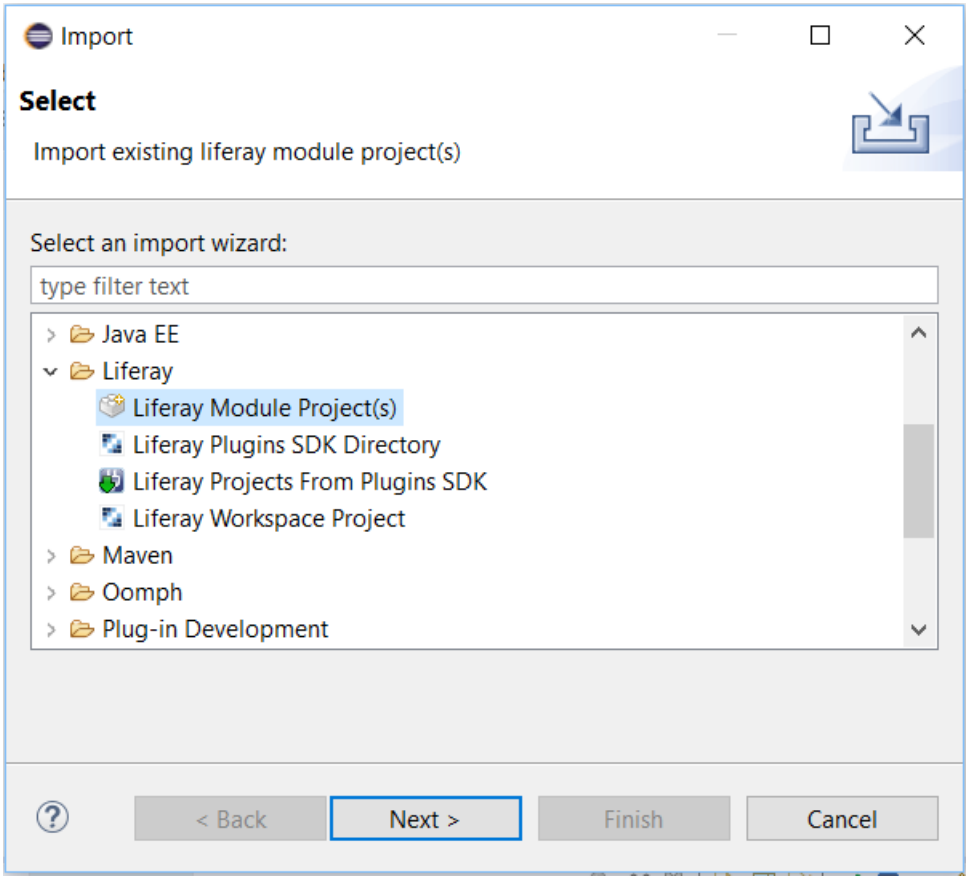


Figure 31.17: Select the *Liferay Module Project(s)* to import a module project.

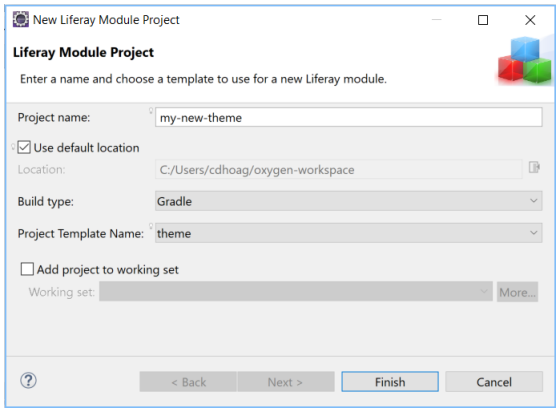


Figure 31.18: Use the theme project template to create a Liferay theme in Dev Studio.

Note that themes created in Dev Studio follow a WAR-style layout. Although the wizard can be misleading by calling the theme a new module project, it is a WAR.

To modify a theme created in Dev Studio, mirror the folder structure of the files you wish to change and copy them into your theme's `webapp` folder.

Important: Under the hood, Dev Studio is using the theme project template. The WAR-style theme created by Dev Studio is Gradle/Maven based; this differs from themes created with the Liferay Theme Generator that use the Liferay JS Theme Toolkit. Do not mix these two development strategies. See the Managing Themes in Liferay Workspace tutorial for more information on how these two strategies are used in Workspace and Dev Studio.

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.

31.7 Deploying Projects with Liferay Dev Studio

Deploying projects in Liferay Dev Studio is a cinch. Before deploying your project, make sure you have a Liferay server configured in Dev Studio. To see how to do this, see the Installing a Server in Liferay Dev Studio

There are two ways to deploy a project to your Liferay server. You should start your Liferay server before attempting to deploy to it.

1. Select the project 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 project(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 Dev Studio, make sure to follow the methods described above.

If you're deploying a project using Liferay Workspace, the `watch` Blade CLI task is used to deploy your project. This watches your local project and quickly propagates any saved changes to the deployed project. With this, project updates are viewable almost instantaneously from your Liferay server. For more info on the `watch` task, see the Deploying Projects with Blade CLI article.

Note: You can deploy standalone projects not residing in a Liferay Workspace with the `watch` task by right-clicking the project and selecting *Liferay → watch*.

That's it! Once your project is deployed to the Liferay server, you can verify its installation in Dev Studio's Console window.

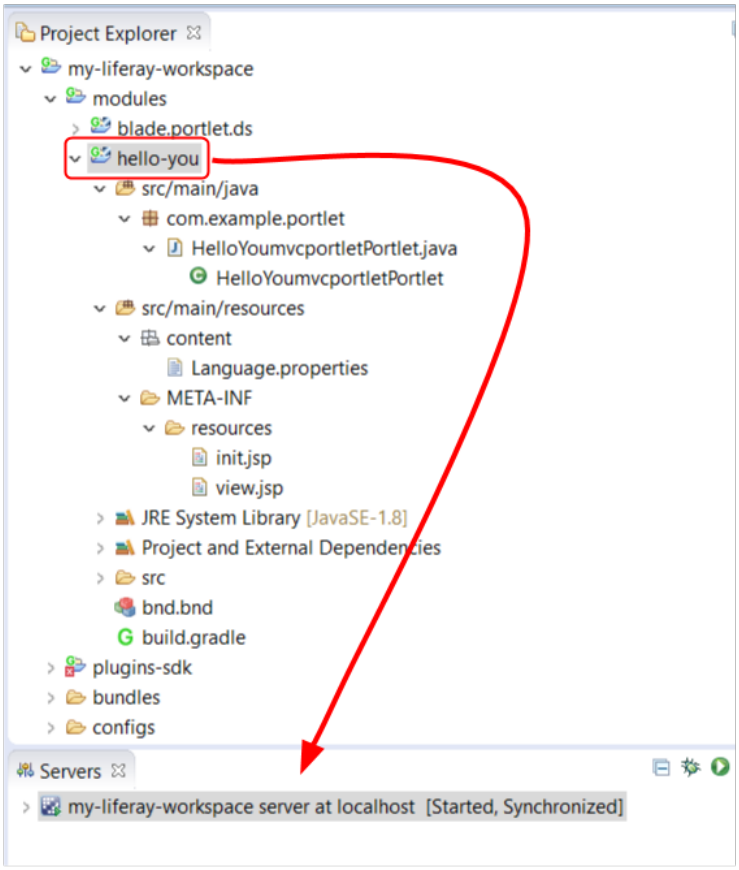


Figure 31.19: You can use the drag-and-drop method to deploy your project to Liferay DXP.

31.8 Managing Projects with Liferay Dev Studio

Liferay Dev Studio provides the ability to manage Liferay projects from a GUI. Before you begin learning about managing your projects from Liferay Dev Studio, 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 Dev Studio tutorial.

Once you've created projects, you can deploy them using Dev Studio. First, make sure your Liferay server is started by clicking the *Start Server* button ( ). Then navigate to your project from the Project Explorer and drag-and-drop it 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 project to your running Liferay instance!

For the deployed project, you can check if it has been deployed successfully by using the Gogo Shell. Right-click the started portal in your 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 projects deployed to it. Enter the `lb` command to view a list of deployed bundles.

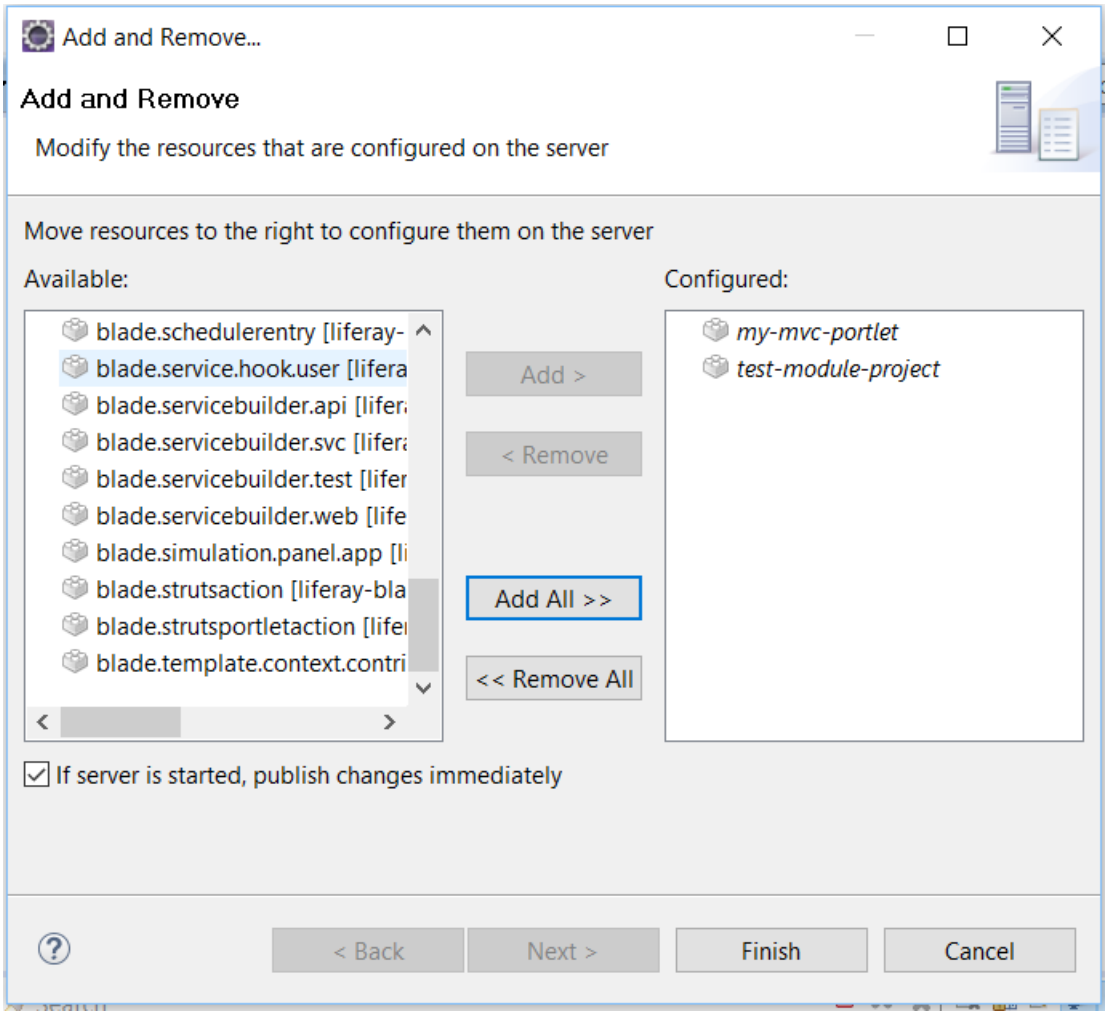


Figure 31.20: Using the this deployment method is convenient when deploying multiple projects.

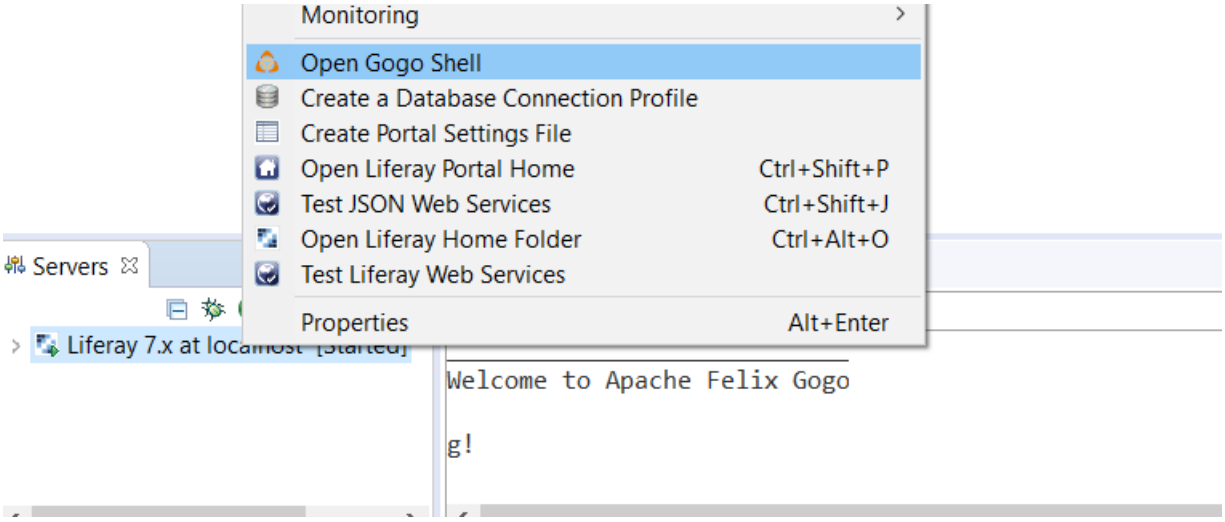


Figure 31.21: Select *Open Gogo Shell* to open a terminal window in Dev Studio using Gogo shell.

If the project status is active, then it deployed successfully.

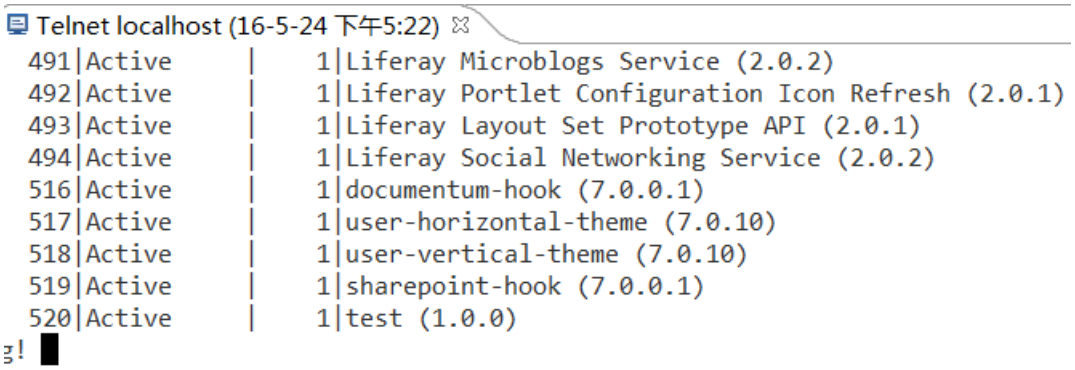


Figure 31.22: You can check to see if your project deployed successfully to Liferay using the Gogo shell.

Dev Studio's Gogo shell usage requires Developer Mode to be enabled. Developer Mode is enabled in Liferay Workspace by default.

Since the Liferay Workspace perspective in Dev Studio 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.

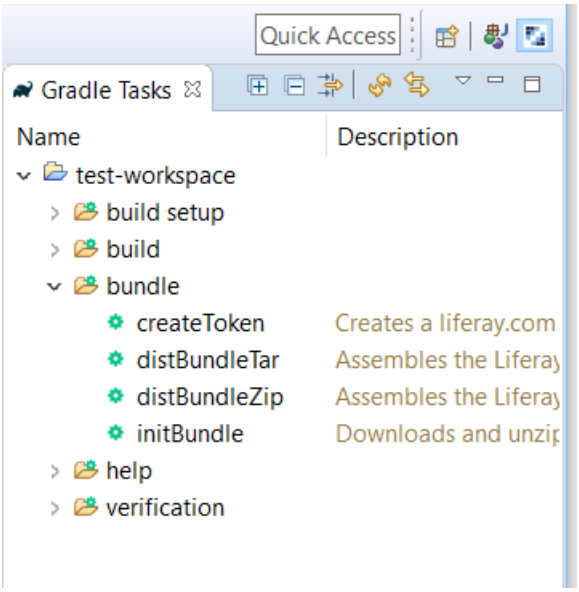


Figure 31.23: The Gradle Task toolbar offers Gradle tasks and their descriptions, which can be executed by double-clicking them.

You can also access various Gradle build operations intended for Liferay module projects. Right-click your module project and select *Liferay* and then the build command you want to execute.

To learn more about Gradle development in Liferay Dev Studio, see the Using Gradle in Liferay Dev Studio tutorial.

Excellent! You've learned how to manage your Gradle-based Liferay Workspace using Dev Studio.

31.9 Installing a Server in Liferay Dev Studio

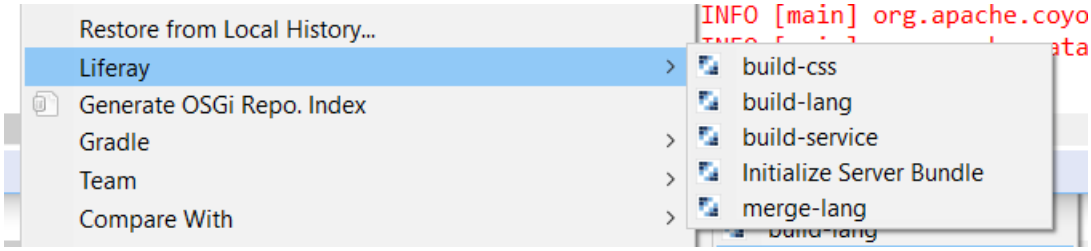


Figure 31.24: You can execute build operations by right-clicking the Gradle project in the Project Explorer.

Installing a server in Liferay Dev Studio 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 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 Dev Studio. 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-7.1-m1\tomcat-8.0.32`.

5. Select a runtime JRE and click *Finish*. Your new server appears under the Servers view.

Your server is now available in Liferay Dev Studio!
For reference, here's how the Dev Studio server buttons work with your Liferay DXP instance:

- *Start* ( ): Starts the server.
- *Stop* ( ): Stops the server.
- *Debug* ( ): Starts the server in debug mode. For more information on debugging in Dev Studio, see the Debugging Liferay DXP source in Liferay Dev Studio article.

Now you're ready to use your server in Liferay Dev Studio!

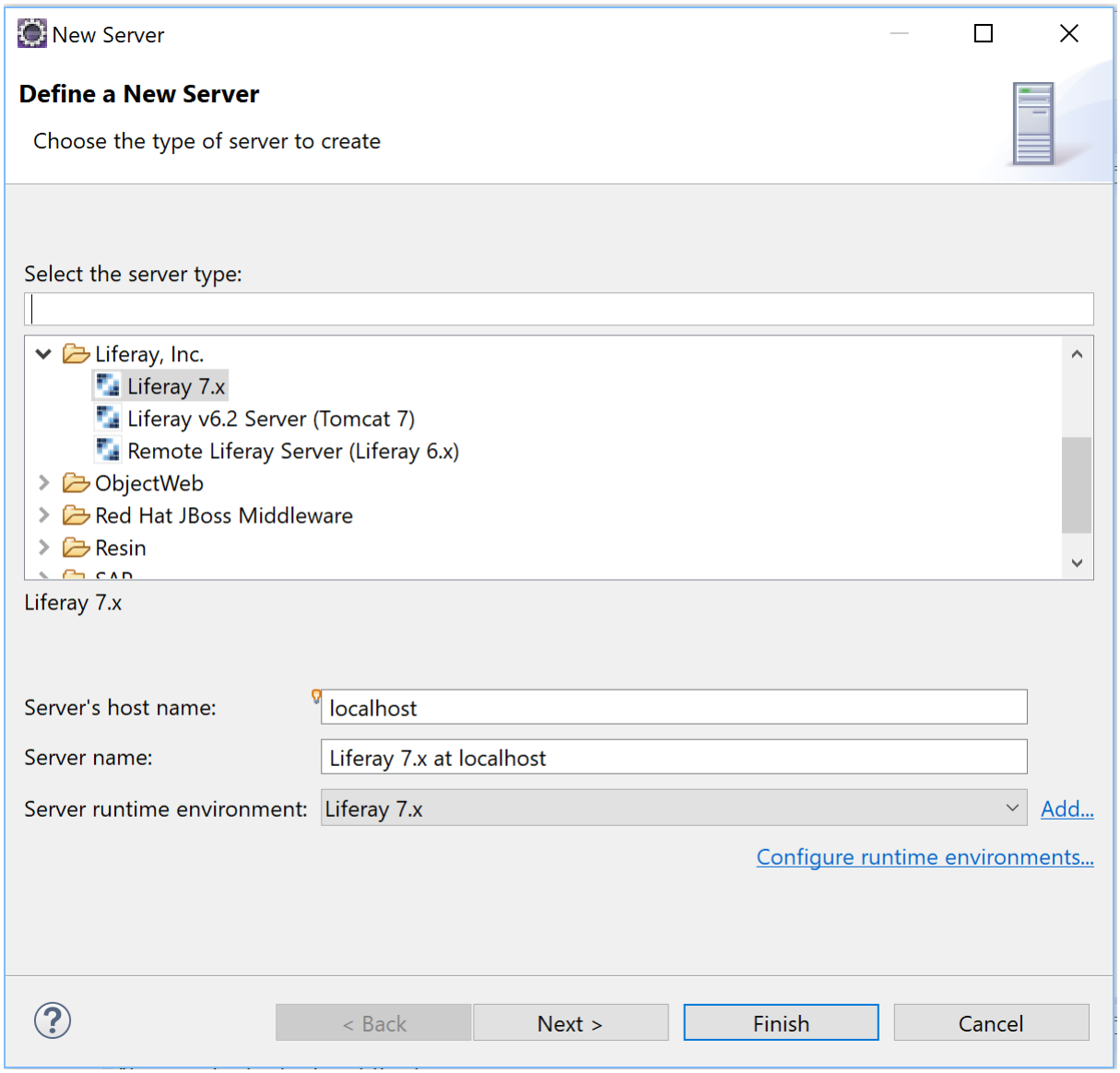


Figure 31.25: Choose the type of server you want to create.

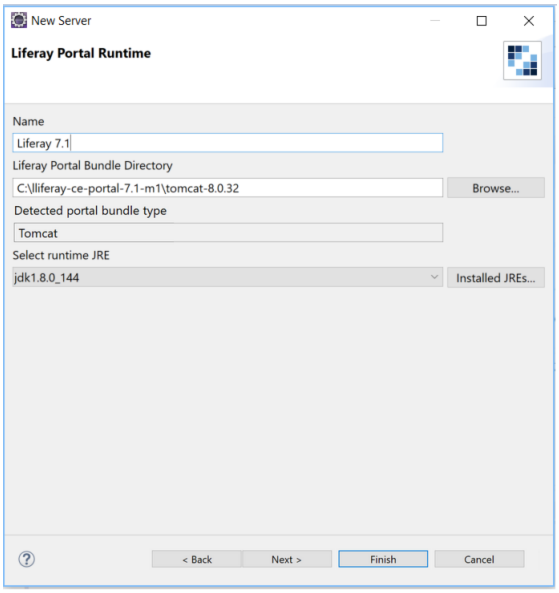


Figure 31.26: Specify the installation folder of the bundle.

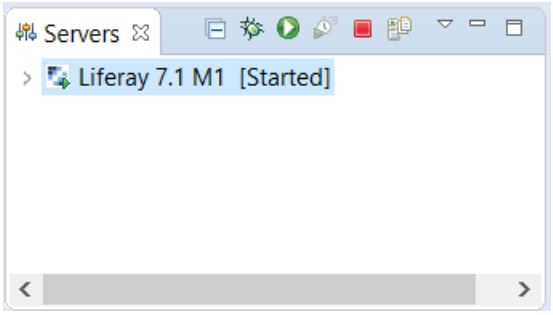


Figure 31.27: Your new server appears under the *Servers* view.

31.10 Searching Liferay DXP Source in Liferay Dev Studio

In Liferay Dev Studio, 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 Dev Studio. 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 Dev Studio 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`.

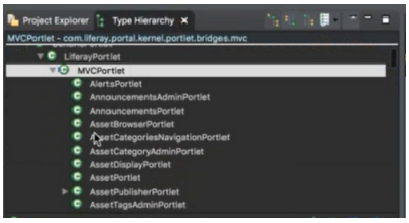


Figure 31.28: Browse the Type Hierarchy window and open the provided classes for examples on how to extend a class.

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 Dev Studio'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*.

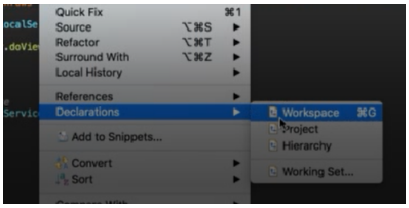


Figure 31.29: All declarations of the method are returned in the Search window.

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*.

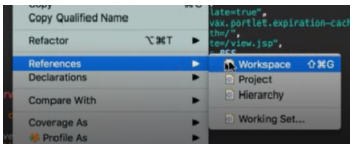


Figure 31.30: All matching annotations are displayed in the Search window.

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!

31.11 Debugging Liferay DXP Source in Liferay Dev Studio

You can use Liferay Dev Studio 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 Dev Studio. See the Managing the Target Platform in 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 Dev Studio. See the Installing a Server in Liferay Dev Studio for more details.
2. Start the server in debug mode. To do this, click the debug button in the Servers pane of Liferay Dev Studio.

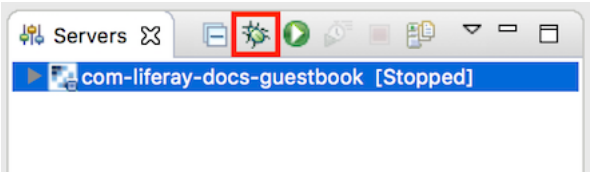


Figure 31.31: The red box in this screenshot highlights the debug button. Click this button to start the server in debug mode.

Awesome! You're now equipped to begin debugging in Liferay Dev Studio!

31.12 Using Gradle in Liferay Dev Studio

Gradle is a popular open source build automation system. You can take full advantage of Gradle in Liferay Dev Studio by utilizing Buildship, which is a collection of Eclipse plugin-ins that provide support for building software using Gradle with Liferay Dev Studio. Buildship is bundled with Liferay Dev Studio versions 3.0 and higher.

The first thing you'll learn about in this tutorial is creating Gradle projects in Dev Studio.

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*.

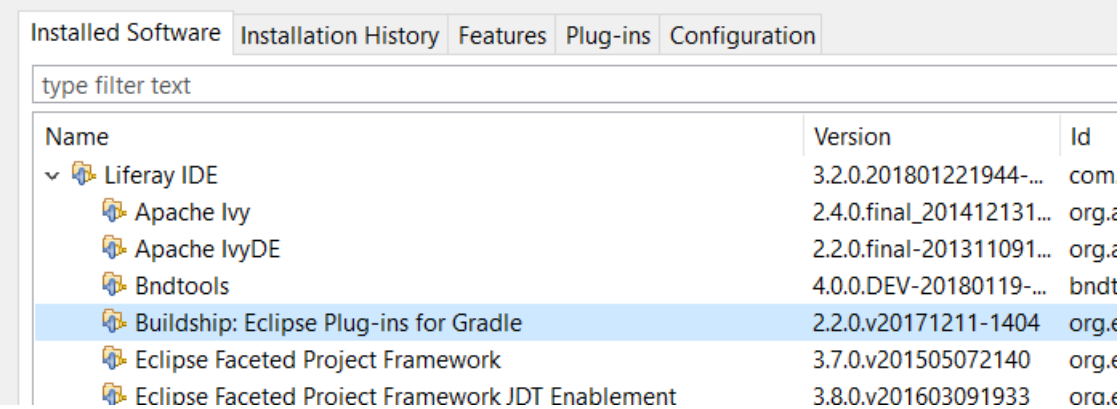


Figure 31.32: Navigate to *Help → Installation Details* to view plugins included in Dev Studio.

2. Enter a valid project name. You can also specify your project location and working sets.

3. Optionally, you can navigate to the next page and specify your Gradle distribution and other advanced options. Once you're finished, select *Finish*.

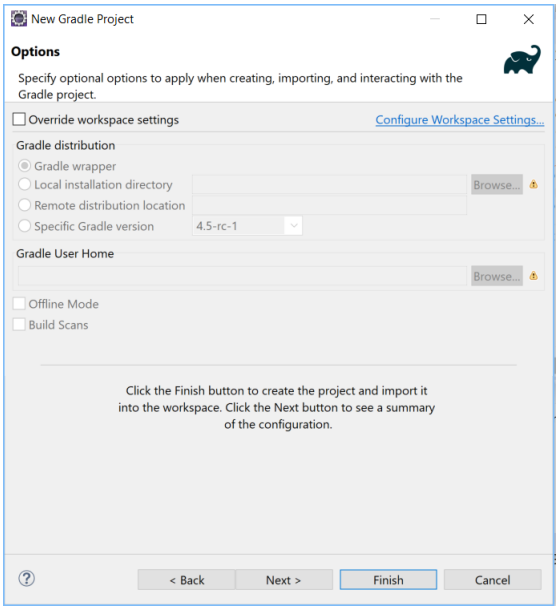


Figure 31.33: You can specify your Gradle distribution and advanced options such as home directories, JVM options, and program arguments.

You can also import existing Gradle projects in Liferay Dev Studio.

1. Go to *File → Import... → Gradle → Existing 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.

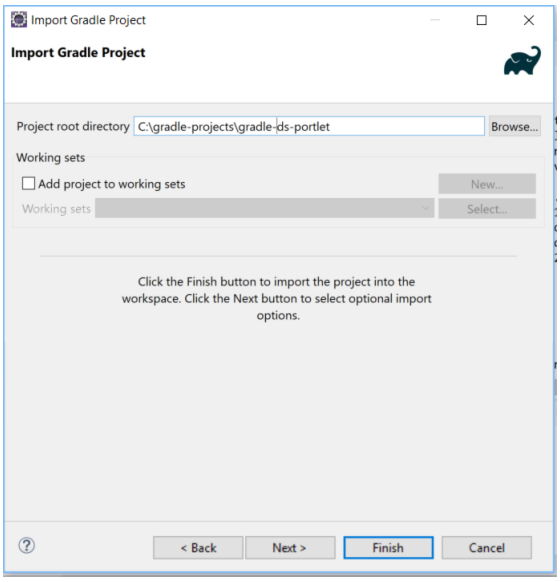


Figure 31.34: You can specify what Gradle project to import from the *Import Gradle Project* wizard.

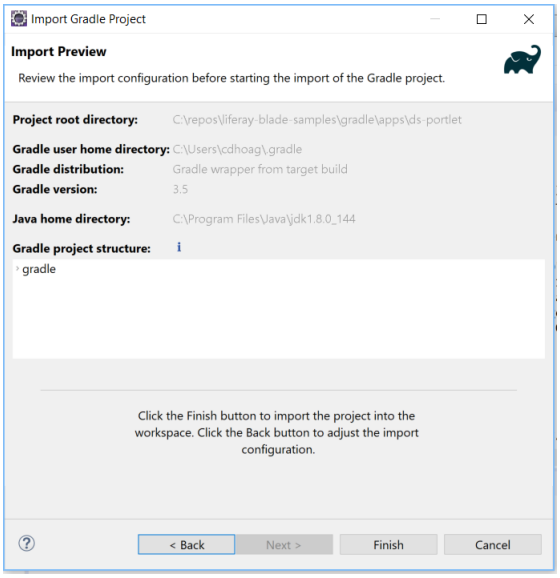


Figure 31.35: You can preview your Gradle project's import information.

Next you'll learn about Gradle tasks and executions, and learn how to run and display them in Liferay Dev Studio.

Gradle Tasks and Executions

Dev Studio 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 lets you 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.

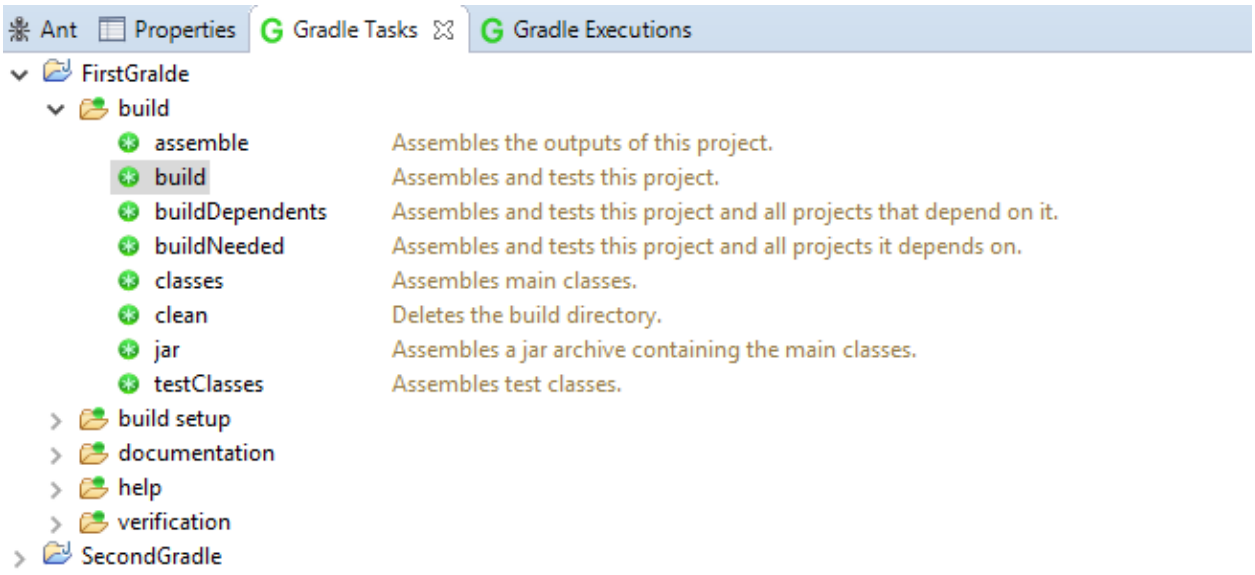


Figure 31.36: Navigate into your preferred Gradle project to view its available Gradle tasks.

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 Dev Studio 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

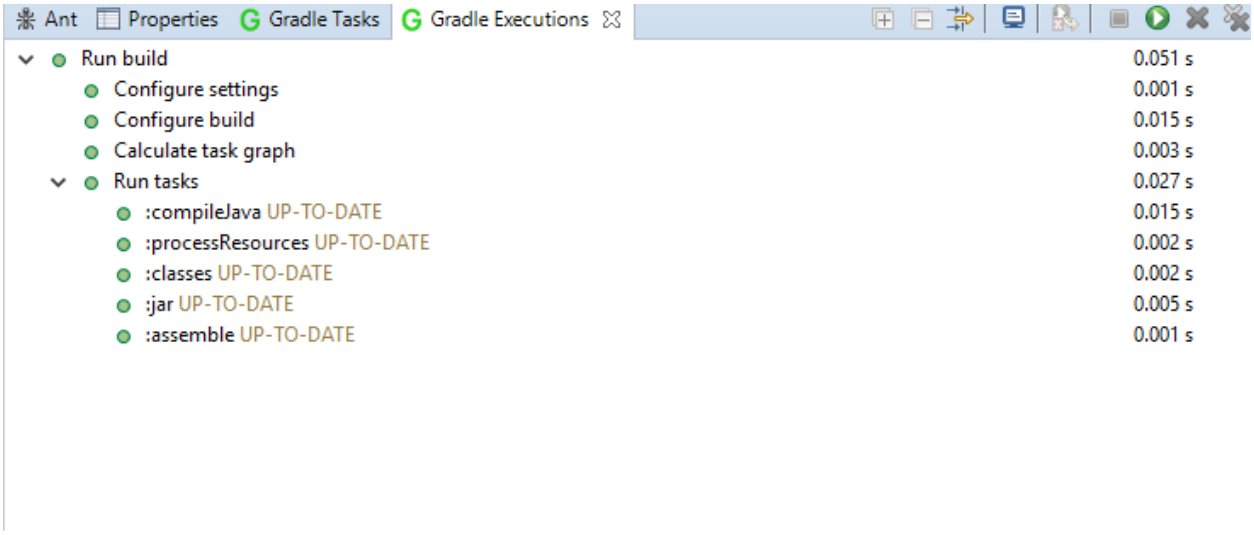


Figure 31.37: The Gradle Executions view helps you visualize the Gradle build process.

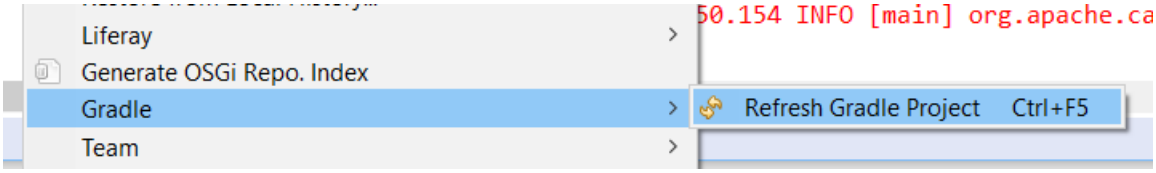


Figure 31.38: Make sure to always refresh your Gradle project in Liferay Dev Studio after build script edits.

automatic synchronization for just one Gradle project, you can right-click a Gradle project and select *Properties → Gradle* and enable auto sync that way.

Excellent! You're now equipped with the knowledge to add, import, and build your Gradle projects in Liferay Dev Studio!

31.13 Using Maven in Liferay Dev Studio

You can take full advantage of Maven in Liferay Dev Studio with its built-in Maven support. In this tutorial, you'll learn about the following topics:

- Installing Maven Plugins for Liferay Dev Studio
- Creating Maven Projects
- Importing Maven Projects
- Using the POM Graphic Editor

First you'll install the necessary Maven plugins for Dev Studio.

Installing Maven Plugins for Liferay Dev Studio

In order to support Maven projects in Dev Studio properly, you first need a mechanism to recognize Maven projects as Dev Studio projects. Dev Studio projects are recognized in Eclipse as faceted

web projects that include the appropriate Liferay plugin facet. Therefore, all Dev Studio projects are also Eclipse web projects (faceted projects with the web facet installed). For Dev Studio 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. Dev Studio relies on the following Eclipse plugins to provide this capability:

- `m2e` (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 Dev Studio, 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 Dev Studio 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/stable/
   ```

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.

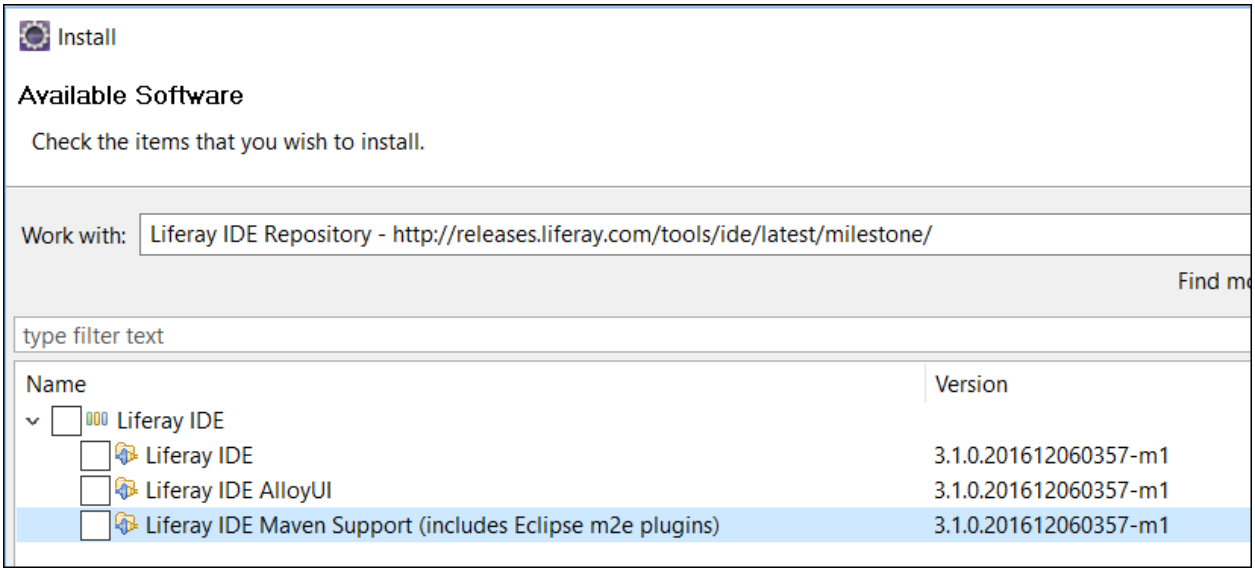


Figure 31.39: You can install all the necessary Maven plugins for Dev Studio by installing the *Liferay IDE Maven Support* option.

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 Dev Studio is ready to develop Maven projects for Liferay DXP!

You'll learn about creating Maven projects in Dev Studio 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` build type, and choose the project template (archetype) to use.

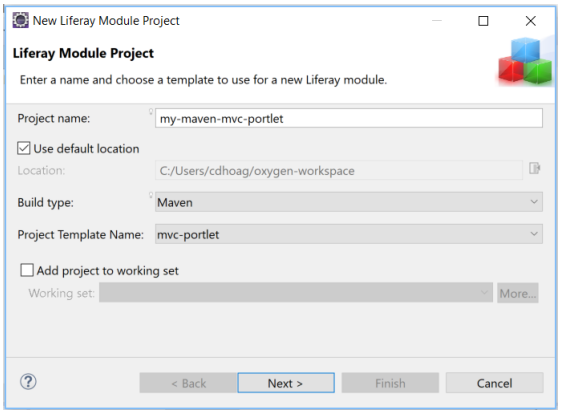


Figure 31.40: The New Liferay Module Project wizard lets you generate a Maven project.

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 Dev Studio with another tool, you can still manage that project in Dev Studio, 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 Dev Studio, follow the steps outlined below:

1. Navigate to *File → Import → Maven → Existing Maven Projects* and click *Next*.

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 Dev Studio's POM graphical editor.

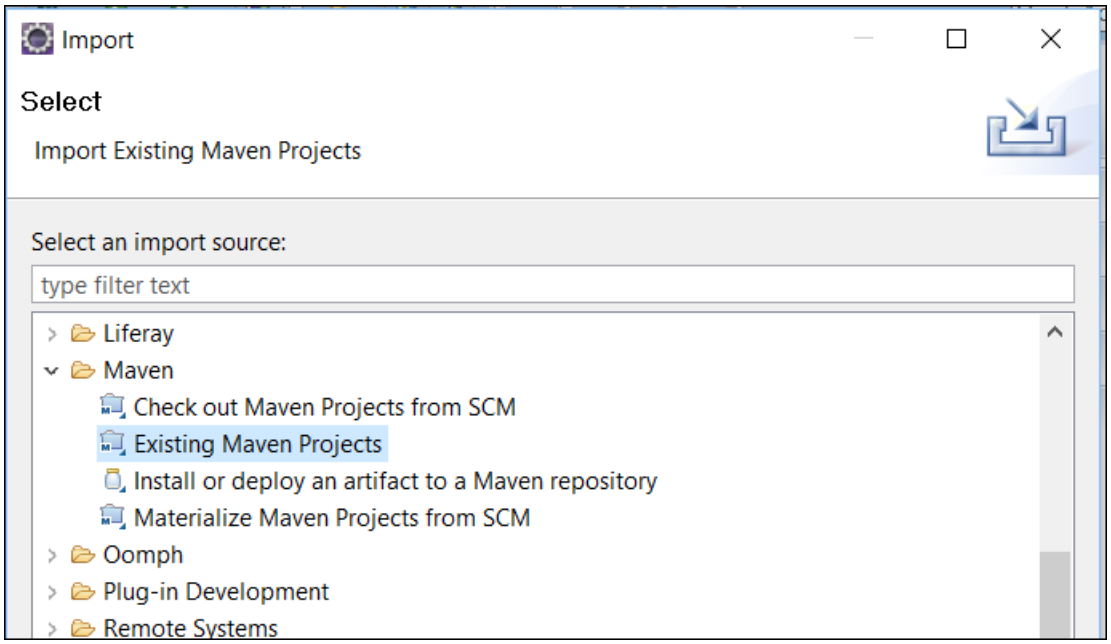


Figure 31.41: Dev Studio offers the Maven folder in the Import wizard.

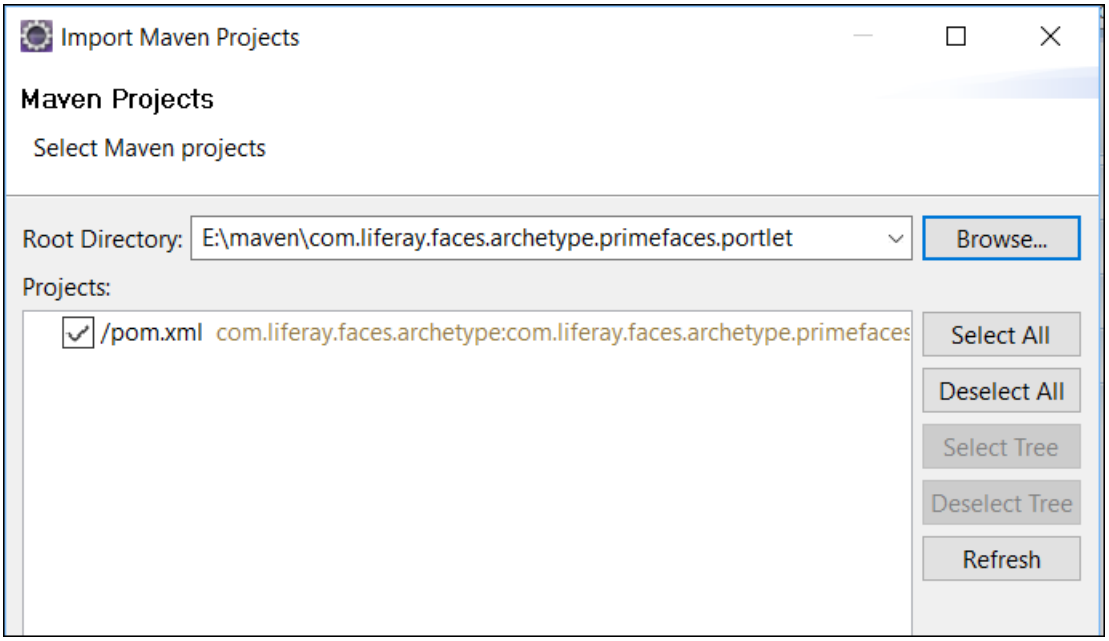


Figure 31.42: Use the Import Maven Projects wizard to import your pre-existing project.

Using the POM Graphic Editor

You're provided a nifty POM graphic editor when opening your Maven project's `pom.xml` in Dev Studio. 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.
- **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.

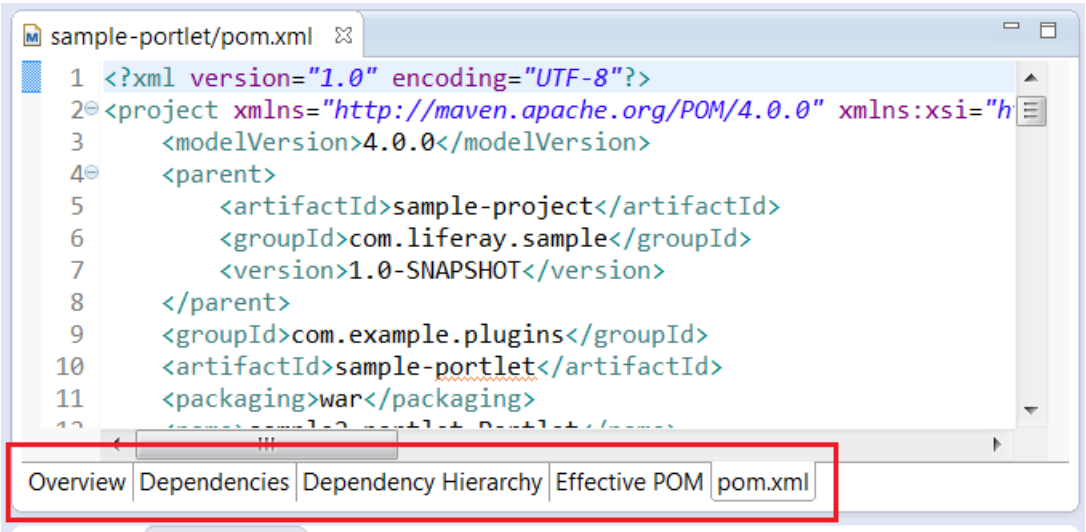


Figure 31.43: Liferay Dev Studio provides five interactive modes to help you edit and organize your POM..

By taking advantage of these interactive modes, Dev Studio makes modifying and organizing your POM and its dependencies a snap!

31.14 Enabling Code Assist Features in Your Project

Liferay Dev Studio'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 Dev Studio instance has the necessary development tooling and Tern integration installed. To do this, go to *Help → About Eclipse → 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.

1. Navigate to *Help → Install New Software...*.

2. Paste the following link into the *Work with* field:

```
http://releases.liferay.com/tools/ide/latest/stable/
```

3. Make sure the *Liferay IDE AlloyUI* option is checked and finish the installation process.

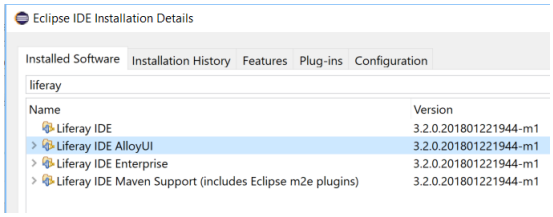


Figure 31.44: The *Liferay IDE AlloyUI* option is actually a sub-option listed within the *Liferay IDE* option.

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 WAR-style portlets generated using the Plugins SDK. For all other project types, (e.g., Liferay module projects), 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. You're presented a menu listing Tern plugins that are installed. For example, 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.

   If you need to refer back to this list of installed Tern plugins, right-click your project and select *Properties*. Then select *Tern → Modules*.

3. Check any additional modules you wish to use in your project and click *OK*.

Your project is now set up to use Dev Studio's Tern features.

Related Topics

Using Front-End Code Assist Features in Dev Studio
Creating Modules with Liferay Dev Studio
Blade CLI

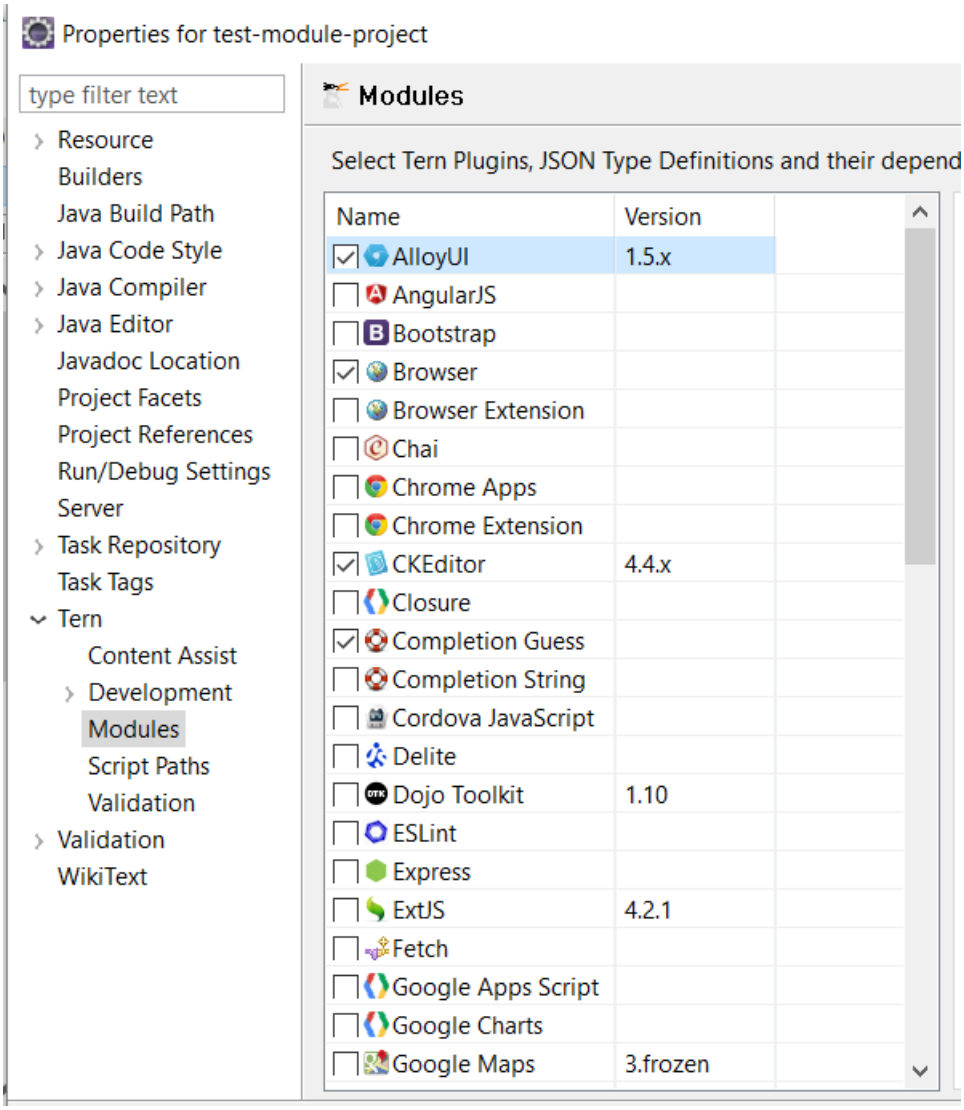


Figure 31.45: By selecting these Tern modules, you can use AlloyUI code assist features in your project.

31.15 Using Front-End Code Assist Features in Dev Studio

Liferay Dev Studio 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 in Dev Studio for

- AlloyUI
- JavaScript
- CSS
- jQuery

Each language is covered in its own section, so you can navigate to the language you're most interested in. There are many languages, including the four listed above, that Dev Studio provides

code assist for. This is provided by Dev Studio's integration of Tern. To access these features, you must be working in a file those languages are expected for (e.g., JavaScript, JSP, HTML, CSS, etc.).

You must have Tern features enabled in your project in order to use them. You should also have the appropriate Tern modules enabled based on the language you're writing in. For example, if you're writing in a jQuery file, you must apply the Tern *jQuery* module to use code assist for that language. See the Enabling Code Assist Features in your Project tutorial to learn how to enable Tern features for your projects.

You'll begin testing the AlloyUI code assist features first.

AlloyUI Code Assist Features

There are several helpful code assist features that can improve your productivity when writing code for AlloyUI. Before beginning, enable the Tern modules required to use AlloyUI features: *AlloyUI*, *Browser*, *JSCS*, *Liferay*, and *YUI Library*. 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.

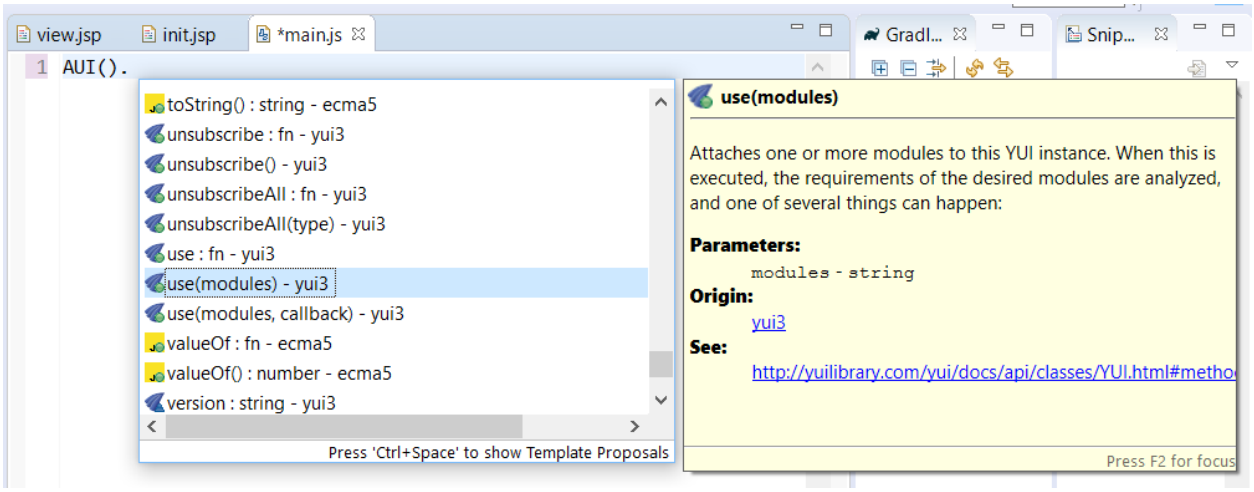


Figure 31.46: This figure demonstrates code inference in a JS file.

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 Dev Studio's Preferences menu. Follow the steps below to enable auto activation:

1. Navigate to *Window → Preferences → JavaScript → Editor → Content Assist*.

2. Check the *Enable auto activation* box and click *Apply and Close*.

   The figure below shows how to enable auto activation:

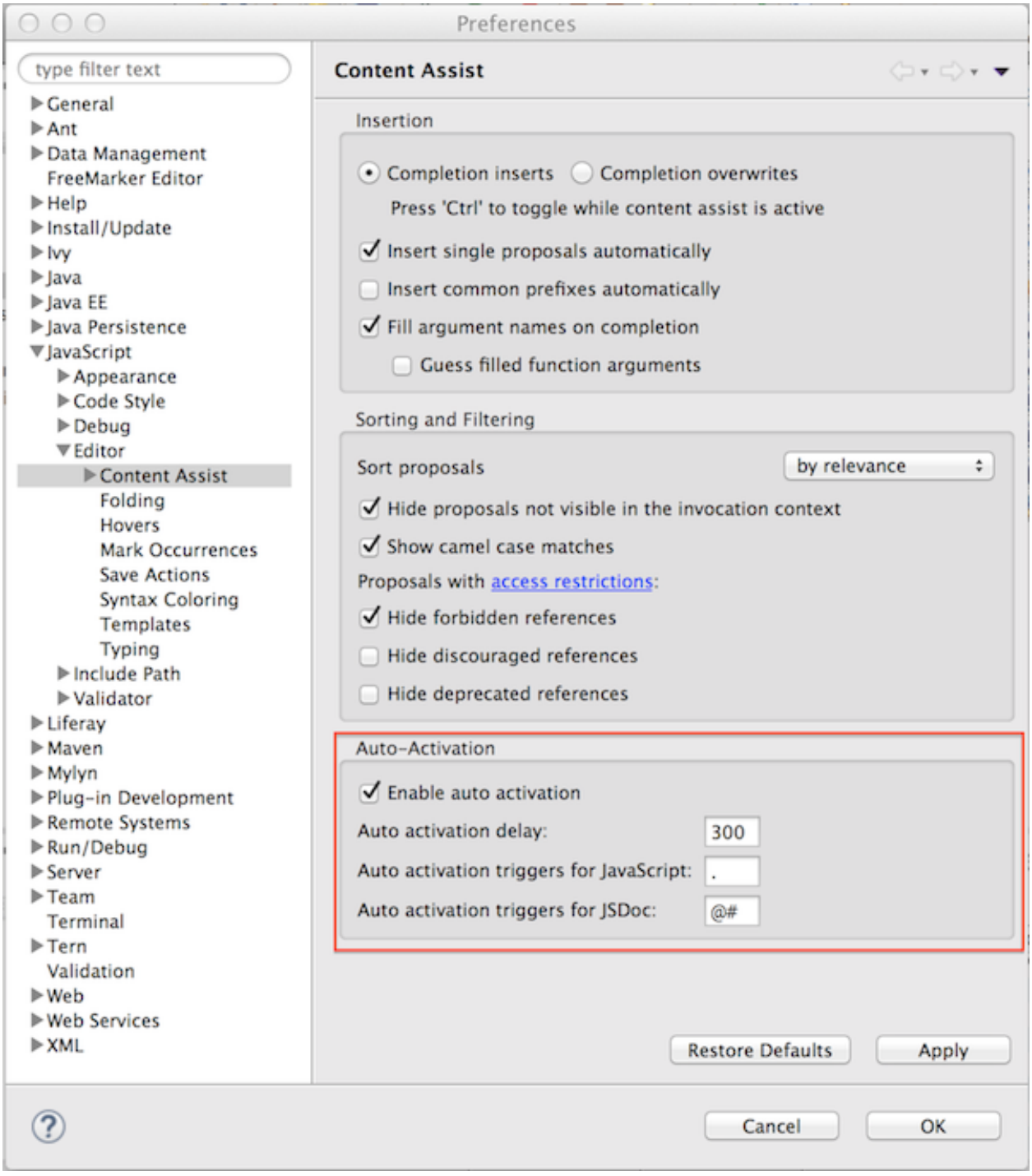


Figure 31.47: The *Enable auto activation* checkbox is listed below the *Auto-Activation* heading.

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.

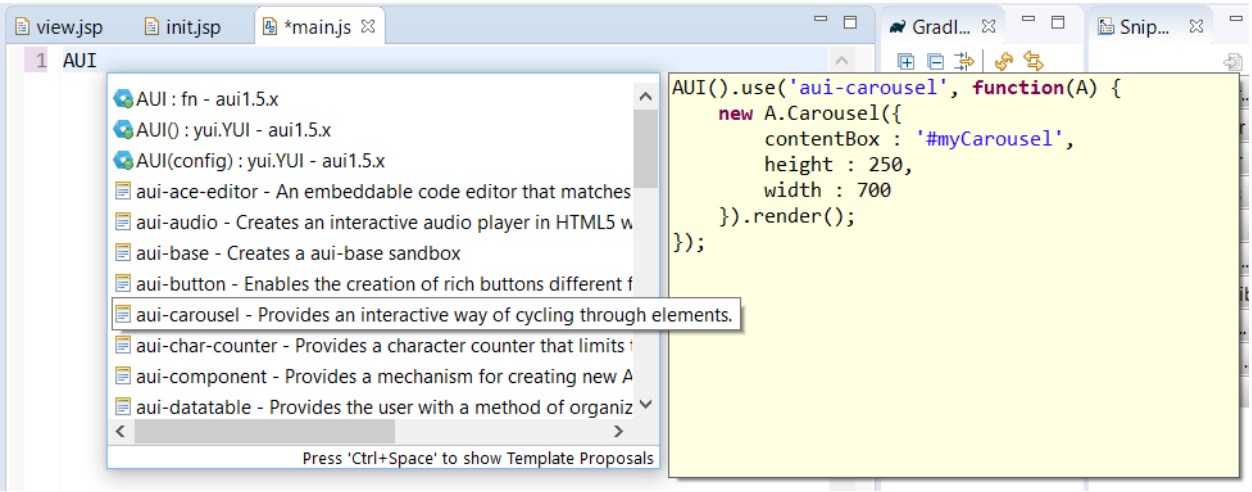


Figure 31.48: Dev Studio gives you access to AUI code templates in the JS and JSP editors.

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 Dev Studio'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's not already in your JSP:

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

You can also add the import from the Snippets menu (*Window → Show View → Snippets*) under *Taglib imports → Liferay AUI Taglib Import v6.0*.

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 Dev Studio.

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.

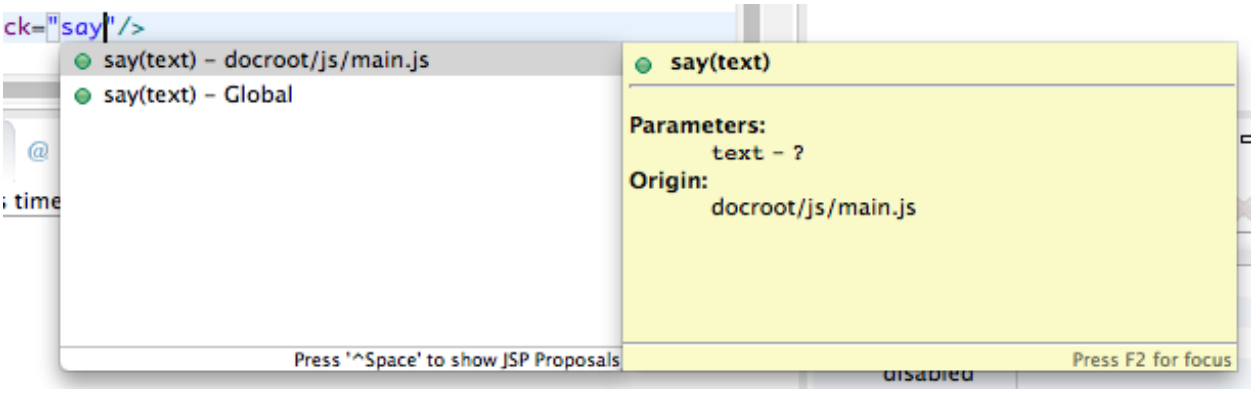


Figure 31.49: JavaScript code assist features give easy access to your functions.

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 Dev Studio'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

Dev Studio 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 Dev Studio 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 Dev Studio. 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 Dev Studio:

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=""/>
```

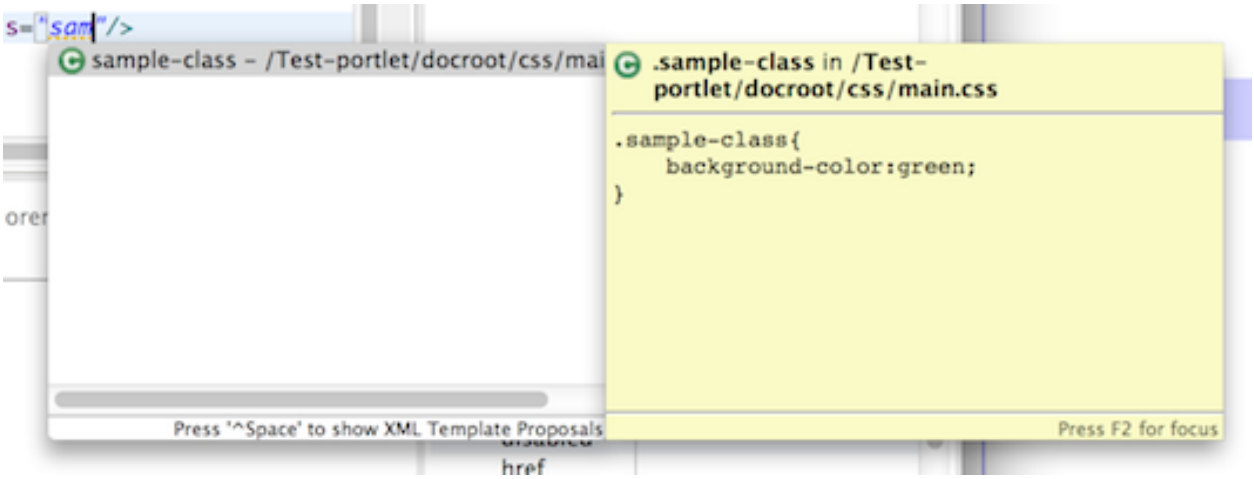


Figure 31.50: CSS code inference improves your workflow when developing in CSS.

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.

   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 =
```

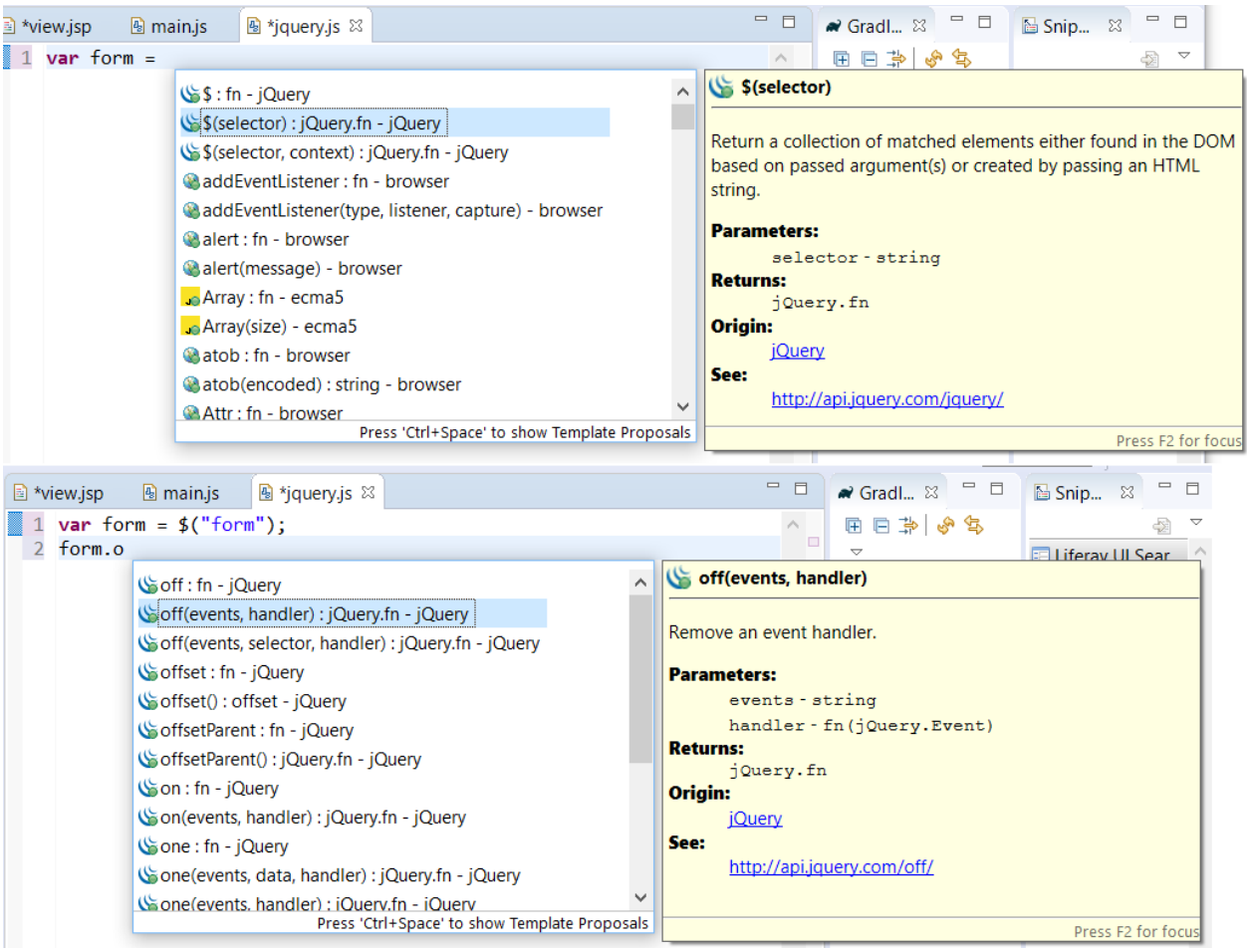


Figure 31.51: Using the jQuery code assist features gives you the convenience of showing you what's available, and the documentation behind each option.

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.

   Furthermore, for jQuery callback handlers, the type information for parameters is also made available.

Excellent! You now know how to use Dev Studio's front-end development code assist features to improve your workflow.

Related Topics

Enabling Code Assist Features in your Project
    Liferay Workspace
    From Liferay 6 to 7.0

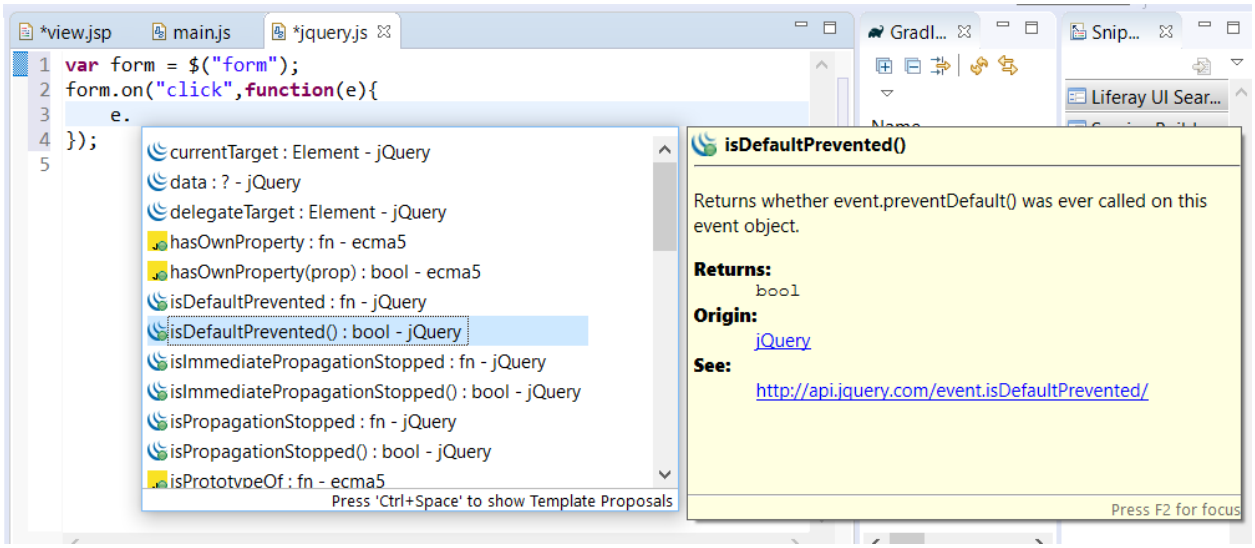


Figure 31.52: jQuery code assist also displays type information for parameters.

CHAPTER 32

BLADE CLI

The Blade CLI tool is the easiest way for Liferay developers to create new Liferay modules. Blade CLI lets you

- create projects (Gradle or Maven) that can be used with any IDE or development environment
- create/manage Liferay DXP instances
- deploy modules (Gradle or Maven)

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 Dev Studio DXP.
- *samples*: Generates a sample project.
- *server*: Starts or stops server defined by your Liferay project.
- *sh*: Connects to Liferay DXP, executes succeeding 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.

For additional information on these sub-commands, run the sub-command with the `--help` flag (e.g., `blade samples --help`).

In this set of tutorials, you'll learn how to use these commands to create and test Liferay DXP instances and modules.

32.1 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 Dev Studio DXP, 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.

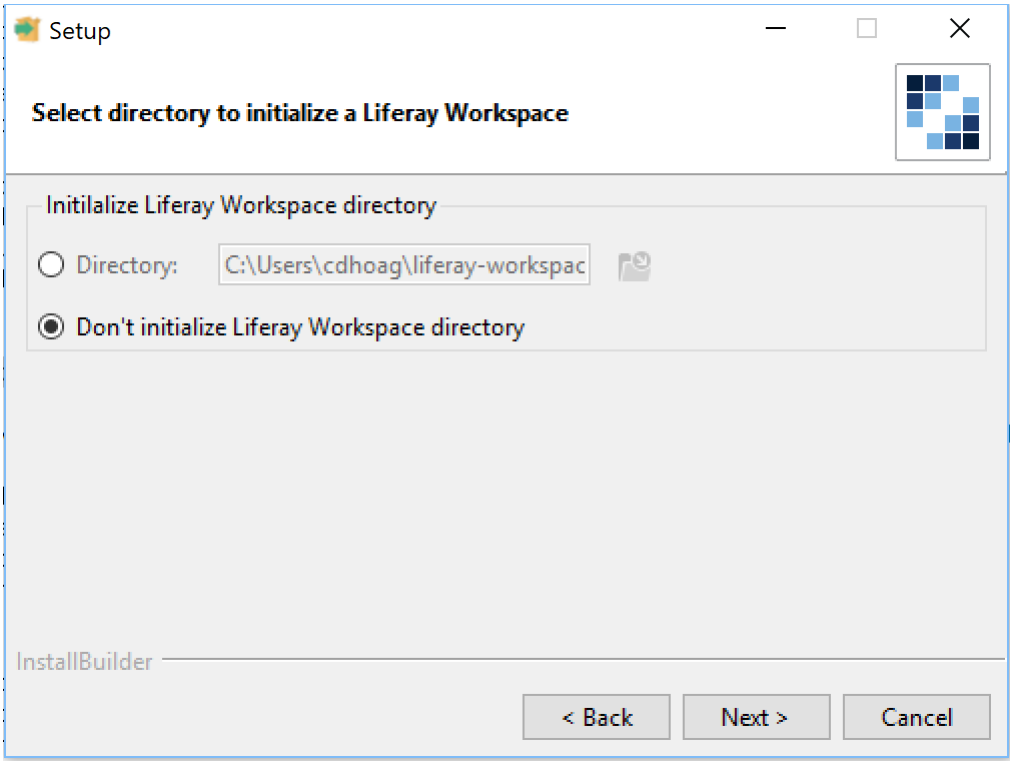


Figure 32.1: Determine where your Liferay Workspace should reside, if you want one.

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*.

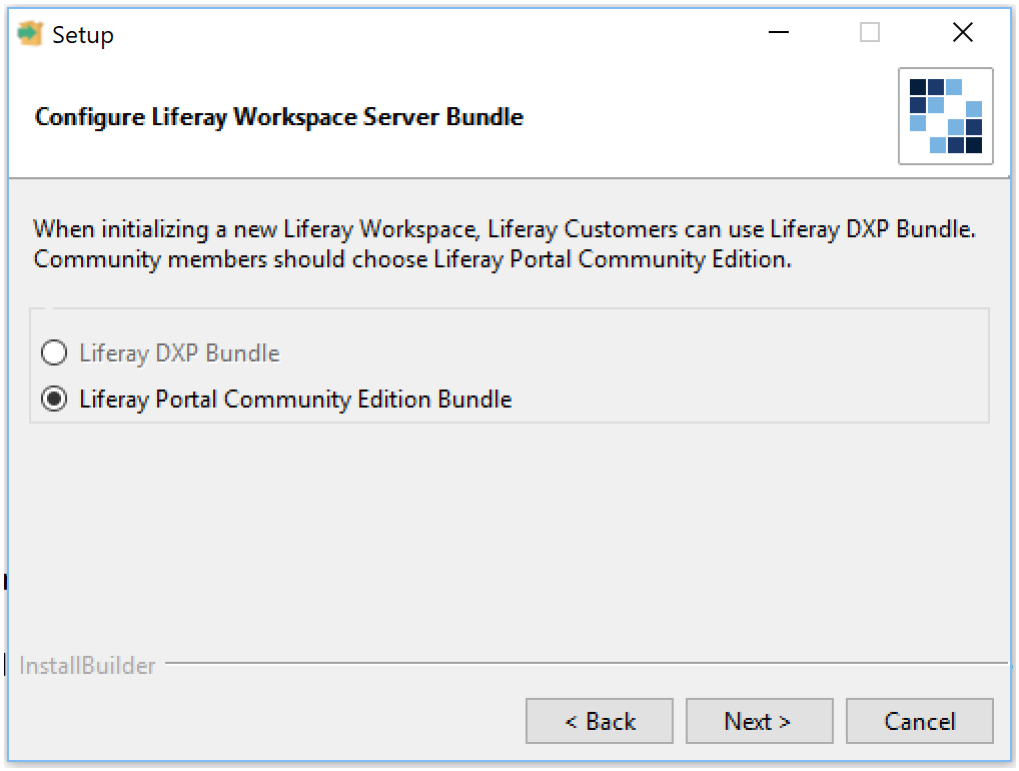


Figure 32.2: Select the product version you'll use with your Liferay Workspace.

5. Click *Next* to begin installing Blade CLI/Liferay Workspace on your computer.

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.

Note: The Liferay Project SDK installer attempts to add JPM to your path. For Windows, it uses the Windows registry. For Mac/Linux, it updates `.bashrc` or `.zshrc`.

At a minimum, Mac/Linux users must open a new shell after the installer finishes for the new features to be available. If, however, you're using a different shell (i.e., Korn, csh, etc.) or you've customized your CLI via `.profile` or some other configuration file, you must add JPM to your path manually.

Blade CLI offers many 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/Linux

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.

32.2 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 the Liferay Project SDK 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.

32.3 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. You can use Blade CLI to generate a Gradle or Maven based workspace. For more information on managing 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 to build a Gradle based workspace:

```
blade init -v 7.1 [WORKSPACE_NAME]
```

To create a Maven based workspace, run this instead:

```
blade init -v 7.1 -b maven [WORKSPACE_NAME]
```

Note: The version you set when first initializing your workspace is stored in the workspace's `.blade.properties` file with the `liferay.version.default` property. This version is applied when creating projects based on the corresponding project template versions.

If you wish to develop projects for a different Liferay DXP version, you can pass a different version in the Blade init command. For example,

```
blade init -v 7.0 [WORKSPACE_NAME]
```

Initializing a workspace requires no downloading or access to the internet.

If you have a Plugins SDK and are looking to migrate to Liferay Workspace using Blade CLI, 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 Configuring a Plugins SDK in Your Workspace section for more details on the `init -u` command. 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`
- `.blade.properties`
- `build.gradle`
- `gradle-local.properties`
- `gradle.properties`
- `gradlew`
- `gradlew.bat`
- `platform.bndrun`
- `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 Configuring a 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 Configuring a 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 Blade CLI. To start your Liferay instance, run

```
blade server start
```

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.

32.4 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:

`--base`: The base directory. The default base directory is the current directory.

`-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, --contributor-type <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, --host-bundle-bsn <String>`: If creating a new JSP hook fragment, provide the name of the host bundle symbolic name. This is required when using the `fragment` project template.

`-H, --host-bundle-version <String>`: If a new JSP hook fragment needs to be created, provide the name of the host bundle version. This is required when using the `fragment` project template.

`-v, --liferay-version`: The version to target when creating a project. The default is `7.2`.

`-l, --list-templates`: Prints a list of available project templates.

`-p, --package-name <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. `--trace`: Prints exception stack traces when they occur. This is `false` by default.

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.

If you want to generate a project for a previous version (e.g., Liferay Portal 7.0), you can specify this using the `-v` flag. For example, to create a project for Liferay Portal 7.0, you would include `-v 7.0` in your `create` command sequence.

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 to 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 must 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. 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
```

Note: Interested in generating legacy versions of Blade samples? Pass in the `-v` param followed by the Liferay DXP version to target. For example,

```
blade samples -v 7.0 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.

32.5 Deploying Projects with Blade CLI

Deploying projects to a Liferay server using Blade CLI is easy. To use the Blade `deploy` command, you must first have built a project to deploy. See the Creating Projects with Blade CLI tutorials for more information about creating Liferay projects. Once you've built a project, 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-7.1.1-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 Project 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.

The `watch` task does not work with JSF portlets or fragment projects.

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.

32.6 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 Dev Studio DXP 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
```

This starts the server in the background. You can tail the logs by adding the `-t` flag. If you prefer starting the server in the foreground, run `blade server run`. Additionally, if you prefer starting the server in debug mode, add the `-d` flag.

Debug mode can be customized by adding the `-p` tag to set the custom remote debugging port (defaults are `8000` for Tomcat and `8787` for Wildfly) and/or the boolean `-s` tag to set whether you want to suspend the started server until the debugger is connected.

Once you've started your server, you can examine its 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.

```
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 32.3: Blade CLI accesses the Gogo shell script to run the `lb` command.

You can run any Gogo command using `blade sh`. This command requires Developer Mode to be enabled. Developer Mode is enabled in Liferay Workspace by default. See the Using the Felix Gogo Shell section for more information on this tool.

To turn off your server, run

```
blade server stop
```

Awesome! You learned how to conveniently interact with Liferay DXP using Blade CLI.

32.7 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
```

Note: For Windows users on Blade CLI 3.3.0 and older, the `blade update` command does not work because Windows cannot update a file that is currently in use. To bypass this issue, use JPM to update your version of Blade CLI:

```
jpm install -f https://releases.liferay.com/tools/blade-cli/latest/blade.jar
```

The `blade update` command for Windows users on Blade CLI 3.4.1+ works as expected.

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 on Nexus by inspecting the com.liferay.blade.cli. artifact. You can check your current installed version by running `blade version`. When running `blade version`, you are notified if there's a newer Blade CLI version available.

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 if you installed an earlier version of Blade CLI, and then used the Liferay Project SDK 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.

32.8 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/7-1/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/7-1/-/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 7.1](/docs/7-1/tutorials/-/knowledge_base/t/upgrading-6-2-themes-to-7-1)
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 33

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. 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 Project SDK installer and run it to install Blade CLI (default CLI for workspace), initialize a new Liferay Workspace, and download Dev Studio DXP.

You can also use it with other developer IDEs. For example, a Liferay Workspace easily integrates with Liferay Dev Studio DXP, providing a seamless development experience. To learn more about Liferay Dev Studio DXP and using workspace with it, see the Creating a Liferay Workspace with Liferay Dev Studio DXP 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.

33.1 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 Dev Studio DXP, 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.

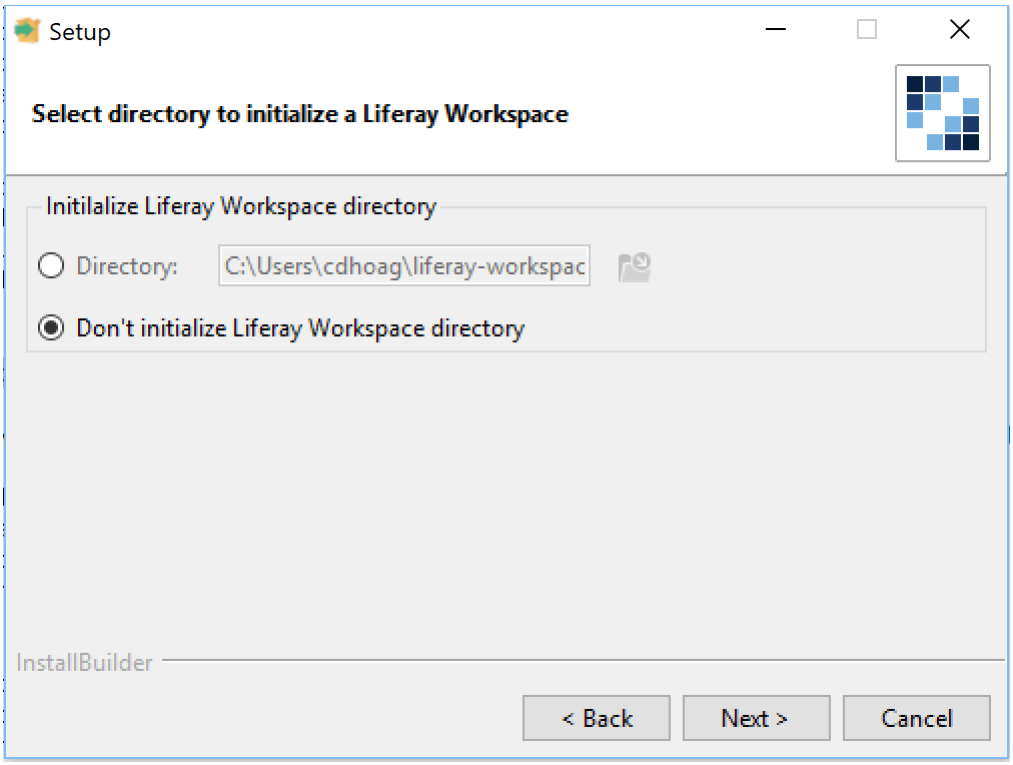


Figure 33.1: Determine where your Liferay Workspace should reside.

   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/7-1/tutorials/-/knowledge_base/t/configuring-a-liferay-workspace#adding-a-
liferay-bundle-to-a-workspace)
 for more information on this topic.
```

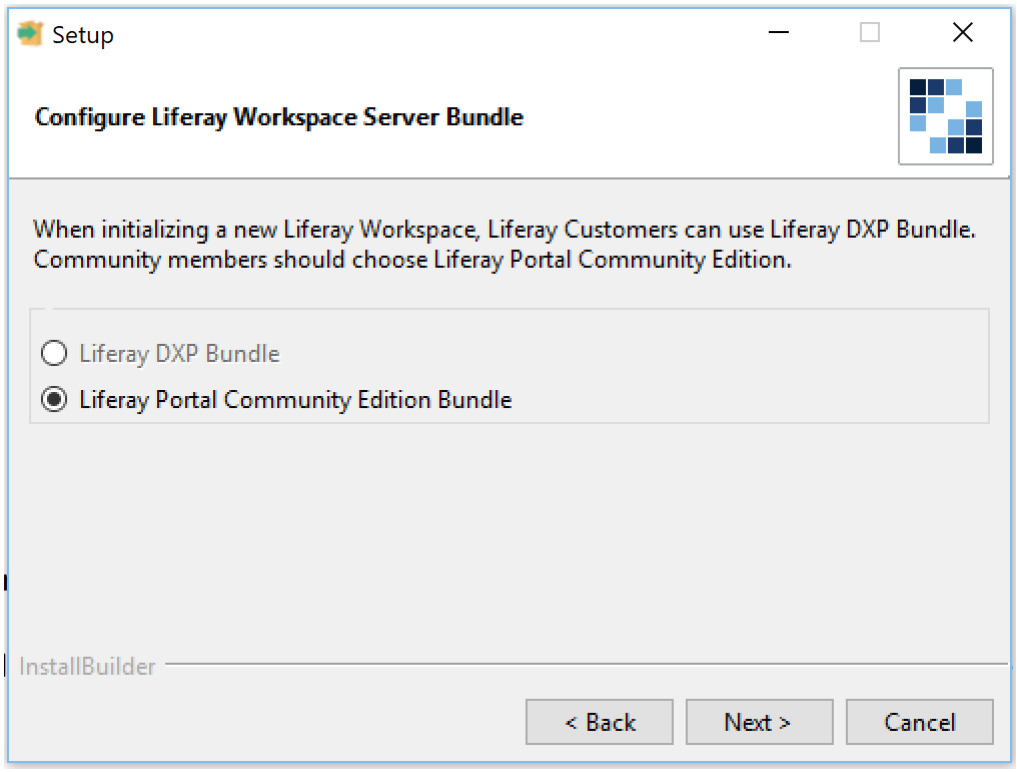


Figure 33.2: Select the product version you'll use with your Liferay Workspace.

5. Click *Next* to begin installing Liferay Workspace on your machine.

That's it! Liferay Workspace is now installed on your machine!

33.2 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 (Gradle) 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 DXP servers and projects residing in your workspace. To learn more about using the `configs` folder, see the Testing Modules section.
- `ext` (generated): holds the Ext OSGi modules and Ext plugins.
- `gradle`: holds the Gradle Wrapper used by your workspace.
- `modules`: holds your custom modules. This can also hold frontend portlets created with the Liferay JS Toolkit
- `plugins-sdk` (generated): holds plugins to migrate from previous releases. These project types should eventually be migrated to the `wars` folder, if possible. This is targeted for Liferay DXP 7.0 to provide a way to migrate from the Plugins SDK to Liferay Workspace. See the Using

a Plugins SDK from Your Workspace section for more information. The Plugins SDK is no longer available for 7.0.

- `themes`: holds Node.js-style themes that use the Liferay JS Theme Toolkit, which are built using the Liferay Theme Generator.
- `wars`: holds traditional WAR-style web application projects and theme projects (i.e., generated by the `theme` project template).
- `build.gradle`: the common Gradle build file.
- `gradle.properties`: specifies the workspace's project locations and Liferay DXP server configuration globally.
- `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.
- `gradlew`: executes the Gradle command wrapper.
- `settings.gradle`: applies plugins to the workspace and configures its dependencies.

If you're using a Liferay Maven Workspace instead, your folder hierarchy is the same, except the Gradle build files are swapped out for a `pom.xml` file. See the Maven Workspace tutorial for more info on configuring that kind of workspace project.

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 many 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.

All properties in the `gradle.properties` file are documented within the file. To learn more about how each property works and what's available, visit your workspace's generated `gradle.properties` file. You can also visit the Gradle Workspace Properties section for a list of these properties.

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/7.1.0-ga1/liferay-ce-portal-tomcat-`

7.1.0-ga1-20180703012531655.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/7.1.10/liferay-dxp-tomcat-7.1.10-ga1-20180703090613030.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!

Gradle Workspace Properties

The following configurable properties are available in your workspace's gradle.properties file:

- liferay.workspace.bundle.cache.dir: Set the directory where the downloaded bundle Zip files are stored. The default value is the .liferay/bundles folder inside the user home directory.
- liferay.workspace.bundle.token.download: Set this to true if the liferay.workspace.bundle.url property is set to a DXP bundle Zip. This property allows the token residing in the ~/.liferay folder to be used to validate your user credentials when downloading the bundle. The default value is false.
- liferay.workspace.bundle.token.email.address: Set the email address to use when downloading a DXP bundle. This is used to create the authentication token. The email address must match the one registered for your DXP subscription. If you wish to create a new token without providing your email address and password in this file, you can create a token manually by navigating to your Liferay profile's Account Setting page and generating a token in the Authentication Tokens menu. Your token must reside in the ~/.liferay folder.

- `liferay.workspace.bundle.token.force`: Set this to `true` to override the existing token with a newly generated token created by the `createToken` task. The default value is `false`.
- `liferay.workspace.bundle.token.password`: Set the password to use when downloading a DXP bundle. This is used to create the authentication token. The password must match the one registered for your DXP subscription. See the `liferay.workspace.bundle.token.email.address` property for more details.
- `liferay.workspace.bundle.token.password.file`: Set the file to hold the Liferay bundle authentication token password. The default file value is `~/.liferay/token`.
- `liferay.workspace.bundle.url`: Set the URL pointing to the bundle Zip to download. If the URL points to a DXP bundle (e.g., https://api.liferay.com/...), set the `liferay.workspace.bundle.token.download` property to `true`. The default value is the URL for the latest version of Liferay Portal CE.
- `liferay.workspace.default.repository.enabled`: Set this to `true` to configure Liferay CDN as the default repository in the root project. The default value is `true`.
- `liferay.workspace.environment`: Set the environment with the settings appropriate for current development. The `configs` folder is used to hold different environments in the same workspace. You can organize environment settings and generate an environment installation with those settings. There are five environments: `common`, `dev`, `local`, `prod`, and `uat`. The default value is `local`.
- `liferay.workspace.home.dir`: Set the folder that contains the Liferay bundle downloaded from the `liferay.workspace.bundle.url` property. The default value is `bundles`.
- `liferay.workspace.modules.default.repository.enabled`: Set this to `true` to configure Liferay CDN as the default repository for module/OSGi projects. The default value is `true`.
- `liferay.workspace.ext.dir`: Set the folder that contains all Ext OSGi modules and Ext plugins. The default value is `ext`.
- `liferay.workspace.modules.dir`: Set the folder that contains all module/OSGi projects. The default value is `modules`.
- `liferay.workspace.modules.jsp.precompile.enabled`: Set this to `true` to compile the JSP files in OSGi modules and have them added to the distributable Zip/Tar. The default value is `false`.
- `liferay.workspace.plugins.sdk.dir`: Set the folder that contains the Plugins SDK environment. The default value is `plugins-sdk`.
- `liferay.workspace.target.platform.version`: Set the Liferay Portal or DXP bundle version to develop against. This property enables target platform features such as the OSGi `resolve` task and specialized dependency management. Use `7.1.1` for the latest Liferay CE release and `7.1.10` for the latest DXP release.
- `liferay.workspace.themes.dir`: Set the folder that contains Node.js-style theme projects, which use the Liferay JS Theme Toolkit. The default value is `themes`.
- `liferay.workspace.themes.java.build`: Set this to `true` to build the theme projects using the Liferay Portal Tools Theme Builder. The default value is `false`.
- `liferay.workspace.wars.dir`: Set the folder that contains classic WAR-style projects. The default value is `wars`.

That's it! You now have the knowledge to fully leverage the power of Liferay Workspace!

33.3 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.

33.4 Development Lifecycle for a Liferay Workspace

Liferay Workspaces provide an environment that supports all phases of a Liferay module's development lifecycle:

- Creating projects
- Building projects
- Deploying projects
- Testing projects
- Releasing projects

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 Dev Studio DXP). Let's get started!

Creating Projects

The first step of Liferay Workspace's development phase is the project creation process. Workspace provides a slew of templates that you can use to create many different types of Liferay projects. Workspace also provides development support for frontend portlets generated with the Liferay JS Toolkit. They're stored in the `modules` folder by default.

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.

Workspace also provides a way to create WAR projects, which are generated in the folder set by the `liferay.workspace.wars.dir` property in the `gradle.properties` file. There are several project templates that create WAR-style projects, which should be stored in the `wars` folder.

To learn more about creating projects in a workspace using Blade CLI or Liferay Dev Studio, visit the Creating Projects with Blade CLI and Creating Modules with Liferay Dev Studio tutorials, respectively.

Building Projects

Liferay Workspace abstracts many build requirements away so you can focus on developing projects instead of worrying about how to build them. Liferay Workspace is built using Gradle, so your projects 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-7.1-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 project 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 Projects

Liferay Workspace provides easy-to-use deployment mechanisms that let you deploy your project to a Liferay server without any custom configuration. To learn more about deploying projects from a workspace using Blade CLI or Liferay Dev Studio DXP, visit the Deploying Projects with Blade CLI and Deploying Modules with Liferay Dev Studio DXP tutorials, respectively.

Testing Projects

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.

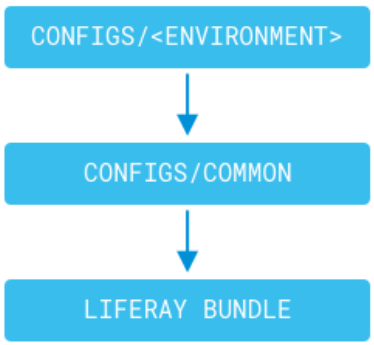


Figure 33.3: The `configs/common` and `configs/[environment]` overlay you Liferay DXP bundle when it's generated.

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]
```

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 Projects

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 projects to a Maven Nexus repository. You could also use other release tools like Artifactory.

Uploading projects to a remote repository is useful if you need to share them with other non-workspace projects. Also, if you're ready for your projects 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 projects, see the Creating a Maven Repository and Deploying Liferay Maven Artifacts to a Repository tutorials.

33.5 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 Dev Studio DXP 3.2+ helps you streamline targeting a specific version even more. Dev Studio DXP 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 Dev Studio DXP 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=7.1.1
   ```

   If you're using Liferay DXP, you can set the property like this:

   ```
   liferay.workspace.target.platform.version=7.1.10
   ```

   The versions following a GA1 release of DXP follow fix pack versions (e.g., `7.1.10.fp1`, `7.1.10.fp2`, 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 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: "7.1.0"
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom.compile.only", version: "7.1.0"
}
```

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`
- `7.1.0` → `7.1.10`

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:
  - `com.liferay.ce.portal.bom`: provides all the artifacts included in Liferay DXP.
  - `com.liferay.ce.portal.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 Dev Studio DXP, 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 Dev Studio DXP.

Now you can define your target platform!

33.6 Managing Themes in Liferay Workspace

Creating a Liferay DXP theme can be accomplished using two different tools:

- Liferay Theme Generator (Node.js-based themes that use the Liferay JS Theme Toolkit)
- Project template/archetype (Gradle/Maven-based)

Liferay Workspace offers an environment where developers can use the Liferay Theme Generator to create themes and their work can be seamlessly integrated into their overall DevOps strategy. You can leverage the Liferay Theme Generator to create Node.js-based themes inside workspace or you can leverage it externally and copy themes into Workspace.

Workspace also offers a traditional Java-based theme approach (leveraging Gradle/Maven) for those that can't use the Liferay JS Theme Toolkit's tools in their CI environment.

Below you'll learn how to manage both Node.js-based themes and Gradle/Maven-based themes in Workspace.

Node.js Themes in Workspace

Liferay Workspace reserves the `themes` folder only for themes that are created with the Themes Generator. There are no Blade CLI-provided commands or Maven archetypes to generate a theme. You must leverage the Liferay Theme Generator from within the `themes` folder to create them; you can also copy a generated theme into the folder.

You'll demo this theme management capability next. Be sure the Liferay Theme Generator's required tooling is installed.

1. Navigate to your workspace's `themes` folder and run the following command:

   ```
   yo liferay-theme
   ```

   Follow the prompts to create your theme.

2. Navigate into your new theme and run `../gradlew build`. Liferay Workspace builds the front-end theme using Gradle. Under the hood, Liferay's Node Gradle Plugin is applied and used to build your theme.

3. Workspace is smart enough to differentiate between theme types. For instance, you can't copy a theme built with the Theme Generator into the `wars` folder and expect it to build. You can test if your project is recognized by Workspace by running this command from Workspace's root folder:

   ```
   ../gradlew projects
   ```

   Your CLI should display your new theme under the `themes` project.

   ```
   Root project 'liferay-workspace'
   +--- Project ':themes'
   ```

```
        \--- Project ':themes:my-generated-theme'
```

```
If you moved a WAR-style theme (Gradle/Maven-based) into the `themes`
folder, it is not recognized by the Gradle `projects` command.

**Note:** Workspace identifies whether a theme was generated by the Theme
Generator by checking whether it has a `package.json` file. Any theme
without this file is not compatible in the `themes` folder.
```

Excellent! You learned how generated themes are recognized in workspace and where they should reside. Next you'll learn how workspace manages WAR-style themes.

Gradle/Maven Themes in Workspace

Liferay Workspace provides the `wars` folder for any WAR-style project. Themes created with Blade CLI or Maven using the `theme` project template or archetype are automatically generated here when creating the project within Workspace.

Themes built using Liferay's `theme` project template are always WARs and should always reside in Workspace's `wars` folder. They should never be moved to the `themes` folder; that folder is reserved for themes generated by the Theme Generator only.

To build an existing WAR-style theme in Workspace, run the `../gradlew build` command. Liferay Workspace builds the theme using Gradle. Under the hood, Liferay's Theme Builder Gradle Plugin is applied and used to build your theme. It works similarly in a Maven workspace. See the Building Themes in a Maven Project tutorial for more information.

Awesome! You know how WAR-style themes are built in workspace and where they should reside.

CHAPTER 34

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.

34.1 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=7.1.0
```

If you're using Liferay DXP, you can set the property like this:

```
liferay.workspace.target.platform.version=7.1.10
```

The versions following a GA1 release of DXP follow fix pack versions (e.g., `7.1.10.fp1`, `7.1.10.fp2`, etc.).

Setting the target platform property 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.

34.2 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 7.1.0
   ```

   Liferay DXP users must replace the `release.portal.distro` artifact name with `release.dxp.distro` and use the `7.1.10` 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.

34.3 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.

34.4 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: "7.1.0"
    targetPlatformBoms group: "com.liferay.portal", name: "release.portal.bom.compile.only", version: "7.1.0"
    targetPlatformDistro group: "com.liferay.portal", name "release.portal.distro", version: "7.1.0"
}
```

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`
- `release.portal.distro` → `release.dxp.distro`
- `7.1.0` → `7.1.10`

Now you can validate your modules against a target platform!

34.5 Leveraging Docker

Docker has become increasingly popular in today's development lifecycle, by providing an automated way to package software and its dependencies into a standardized unit that can be shared cross-platform. Read Docker's extensive documentation to learn more.

Liferay provides Docker images for

- Liferay Portal
- Liferay DXP
- Liferay Commerce
- Liferay Portal Snapshots

You can pull Liferay's Docker images from those resources and manage them yourself. Liferay Workspace, however, provides an easy way to integrate Docker development into your existing development workflow with preconfigured Gradle tasks.

Note: Leveraging Docker in Liferay Workspace is only available for Gradle projects at this time.

In this tutorial, you'll learn how to do the following tasks within a workspace:

- Creating a Docker container based on a provided Liferay DXP image
- Configuring the container
- Interacting with the container
- Building a custom Liferay DXP image

Creating a Liferay DXP Docker Container

1. Choose the Docker image you need. This is configured in your workspace's `gradle.properties` file by customizing this property:

   ```
   liferay.workspace.docker.image.liferay
   ```

   To find the possible property values you can set, see the official Liferay DXP Docker Hub's Tags section (e.g., Liferay Portal Docker Tags. For example, if you want to base your container on the Liferay Portal 7.1 GA2 image, you would set this property:

   ```
   liferay.workspace.docker.image.liferay=liferay/portal:7.1.1-ga2
   ```

2. Run the following command from your workspace's root folder:

   ```
   ./gradlew createDockerContainer
   ```

This command creates a new container named `[projectName]-liferayapp`. A new `build/docker` folder is generated in your workspace. This folder is mounted into the container's file system. This means files in workspace's `build/docker` folder are also available in the container's `/etc/liferay` folder.

Any projects in your workspace are automatically compiled and copied to the `build/docker/deploy` folder when the container is created; this means that when the container is started, all your projects are deployed to the container. All configurations are also applied to the container. You'll learn more about configuring your container next.

Configuring the Container

Before starting your container, you may want to add additional portal configurations. This could include things like

- Property overrides (e.g., `portal-ext.properties`)
- Marketplace app overrides
- App server configurations
- License files

You can do this by applying files (and their accompanying folder structures, if necessary) to your workspace's `configs/docker` folder. This folder is treated as your Liferay Home for Docker development; you add additional files that overlay your workspace's `configs/common` folder and your Liferay DXP container's default configuration.

For example, to enable the Gogo shell for your container, add a `configs/docker/portal-ext.properties` file to your workspace with the following configuration:

```
module.framework.properties.osgi.console=0.0.0.0:11311
```

This lets you access your container using Gogo shell via telnet session.

Once the container is started, the configurations stored in `configs/commmon` and `configs/docker` are transferred to the `build/docker/files` folder, which applies all configurations to the container's file system. For more information on workspace's `configs` folder, see this section.

Note: You can call the `deployDocker` Gradle task from your workspace's root folder to initiate the Docker configuration transfer to the `build/docker/files` folder manually. It's executed automatically when creating or starting the container.

Next, you'll explore the commands for interacting with the container.

Interacting with the Container

`startDockerContainer`: starts the container.

`logsDockerContainer`: prints the portal runtime's logs. You can exit log tracking mode while maintaining a running container (e.g., [Ctrl|Command] + C).

`dockerDeploy`: deploys the project to the container's `deploy` folder by copying the project archive file to workspace's `build/docker/deploy` folder. This command can also be executed from workspace's root folder to deploy all projects and copy all Docker configurations (i.e., from the `configs/common` and `configs/docker` folders) to the container.

`stopDockerContainer`: stops the container.

`removeDockerContainer`: removes the container from Docker's system.

Note: During your container's startup, you may run into the following error:

```
/etc/liferay/entrypoint.sh: line 3:    11 Killed
${LIFERAY_HOME}/tomcat/bin/catalina.sh run
```

This usually means you have not allocated enough memory to your Docker engine to successfully run your container. See Docker's documentation to learn how to increase resources available to Docker.

Next, you'll learn how to build a custom image.

Building a Custom Liferay DXP Image

You can preserve your container's configuration by building it as an image. To build your custom Liferay DXP image, run

```
./gradlew buildDockerImage
```

A `Dockerfile` is generated for your container when building your image. To do this manually, run

```
./gradlew createDockerfile
```

The `Dockerfile` is generated in your workspace's `build/docker` folder. For more information on how to configure the `Dockerfile`, see Docker's Dockerfile reference documentation.

Your custom image is now available! Run `docker image ls` to verify its availability.

You can now manage Liferay's Docker images in Liferay Workspace!

34.6 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`).

Note: The Gradle wrapper provided in a Gradle-based Liferay Workspace must be updated if you're migrating from a workspace before version `1.10.14` to the latest available version. To update your Gradle wrapper, run

```
./gradlew wrapper --gradle-version=4.10.2
```

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.

34.7 Updating Default Plugins Provided by Liferay Workspace

Liferay Workspace comes with a slew of plugins like these:

- CSS Builder
- Javadoc Formatter
- Lang Builder
- Service Builder
- Source Formatter
- Theme Builder
- etc.

Bundled plugins are updated with each release of workspace. Suppose you need a new feature in the Source Formatter plugin, but the latest workspace version has not yet been updated to include it. You can upgrade it yourself!

To upgrade one of workspace's bundled plugins, follow these steps:

1. Find the bundle symbolic name (BSN) for the plugin you want to update. You can find this value in the `portal-tools.properties` file. For example, the Source Formatter's BSN is `com.liferay.source.formatter`.

2. Open your workspace's `gradle.properties` file and copy the plugin's BSN followed by `.version` and set the desired plugin version you want to use. For example,

   ```
   com.liferay.source.formatter.version=1.0.654
   ```

   If you're most interested in the latest and greatest plugins, you can set the above property to `latest.release` to always use the latest available version. This could, however, cause your workspace to become unstable.

That's it! You're no longer tied to particular plugin versions provided by your workspace.

CHAPTER 35

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 project 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!

35.1 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>2.61.2</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>
```

When the Liferay repository is configured in your `settings.xml` file, archetypes are generated based on that repository's contents. See the Generating New Projects Using Archetypes tutorial for details on using Maven archetypes for Liferay development.

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.

35.2 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 to easily create Liferay projects. In this tutorial, you'll learn how to use Liferay's Maven archetypes to generate a project.

At the time of this writing, Liferay provides approximately 60 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., 7.0, 6.2). For example, archetypes with the `com.liferay.maven.archetypes` prefix are legacy archetypes targeted for Liferay Portal 6.2. 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
- Fragment
- MVC Portlet
- npm Angular Portlet
- npm React Portlet
- Panel App
- Portlet Provider
- Service Builder
- Soy Portlet
- Theme
- and many more...

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., 11).

   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`: `1.0.6`
   - `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 you ran the archetype generation command from. 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 projects, but they're bare bones and likely require further customizations.

If you want to generate a quick foundation for a Liferay project built with Maven, using Liferay Maven archetypes is your best option.

35.3 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 DXP, 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>
    <dependencies>
        <dependency>
            <groupId>biz.aQute.bnd</groupId>
            <artifactId>biz.aQute.bndlib</artifactId>
            <version>3.2.0</version>
        </dependency>
        <dependency>
            <groupId>com.liferay</groupId>
            <artifactId>com.liferay.ant.bnd</artifactId>
            <version>2.0.41</version>
        </dependency>
    </dependencies>
</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/7-1/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 Project Built with Maven to Liferay DXP tutorial.

Fantastic! You've configured your Liferay Maven project to package itself into a deployable OSGi module.

35.4 Deploying a Project Built with Maven to Liferay DXP

There are two ways to deploy a Maven-built Liferay project:

1. Copy your generated Maven JAR/WAR 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 line.

Although manually copying your JAR/WAR 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 project to Liferay DXP with one command execution.

Note: 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.

If you're deploying a module JAR, 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>3.2.1</version>
                <executions>
                    <execution>
                        <id>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-7.1-ga1</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.

35.5 Creating a Maven Repository

You'll frequently want to share Liferay artifacts and modules with teammates or manage your repositories using a GUI. You can do this using Sonatype Nexus. 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 on Nexus' Installation page 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*.

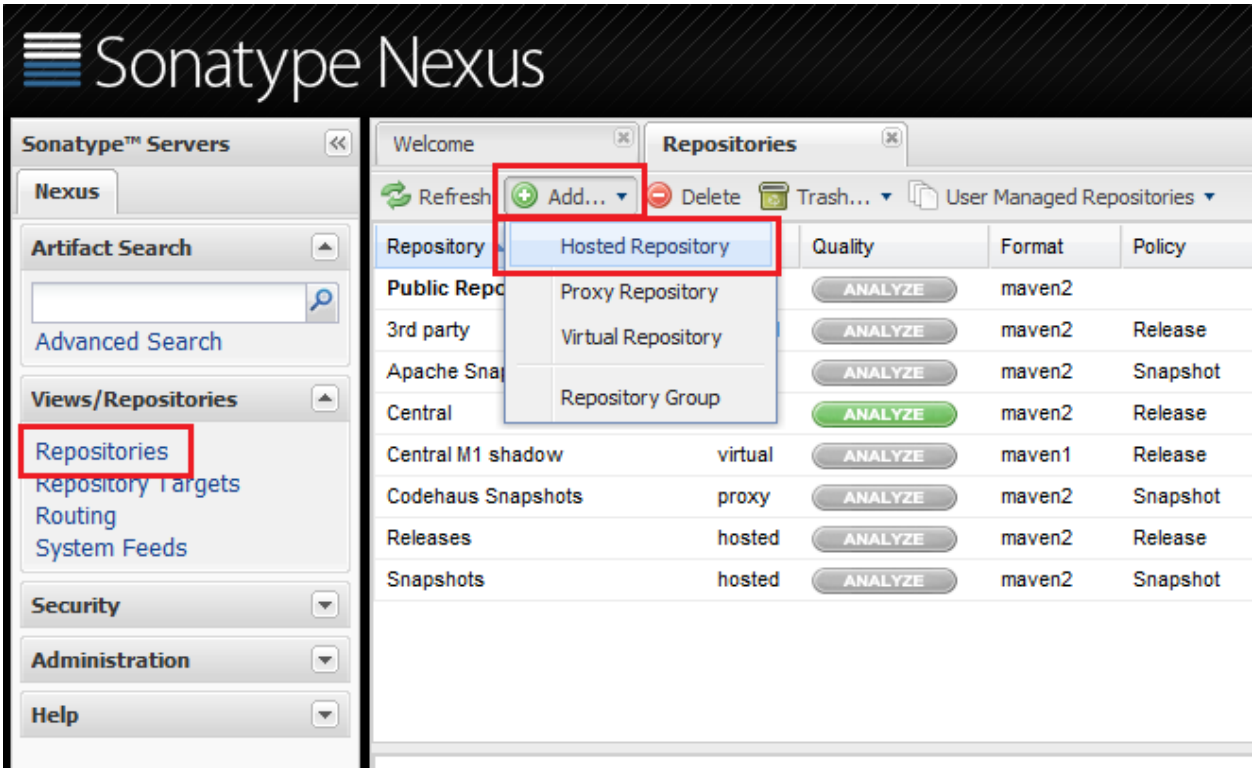


Figure 35.1: Adding a repository to hold your Liferay artifacts is easy with Nexus.

   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*

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 project 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!

35.6 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 project 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`) using the command line 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.

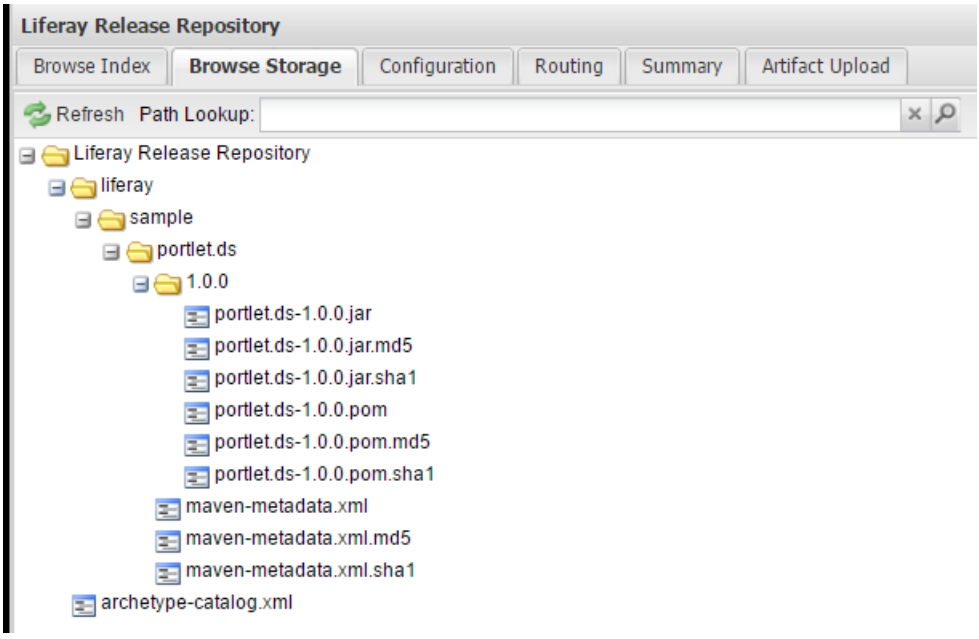


Figure 35.2: Your repository server now provides access to your Liferay Maven artifacts.

Awesome! You can now share your Liferay module projects with anyone by deploying them as artifacts to your remote repository!

35.7 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 projects; the answer is a resounding yes!

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 article. In some cases, you should not use this template 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.

If you have questions about upgrading your legacy Service Builder project, see the From Liferay 6 to 7 tutorial section.

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>1.0.182</version>
            <configuration>
                <apiDirName>../basic-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>
                <modelHintsFileName>src/main/resources/META-INF/portlet-model-hints.xml</modelHintsFileName>
                <mergeModelHintsConfigs>src/main/resources/META-INF/portlet-model-hints.xml</mergeModelHintsConfigs>
                <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 sample `basic-service` module. When running Service Builder with this configuration, the project's API classes are generated in the `basic-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 Plugin 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>1.0.3</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.

35.8 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>2.1.0</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.

35.9 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>1.1.4</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 defines tag 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>2.1.0</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.

35.10 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 either an archetype or via Blade CLI. You can generate a workspace via archetype by executing the following command:

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.workspace \
    -DgroupId=[GROUP_ID] \
    -DartifactId=[WORKSPACE_NAME] \
    -Dversion=[VERSION]
```

If you have Blade CLI installed, and want to use that instead of generating an archetype, you can run the following command:

```
blade init -b maven [WORKSPACE_NAME]
```

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/7.1.0-ga1/liferay-ce-portal-tomcat-7.1.0-ga1-20180703012531655.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 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 some configurable workspace properties you can set in the root `pom.xml` file:

- `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.environment`: the name of a `configs` subfolder holding the Liferay DXP server configuration to use. See Testing Modules for more information.

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.bundle.url>https://releases-cdn.liferay.com/portal/7.1.0-ga1/liferay-ce-portal-tomcat-7.1.0-ga1-
20180703012531655.zip</liferay.workspace.bundle.url>
    <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 Project 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 36

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.

36.1 Installing the Liferay IntelliJ Plugin

There are two ways to install the IntelliJ plugin for IDEA:

- via IntelliJ Marketplace
- via Zip file

Follow the steps pertaining to your preferred installation process.

Installing Via IntelliJ Marketplace

1. In IntelliJ, navigate to *File → Settings → Plugins*.
2. In the Marketplace tab, search for *Liferay* in the provided search bar.
3. Click *Install* next to the Liferay IntelliJ Plugin.
4. After the plugin has downloaded, select *Restart IDE*.

Once IntelliJ restarts, the Liferay IntelliJ plugin is installed and ready for use.

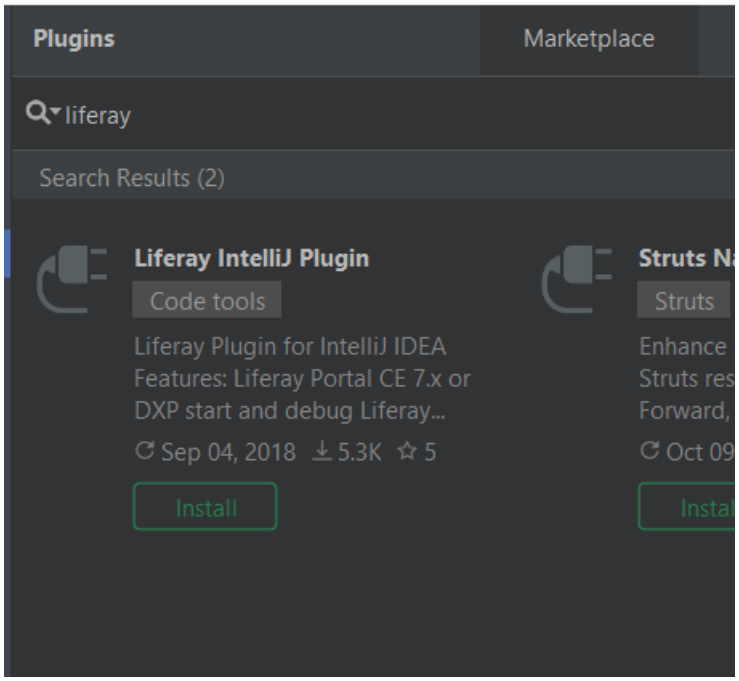


Figure 36.1: IntelliJ Marketplace offers a streamlined way to install plugins.

Installing Via Zip File

1. Navigate to the JetBrains' Liferay IntelliJ plugin page and download it to your local machine.
2. In IntelliJ, navigate to *File → Settings → Plugins*.
3. Click the gear icon from the top menu and select *Install Plugin from Disk...*.
4. Navigate to the Installed tab in the top menu and select *Restart IDE*.

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!

36.2 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*.

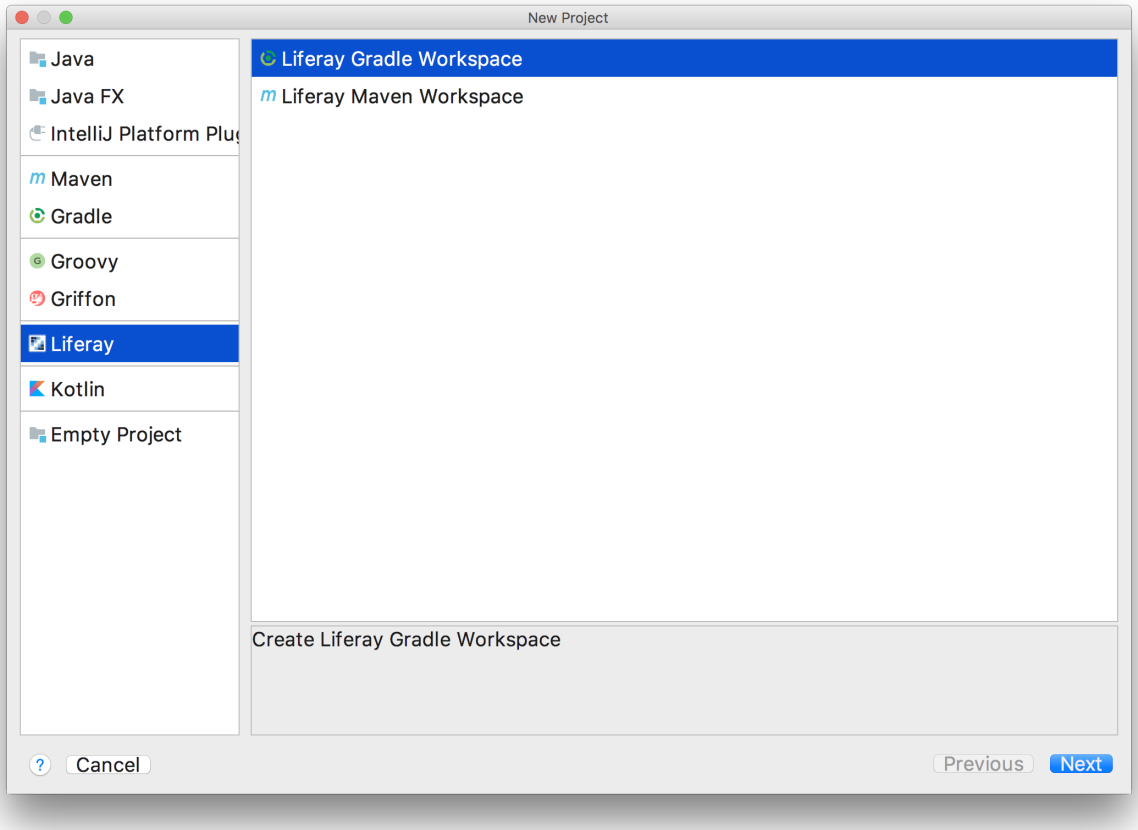


Figure 36.2: Choose *Liferay Gradle Workspace* or *Liferay Maven Workspace*, depending on the build you prefer.

4. Specify your workspace's name, location, intended Liferay DXP version, target platform, 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!

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*.

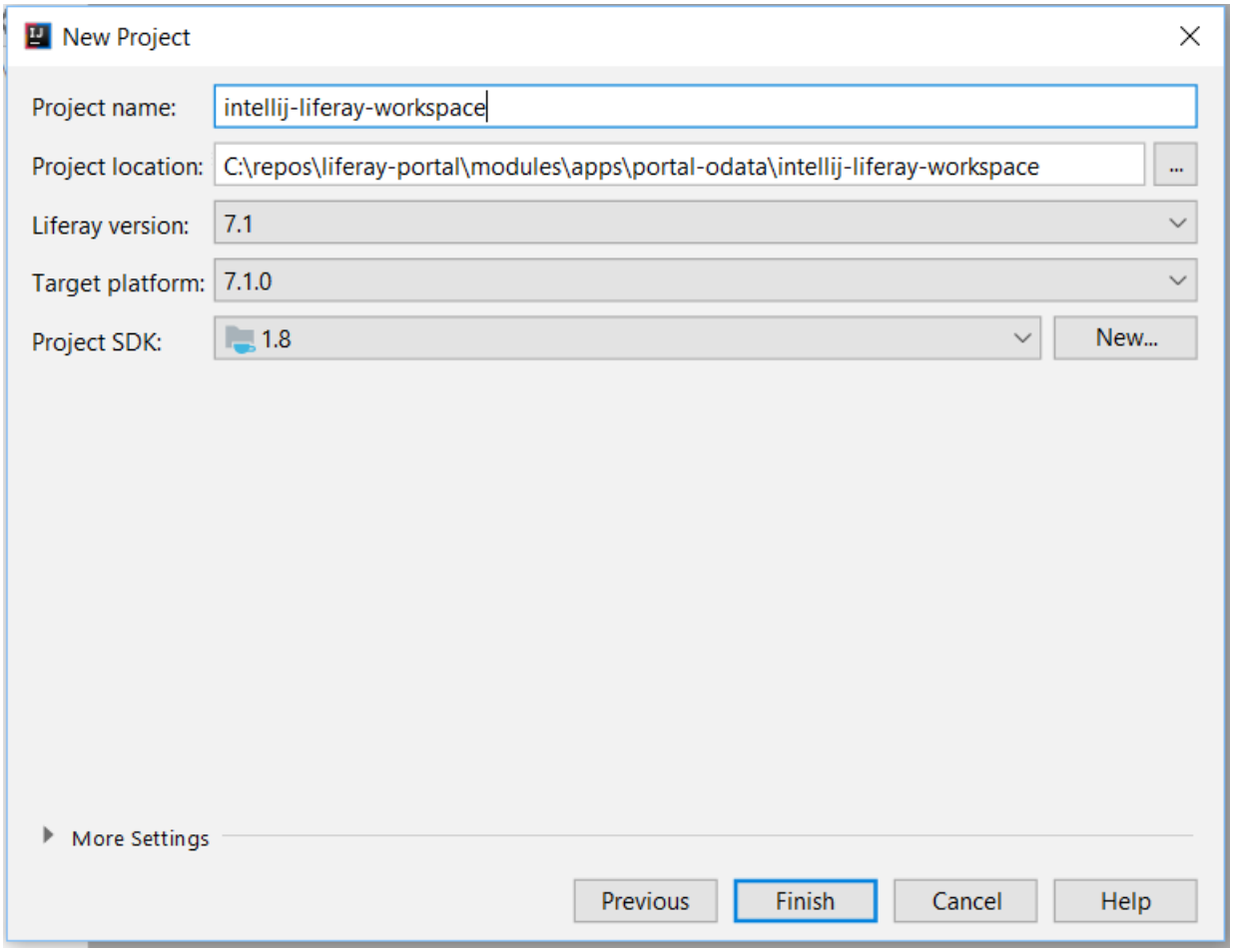


Figure 36.3: Specify your workspace's configurations.

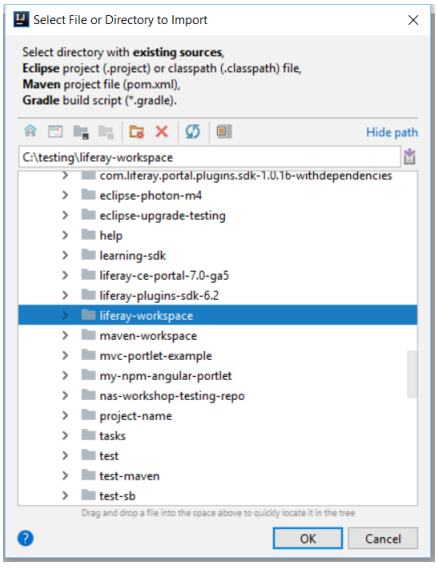


Figure 36.4: Specify your workspace's configurations.

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!

36.3 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, 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*.

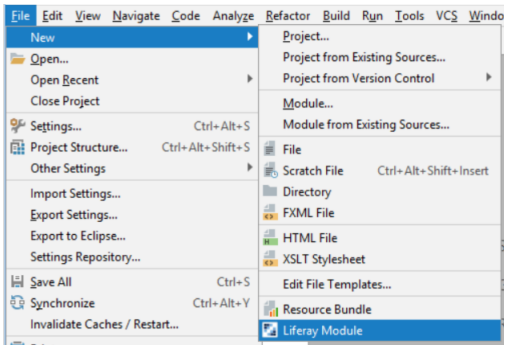


Figure 36.5: Selecting *Liferay Module* opens the New Liferay Modules wizard.

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.

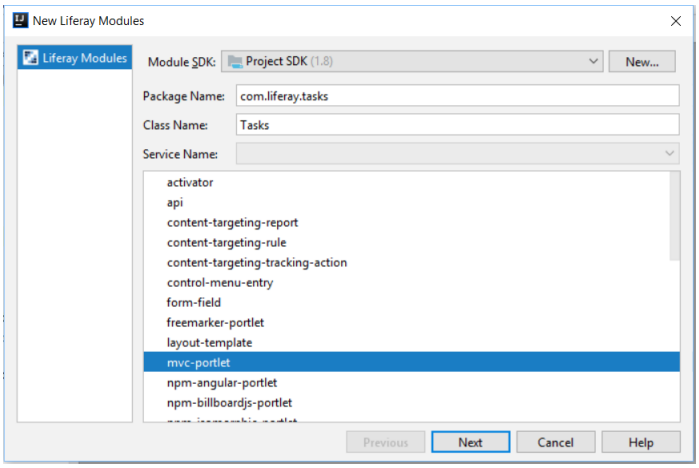


Figure 36.6: Choose the project template to create your module.

36.4 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.

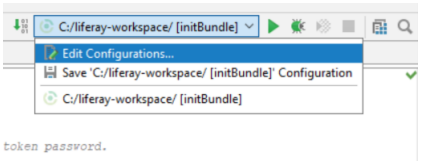


Figure 36.7: You have several options to choose from the server dropdown menu.

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* .

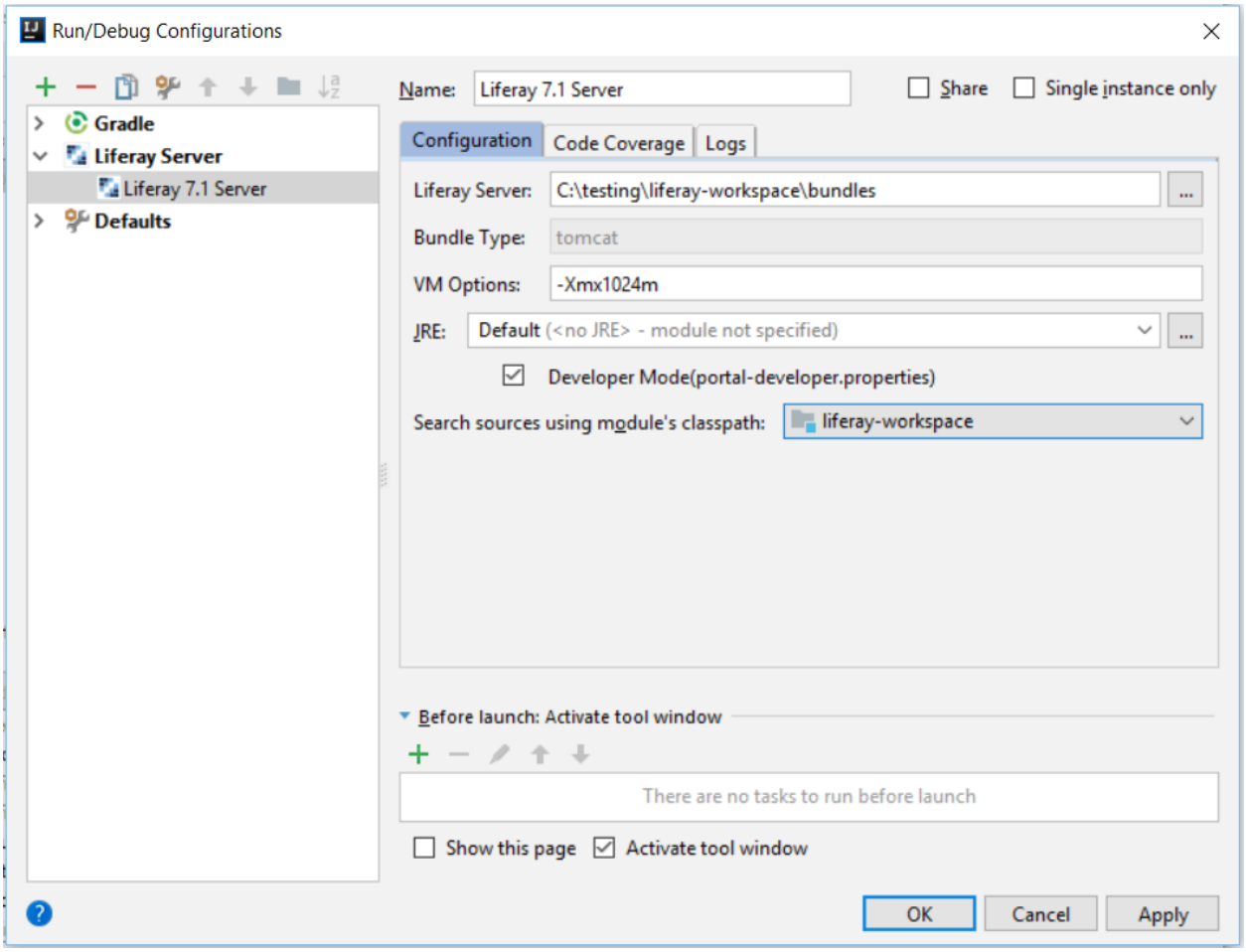


Figure 36.8: Set your Liferay server's configurations in the Run/Debug Configurations menu.

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.
- *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!

36.5 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.

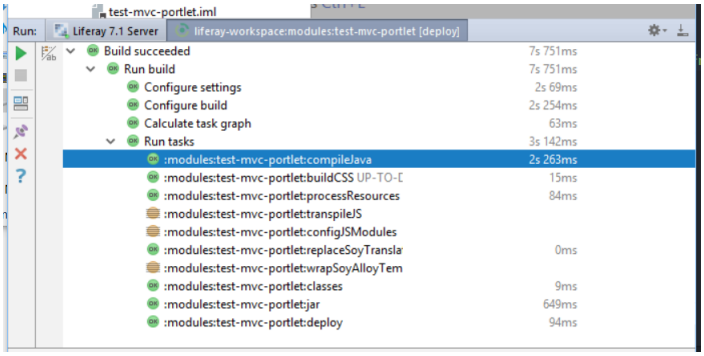


Figure 36.9: Verify that your project build successfully.

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:

```
INFO  [fileinstall-C:/liferay-workspace/bundles/osgi/modules][BundleStartStopLogger:35] STARTED com.liferay.docs_1.0.0 [652]
```

The `watch` Blade CLI task deploys your project. This watches your local project and propagates saved changes to the deployed project. With this, project updates are viewable almost instantaneously from your Liferay server. For more info on the `watch` task, see the Deploying Projects with Blade CLI article.

That's it! You've successfully deployed your project to Liferay DXP!

CHAPTER 37

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.

You can also use Blade CLI to create samples by running this command:

```
blade samples [SAMPLE_NAME]
```

For example, the following command generates the ds-portlet sample:

```
blade samples ds-portlet
```

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 DXP.

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.

37.1 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.

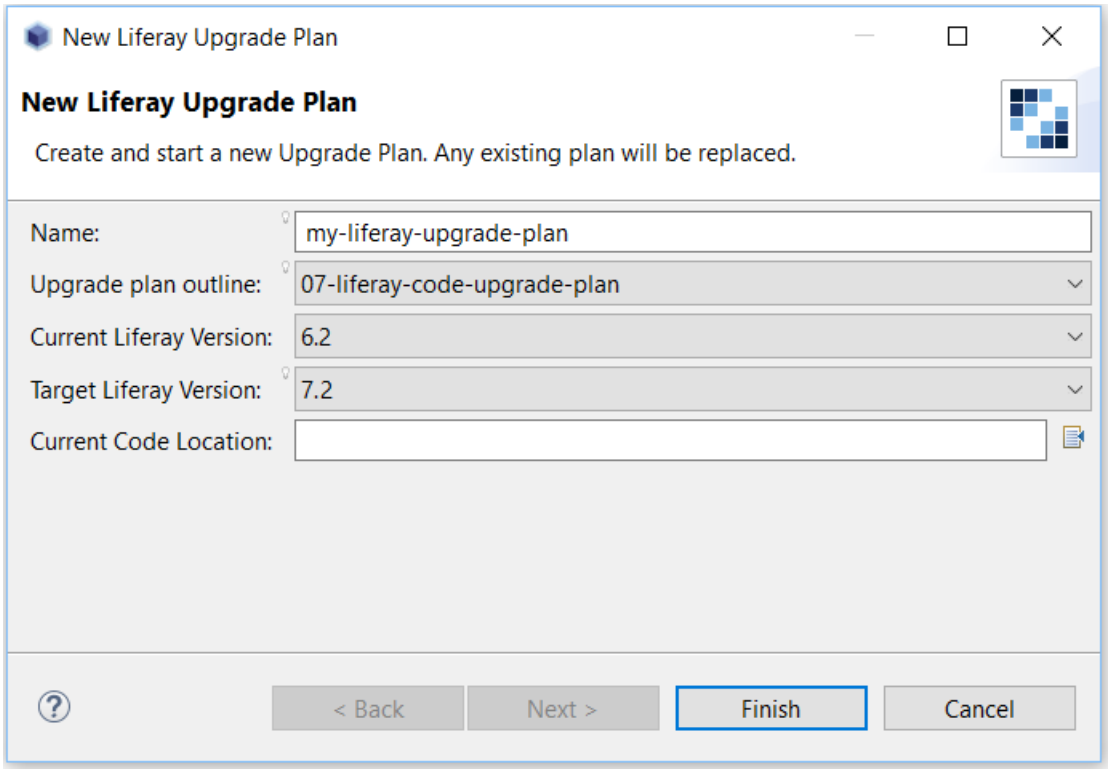


Figure 37.1: Configure your upgrade plan before beginning the upgrade process.

- *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.
- *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.

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

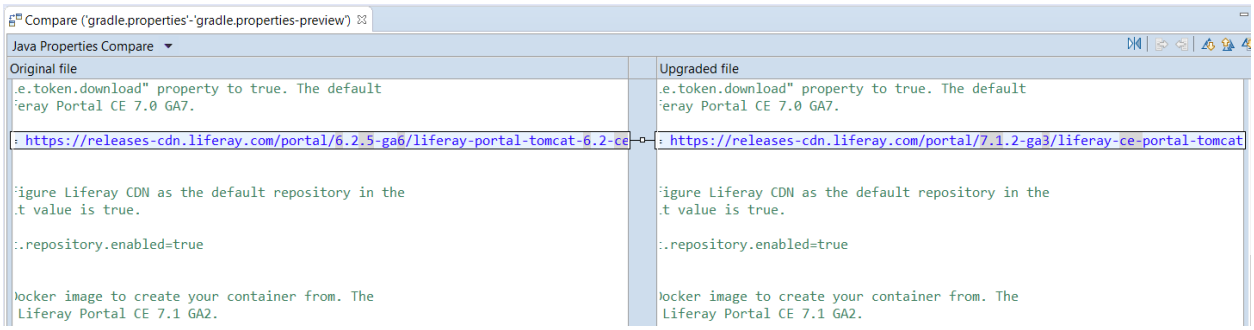


Figure 37.2: You can preview the Upgrade Planner's automated updates before you perform them.

37.2 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!

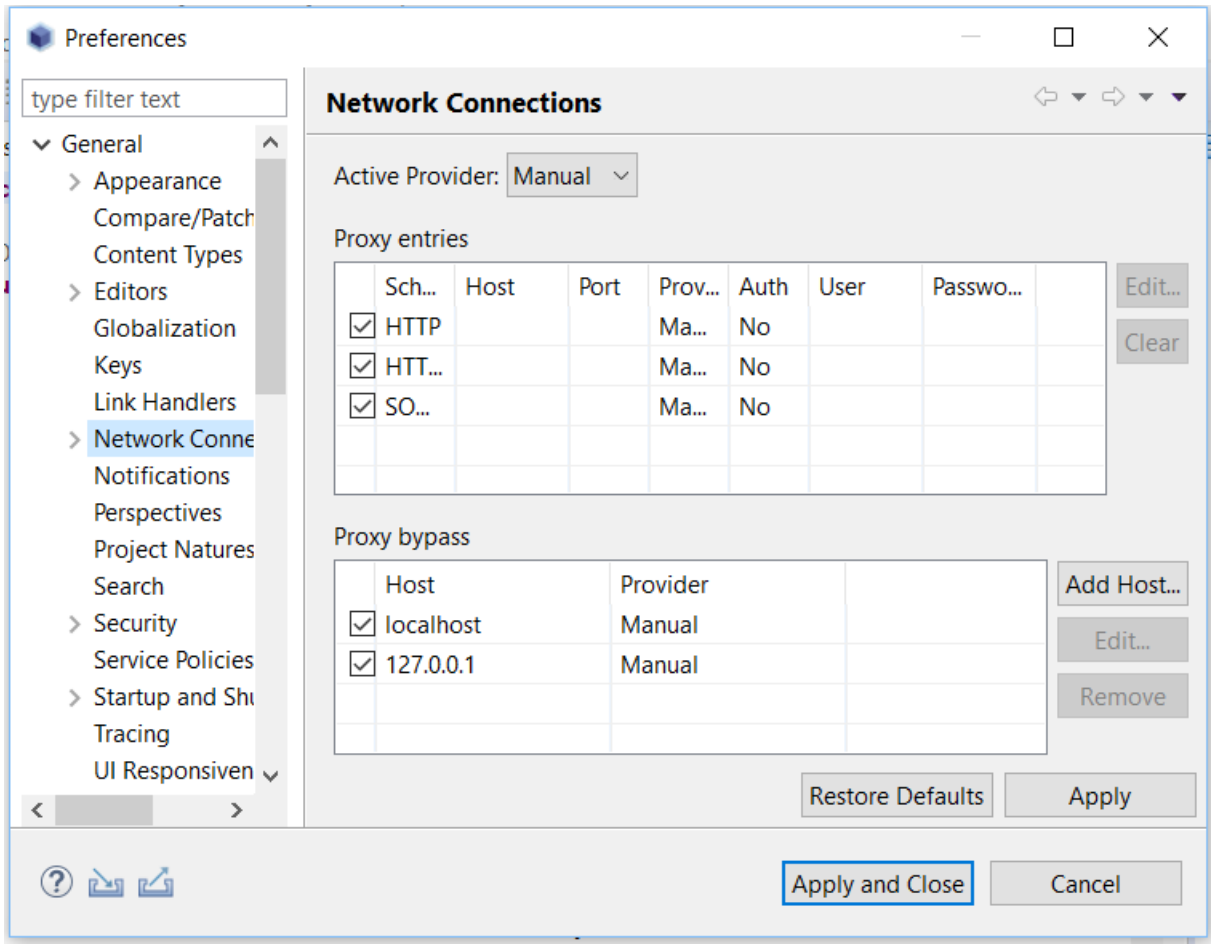


Figure 37.3: You can configure your proxy settings in Dev Studio's Network Connections menu.

CHAPTER 38

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. One key difference between portlets and other web apps 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 components of the interface–is handled by other components. Another difference is that portlets run only in a portal server. 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 (if they have permission) 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? The links above show the complete definition; here we'll briefly summarize 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

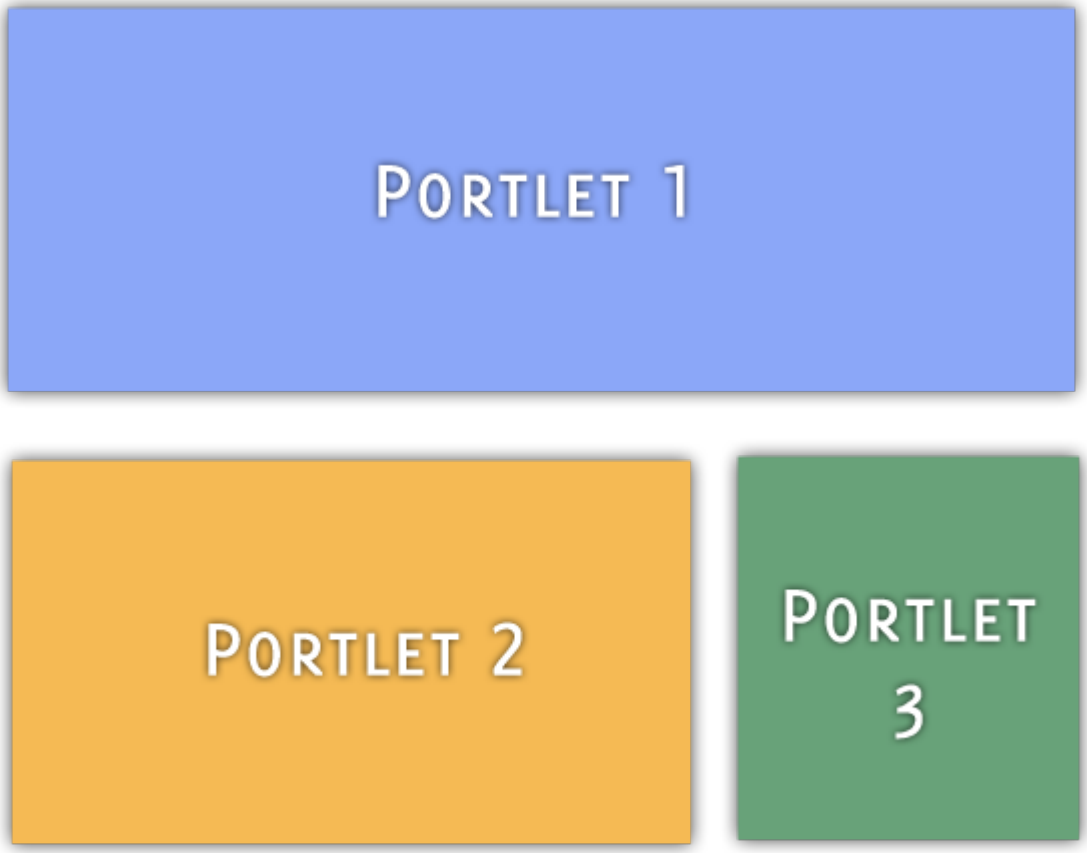


Figure 38.1: You can place multiple portlets on a single page.

runs when any portlets on the page complete the Action or Event phases.

- **Action:** In response to a user action, the Action phase performs some operations that change 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 inter-process communication (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, 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 DXP. 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 our 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 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.

So what's the benefit to adopting Liferay's frameworks and APIs? There are several:

- They follow Liferay's design patterns. The better you understand them, the better you understand Liferay DXP.

- They are the result of nearly 15 years of portlet development.
- They provide many conveniences that make development easier and faster.
- They make your applications fit more naturally with the rest of the system.
- If necessary, they're easy to migrate from, because they're built on top of the standards.

With that said, you can use a variety of technologies to develop portlets. This section shows you how to develop portlets using the following frameworks and techniques:

- Liferay's MVCPortlet
- Liferay Soy Portlet
- Spring MVC
- Making URLs Friendlier
- Automatic Single Page Applications
- Applying Clay Styles to Your App
- Creating Layouts Inside Portlets
- Using JavaScript Inside Portlets

38.1 Related Topics

Configuring Dependencies
    Importing Packages
    Exporting Packages

CHAPTER 39

LIFERAY MVC PORTLET

Web applications often follow 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 is not the first time you've heard about Model View Controller. In this article you must stay focused, because there are 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), here's an overview of the Liferay MVC Portlet:

- MVC layers and modularity
- Liferay MVC command classes
- Liferay MVC portlet component
- Simple MVC portlets

Review how each layer of the *Moody Vase Conscription* pattern helps you separate the concerns of your application.

39.1 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. For the web, 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.

39.2 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 serve resources based on 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.

39.3 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 the `javax.portlet.Portlet` interface. Import that, and not, for example, `com.liferay.portal.kernel.model.Portlet`.

Note: The DTD liferay-portlet-app_7_1_0.dtd defines all the Liferay-specific attributes you can specify as properties in your portlet components.

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.

39.4 A Simpler MVC Portlet

With all this focus on MVC commands, don't be concerned that you'll be forced into a more complex pattern than you need, especially if you're developing only a small MVC application. Not so; just put all 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 JSP paths in `mvcPath` parameters.

```
<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. To get into the details of creating an MVC Portlet application, follow the Creating an MVC Portlet tutorial.

CHAPTER 40

CREATING AN MVC PORTLET

MVC Portlet applications are web modules containing at least one portlet class that's registered in Liferay's runtime environment as a component. Web modules describe themselves using standard OSGi metadata and can use any build environment.

Here are the general steps for implementing a Liferay MVC Portlet component module:

1. Configuring a Web module
2. Specifying OSGi metadata
3. Creating a portlet Component

Start by creating a web module for your portlet.

40.1 Step 1: Configuring a Web Module

The folder structure for a web module generally follows this pattern:

- 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

The MVC portlet template, available for both Maven and Gradle in Liferay Dev Studio DXP and Blade CLI, makes creating such Web modules a snap. Of course you're not tied to using Gradle or bnd to build your project. However, you must build your module as a JAR and define your module with proper OSGi headers.

40.2 Step 2: Specifying OSGi Metadata

OSGi metadata describes your module to the OSGi runtime environment. At a minimum, you should specify the bundle symbolic name and the bundle version. We recommend a human-readable bundle name.

```
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 is generated from the project's folder path, which is suitable for many cases. Liferay's convention is to specify the root package name as your bundle symbolic name.

40.3 Step 3: Creating a Portlet Component

The OSGi Declarative Services component model makes it easy to publish service implementations to the OSGi runtime. For example, publishing your portlet class as a `javax.portlet.Portlet` requires an `@Component` annotation like this one:

```
@Component(
    immediate = true,
    service = Portlet.class
)
public class LiferayMVCPortlet extends MVCPortlet {
}
```

The `immediate = true` attribute tells the runtime to publish the portlet as soon as its dependencies resolve. The attribute `service = Portlet.class` specifies that the portlet provides the `javax.portlet.Portlet` service.

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 must be configured:

```
@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.name=MyMVCPortlet",
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
```

```
    service = Portlet.class
)
public class LiferayMVCPortlet extends MVCPortlet {
}
```

Liferay's MVC portlet template includes these component properties in the portlet class it generates.

Some of the properties might look familiar to you if you've developed Liferay MVC portlets for Liferay Portal 6.2. That's because they're the same as the XML attributes you used to specify in `liferay-portlet.xml`, `liferay-display.xml`, and `portlet.xml`. The mapping of portlet descriptors to OSGi properties can you help find OSGi properties for descriptors you already know.

To keep compatibility with the JSR-168 and JSR-286 portlet specs, these DTDs define the Liferay-specific portlet attributes:

- liferay-portlet-app_7_1_0.dtd.
- liferay-display_7_1_0.dtd

For example, consider the `<instanceable>` element from `liferay-portlet-app_7_1_0.dtd`. To specify that property in your Component, use this syntax in your `@Component` property list:

```
"com.liferay.portlet.instanceable=true",
```

The properties namespaced with `javax.portlet....` are elements of the portlet.xml descriptor.

Also note that you can use the `com.liferay.portlet.display-category` property to create nested categories. Use // to separate the category root and all categories and sub-categories that comprise your portlet's category path. Here's an example:

```
com.liferay.portlet.display-category=root//category.category1//category.category2
```

You now know how to extend Liferay's `MVCPortlet` and register it as a Component in the OSGi runtime. It's time to write your controller code.

40.4 Writing Controller Code

In MVC, your controller receives requests from the front-end, and it pulls data from the back-end. It's a traffic director: it provides data to the right front-end view for display to the user, and it takes data the user entered in the front-end and passes it to the right back-end service. For this reason, the controller must process requests from the front-end, and it must determine the right front-end view to pass data back to the user.

If you have a small application that's not 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 Render Command classes, MVC Action Command classes, and MVC Resource Command classes. This tutorial demonstrates implementing controller logic for small applications.

In this tutorial you'll learn to implement a Liferay MVC portlet with all the controller code in the `-Portlet` class. It involves these things:

- Action methods

- Render logic
- Setting and retrieving request parameters and attributes

Start with creating action methods.

Action Methods

If you have a small application, you can implement all your controller logic in the portlet class you created in the last step. It can act as your controller by itself. Use action methods to process requests. Here's a sample action method:

```
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");
    }

}
```

This action has one job: call a service to add a Guestbook. If this call succeeds, the message `"guestbookAdded"` is associated with the request and added to the `SessionMessages` object. If an exception is thrown, it's caught, and the class name is associated with the request and added to the `SessionErrors` object and the response is set to render `edit_guestbook.jsp`. Setting the `mvcPath` render parameter is a Liferay `MVCPortlet` framework convention that denotes the next view to render to the user.

While action methods respond to user actions, render logic determines the view to display to the user. Render logic is next.

Render Logic

Here's how MVC Portlet determines which view to render. Note the `init-param` properties you set in your component:

```
"javax.portlet.init-param.template-path=/",
"javax.portlet.init-param.view-template=/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 in your project's root `resources` folder. That's why it's important to follow Liferay's standard folder structure. The `view-template` property directs the default rendering to `view.jsp`.

Here's the path of a hypothetical Web module's resource folder:

```
docs.liferaymvc.web/src/main/resources/META-INF/resources
```

Based on that resource folder, 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 portlet's `init` method (e.g., your portlet's override of `MVCPortlet.init()`) is called, the initialization parameters you specify are read and used to direct rendering to the default JSP. Throughout the controller, you can render different views (JSP files) by setting the render parameter `mvcPath` like this:

```
actionResponse.setRenderParameter("mvcPath", "/error.jsp");
```

It's possible to avoid render logic by using initialization parameters and render parameters, but most of the time you'll override the portlet's `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);

}
```

This render logic provides the view layer with data to display to the user. The `render` method above sets the render request attribute `guestbookId` with the ID of a guestbook to display. If guestbooks exist, it chooses the first. Otherwise, it creates a guestbook and sets it to display. Lastly the method passes the render request and render response objects to the base class via its `render` method.

Note: Are you wondering how to call Service Builder services in 7.0? Invoking Services from Service Builder Code can help. In short, obtain a reference to the service by annotating one of your fields of that service type with the `@Reference` Declarative Services annotation.

```
@Reference
private GuestbookService _guestbookService;
```

Once done, you can call the service's methods.

```
_guestbookService.addGuestbook(serviceContext.getUserId(), "Main",
        serviceContext);
```

Before venturing into the view layer, the next section demonstrates ways to pass information between the controller and view layers.

Setting and Retrieving Request and Response Parameters and Attributes

You can use a handy utility class called `ParamUtil` to retrieve parameters from an `ActionRequest` or a `RenderRequest`.

For example, a JSP could pass a parameter named `guestbookId` in an action URL.

```
<portlet:actionURL name="doSomething" var="doSomethingURL">
    <portlet:param name="guestbookId"
            value="<%= String.valueOf(entry.getGuestbookId()) %>" />
</portlet:actionURL>
```

The `<portlet:actionURL>` tag's `name` attribute maps the action URL to a controller action method named `doSomething`. Triggering an action URL invokes the corresponding method in the controller.

The controller's `doSomething` method referenced in this example can then get the `guestbookId` parameter value from the `ActionRequest`.

```
long guestbookId = ParamUtil.getLong(actionRequest, "guestbookId");
```

To pass information back to the view layer, the controller code can set render parameters on response objects.

```
actionResponse.setRenderParameter("mvcPath", "/error.jsp");
```

The code above sets a parameter called `mvcPath` to JSP path `/error.jsp`. This causes the controller's render method to redirect the user to the JSP `/error.jsp`.

Your controller class can also set attributes into response objects using the `setAttribute` method.

```
renderRequest.setAttribute("guestbookId", guestbookId);
```

JSPs can use Java code in scriptlets to interact with the request object.

```
<%
    long guestbookId = Long.valueOf((Long) renderRequest
            .getAttribute("guestbookId"));
%>
```

Passing information back and forth from your view and controller is important, but there's more to the view layer than that. The view layer is up next.

40.5 Configuring the View Layer

This section briefly covers how to get your view layer working, from organizing your imports in one JSP file, to creating URLs that direct processing to methods in your portlet class.

Note: As you create JSPs, you can apply Clay styles to your app to match Liferay's apps.

Liferay's best practice puts all Java imports, tag library declarations, and variable initializations into a JSP called `init.jsp`. If you use Blade CLI or Liferay Dev Studio DXP to create a module based on the `mvc-portlet` project template, these taglib declarations and initializations are added automatically 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 to include the `init.jsp` in your other JSPs:

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

JSPs can use action URLs to invoke controller methods. To create a link to another page, use a render URL with the `mvcPath` parameter. The `<portlet:renderURL>` tag constructs the URL and assigns it to a variable.

```
<portlet:renderURL var="searchURL">
        <portlet:param name="mvcPath" value="/admin/view.jsp" />
</portlet:renderURL>
```

The render URL is assigned to the `searchURL` variable specified by the `var` attribute. The `<portlet:param>` tag above assigns JSP path `/admin/view.jsp` to the render parameter `mvcPath`. The controller's `render` method gets the JSP path from the `mvcPath` parameter to render the following JSP:

```
docs.liferaymvc.web/src/main/resources/META-INF/resources/admin/view.jsp
```

To invoke the render URL, assign its variable (the one set to the `var` attribute of the `<portlet:renderURL>`) to an action for a UI component, such as a button or navigation bar item.

Action methods are different because they invoke an action (i.e., code), rather than just linking to another page. In your JSP, use a `<portlet:actionURL>` tag to create an action URL and then assign that URL as an action for a UI component. Here's an action URL that calls a controller method named `doSomething`.

```
<portlet:actionURL name="doSomething" var="doSomethingURL">
    <portlet:param name="redirect" value="<%= redirect %>" />
</portlet:actionURL>
```

The portlet parameter named `redirect` is assigned to a JSP path for the portlet to redirect to after invoking the portlet action. This action URL is assigned to a variable named `doSomethingURL`. As with a render URL, you can assign an action URL to a UI component action by the action URL's variable (the one set to `var`).

These simple examples demonstrate how the Liferay MVC framework facilitates communication between a smaller application's view layer and controller.

40.6 Beyond the Basics for Portlets

The tutorial you've just completed should get you up and running with a Liferay MVC Web module, but there's more to know about creating an app in Liferay. To support more actions, complex render logic, and or serving resources, continue reading the MVC command tutorials that follow.

Regardless of your application's size or complexity, there are more conveniences and features to leverage in your portlets. Here are a few useful jumping off points:

- Making URLs Friendlier
- Applying Clay Styles to your App
- Localizing your Application
- Liferay's Workflow Framework
- Model Listeners
- Application Security
- Asset Framework
- Service Builder

Enjoy creating your own portlets!

40.7 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, configure your view layer and use the `<portlet:actionURL>` tag to create the action URL in your JSP. For example, the Blogs app's `edit_entry.jsp` file defines the following action URL for editing blog entries:

```
<portlet:actionURL name="/blogs/edit_entry" var="editEntryURL" />
```

The `name` attribute declares a variable to hold the portlet action URL object. Assign that variable to a UI component, such as a button or icon. When the user triggers the UI component, the `*MVCActionCommand` class that matches the action URL processes the action request and response. Create an action class by implementing the `MVCActionCommand` interface, or extending the `BaseMVCActionCommand` class. The latter may save you time, since it already implements `MVCActionCommand`.

Naming your `*MVCActionCommand` class after the action it performs is a good convention. For example, if your action class edits some kind of entry, you could name its class `EditEntryMVCActionCommand`. If your application has several MVC command classes, naming them this way helps differentiate them.

Your *MVCActionCommand class must also have an @Component annotation like the following example. Set the property javax.portlet.name to your portlet's internal ID. Set 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 an MVCActionCommand service, 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
}
```

The Blogs app's EditEntryMVCActionCommand class is a real world example of a *MVCActionCommand 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 {

        @Override
        protected void doProcessAction(
                ActionRequest actionRequest, ActionResponse actionResponse)
            throws Exception {

            String cmd = ParamUtil.getString(actionRequest, Constants.CMD);

            try {
                BlogsEntry entry = null;

                UploadException uploadException =
                    (UploadException)actionRequest.getAttribute(
                        WebKeys.UPLOAD_EXCEPTION);

                if (uploadException != null) {
                    Throwable cause = uploadException.getCause();

                    if (uploadException.isExceededFileSizeLimit()) {
                        throw new FileSizeException(cause);
                    }

                    if (uploadException.isExceededLiferayFileItemSizeLimit()) {
                        throw new LiferayFileItemException(cause);
                    }

                    if (uploadException.isExceededUploadRequestSizeLimit()) {
                        throw new UploadRequestSizeException(cause);
                    }

                    throw new PortalException(cause);
                }
                else if (cmd.equals(Constants.ADD) ||
```

```
                    cmd.equals(Constants.UPDATE)) {

                Callable<BlogsEntry> updateEntryCallable =
                    new UpdateEntryCallable(actionRequest);

                entry = TransactionInvokerUtil.invoke(
                    _transactionConfig, updateEntryCallable);
            }
            else if (cmd.equals(Constants.DELETE)) {
                deleteEntries(actionRequest, false);
            }
            else if (cmd.equals(Constants.MOVE_TO_TRASH)) {
                deleteEntries(actionRequest, true);
            }
            else if (cmd.equals(Constants.RESTORE)) {
                restoreTrashEntries(actionRequest);
            }
            else if (cmd.equals(Constants.SUBSCRIBE)) {
                subscribe(actionRequest);
            }
            else if (cmd.equals(Constants.UNSUBSCRIBE)) {
                unsubscribe(actionRequest);
            }

            ... do more action processing
        }

        ... handle exceptions
    }
}
```

The `@Component`'s multiple `javax.portlet.name` property values make this `*MVCActionCommand` class available to those portlets as a Service Component. The `mvc.command.name` property setting `/blogs/edit_entry` matches the `actionURL`'s `name` attribute shown earlier, and the `service` property set to `MVCActionCommand.class` makes the class an `MVCActionCommand` Service Component.

The `EditEntryMVCActionCommand` class extends `BaseMVCActionCommand` and overrides the `doProcessAction` method. Similarly, `*MVCActionCommand` classes that implement `MVCActionCommand` directly must implement the `processAction` method. Both methods process resource requests and responses via their `javax.portlet.ActionRequest` and `javax.portlet.ActionResponse` parameters, respectively.

`EditEntryMVCActionCommand`'s `doProcessAction` method gets the value of a command parameter named by constant `Constants.CMD` from the `ActionRequest`.

```
String cmd = ParamUtil.getString(actionRequest, Constants.CMD);
```

Then the `doProcessAction` method checks whether an entry-related upload occurred or handles any exceptions the upload throws. Based on the command (stored in `cmd`) accessed from the action request, one of the following actions is performed:

- add or update an entry
- delete an entry
- move an entry to the Recycle Bin
- restore an entry from the Recycle Bin
- subscribe a user to a blog
- unsubscribe a user from a blog

`EditEntryMVCActionCommand`'s `doProcessAction` method continues with some more processing and prepares to redirect the portlet to an appropriate view. This shows you can do as much as you need for processing your portlet's actions.

Note: Liferay Blade Sample `action-command-portlet` demonstrates implementing `MVCActionCommand` directly.

Now you can create your own action URLs and `*MVCActionCommand` classes in your applications that use Liferay's MVC framework. Your `*MVCActionCommands` can do whatever you need them to do.

Related Topics

Creating an MVC Portlet
    MVC Render Command
    MVC Resource Command
    MVC Command Overrides

40.8 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.

First, configure your view layer and use the `<portlet:renderURL>` to create the render URL in your JSP. For example, the following render URL invokes an MVC render command named `/hello/edit_entry`. This might direct the user to a page with a form for editing.

```
<portlet:renderURL var="editEntryURL">
    <portlet:param name="mvcRenderCommandName" value="/hello/edit_entry" />
    <portlet:param name="entryId" value="<%= String.valueOf(entry.getEntryId()) %>" />
</portlet:renderURL>
```

The `<portlet:param>` named `mvcRenderCommandName` declares the render URL. The `<portlet:param>` named `entryId` declares a variable to hold the portlet render URL object. Assign that variable to a UI component such, as a button or menu item. When the user triggers the UI component, the `*MVCRenderCommand` class that matches the render URL processes the render request and response.

What is it you want to do when a particular portlet render URL is invoked? By implementing the `MVCRenderCommand` interface and overriding its `render` method, you can perform your own logic to render JSPs. Some `*MVCRenderCommands`, such as the one below, always render the same JSP.

```
public class BlogsViewMVCRenderCommand implements MVCRenderCommand {

    @Override
    public String render(
        RenderRequest renderRequest, RenderResponse renderResponse) {

        return "/blogs/view.jsp";
    }

}
```

Other `*MVCRenderCommands` render JSPs based on 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";
}
```

In the method above, if no exceptions are thrown on invoking `ActionUtil.getEntry`, the method renders `/hello/edit_entry.jsp`. If a `NoSuchEntryException` is thrown, it renders `/hello/error.jsp`. If any other exception is thrown, the method re-throws it as a `PortletException`.

To respond to a particular render URL, your `MVCRenderCommand` must be an OSGi Declarative Services Component (e.g., annotated with @Component) that specifies these properties:

- `javax.portlet.name`
- `mvc.command.name`

Here's an example of these two properties:

```
 "javax.portlet.name=" + HelloWorldPortletKeys.HELLO_WORLD,
 "mvc.command.name=/hello/edit_entry"
```

The portlet name, in this case, is defined by the constant `HelloWorldPortletKeys.HELLO_WORLD`. The `mvc.command.name` is set to `/hello/edit_entry`, which seems to indicate the MVC render command is related to editing entries–just a hunch.

The Component must also publish to the OSGi runtime as a `MVCRenderCommand.class` service. Here's a basic Component that specifies the example properties and publishes itself as an `MVCRenderCommand.class` service:

```
@Component(
    immediate = true,
    property = {
       "javax.portlet.name=" + HelloWorldPortletKeys.HELLO_WORLD,
       "mvc.command.name=/hello/edit_entry"
    },
    service = MVCRenderCommand.class
)
public class EditEntryMVCRenderCommand implements MVCRenderCommand {
    ...
}
```

The `mvc.command.name` value `/hello/edit_entry` matches the value of `portlet:renderURL`'s `mvcRenderCommand` parameter shown earlier. That render URL invokes this `*MVCRenderCommand` class. In fact, any render URL of JSPs in this portlet (`HelloWorldPortletKeys.HELLO_WORLD`) whose `mvcRenderCommand` is `/hello/edit_entry` invokes this `*MVCRenderCommand`.

To make an `MVCRenderCommand` respond to multiple portlets, add them to your `@Component` as `javax.portlet.name` properties assigned to the portlet names. Likewise, to make it respond to

multiple render URLs, add them as `mvc.command.name` properties. 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 easy to implement and can respond to multiple command names for multiple portlets.

Related Topics

Creating an MVC Portlet
    MVC Resource Command
    MVC Action Command
    MVC Command Overrides

40.9 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, configure your view layer and 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" />
```

The `id` attribute declares the resource URL. The `var` attribute declares a variable to hold the portlet resource URL object. Assign that variable to a UI component, such as a button or icon. When the user triggers the UI component, the `*MVCResourceCommand` class that matches the resource URL 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 helps differentiate them.

Your `*MVCResourceCommand` class must also have an `@Component` annotation like the following example. 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/login/login-web/src/main/java/com/liferay/login/web/internal/por`

```
@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

Creating an MVC Portlet
- MVC Render Command
- MVC Action Command
- MVC Command Overrides
- OSGi Basics for Liferay Development

CHAPTER 41

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.

41.1 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/`
            · `MySoyPortletRegister.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: 2.0.7 Provide-Capability:
soy;
type="hello-soy";
version:Version="1.0.10" Require-Capability:
soy;
filter:="(type=metal)" Web-ContextPath: /hello-soy-web
The Provide-Capability header specifies that this bundle provides the soy capability, so the template engine can track the bundle. The Require-Capability header specifies that the bundle requires modules that provide the capability soy with a type of metal to work. The Web-ContextPath header specifies the relative path of the application so you can reference resources.

Specifying JavaScript Dependencies

Specify the JavaScript module dependencies in your package.json. At a minimum, you should have the following dependencies and configuration parameters. Always use the latest component versions (the versions shown below may not be the latest).

```
{
    "dependencies": {
        "metal-component": "^2.16.8",
        "metal-soy": "^2.16.8"
    },
    "scripts": {
        "build": "liferay-npm-scripts build",
        "checkFormat": "liferay-npm-scripts check",
    "format": "liferay-npm-scripts fix"
    },
    "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 {
    compileOnly group: "com.liferay", name: "com.liferay.portal.portlet.bridge.soy.api", version: "1.0.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "2.0.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.util.java", version: "3.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
    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"
}
```

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 Register Component

Create a Soy Portlet component that extends the `SoyPortletRegister` 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 = SoyPortletRegister.class
)
public class MySoyPortletRegister extends SoyPortletRegister {

}
```

Liferay DXP's `SoyPortletRegister` class is imported in the `SoyPortlet` class which extends `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.single-page-application=false",
            "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=/META-INF/resources/",
            "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 = SoyPortletRegister.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 full list of the available Liferay-specific portlet component properties in the `liferay-portlet-app_7_1_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 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 specifies the location of your Soy templates. The / above means that the Soy files are located in the project's root `resources` folder. That's why it's important to follow the 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 that you want to render. 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. Below is an explanation of the example above:

- The MVC command name is Navigation (the Soy template with namespace Navigation). This means that this logic is for the Navigation view.
- A PortletURL (navigationURL) is defined and its mvcRenderCommandName is set to View(the Soy template with namespace View).
- The navigationURL is converted to a String and passed as the variable navigationURL to the Navigation Soy template with the template.put() method.

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 HelloSoyViewMVCRenderCommand class that also defines a navigationURL parameter:

```
public class HelloSoyViewMVCRenderCommand implements MVCRenderCommand {

@Override
public String render(
        RenderRequest renderRequest, RenderResponse renderResponse) {

        Template template = (Template)renderRequest.getAttribute(
            WebKeys.TEMPLATE);

        ThemeDisplay themeDisplay = (ThemeDisplay)renderRequest.getAttribute(
            WebKeys.THEME_DISPLAY);
```

```
        template.put("layouts", themeDisplay.getLayouts());

        PortletURL navigationURL = renderResponse.createRenderURL();

        navigationURL.setParameter("mvcRenderCommandName", "Navigation");

        template.put("navigationURL", navigationURL.toString());

        template.put("releaseInfo", ReleaseInfo.getReleaseInfo());

        return "View";
    }

}
```

Below is an explanation of the example above:

- 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, the declaration below is for a `View` template:

```
{namespace View}
```

It can be accessed in another Soy template by calling the `render` method on the namespace as shown below. The `data='all'` attribute specifies that the template should include all its parameters as well:

```
{call View.render data="all"}{/call}
```

Note: Template namespaces must be unique.

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` declaration. For instance, the `hello-soy-web` portlet's `View` Soy template defines the `navigationURL` parameter with the code below:

```
@param navigationURL
```

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 Hello Soy portlet main view.
 *
 * @param id
```

```
 * @param layouts
 * @param navigationURL
 */
{template .render}
    <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

CHAPTER 42

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 and watches these properties for changes. Any template parameters defined in your portlet classes are added automatically 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.

42.1 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 returns 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

42.2 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

42.3 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.

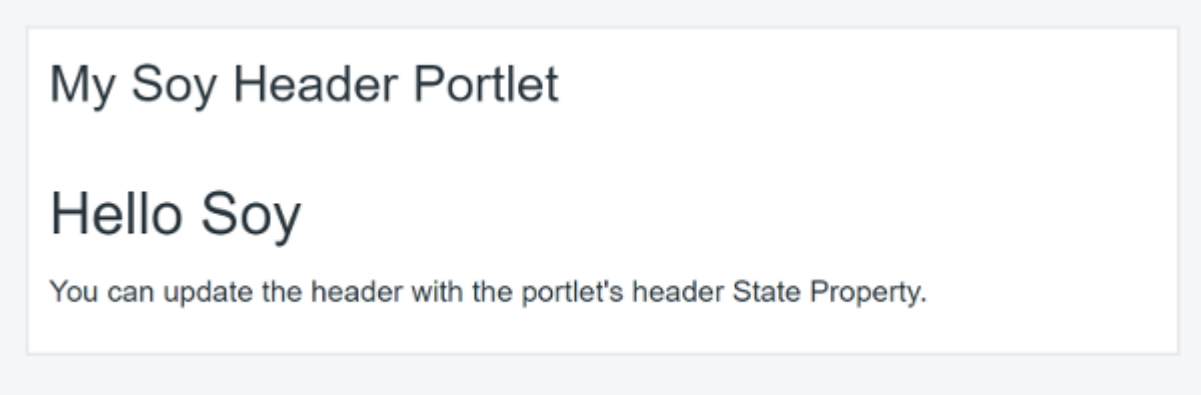


Figure 42.1: The example Soy portlet has a configurable header.

The text in the header following *Hello* is provided by the `header` state property defined in its `*mvcRenderCommand` class:

*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.
 *
 * @param id: string
 * @param header: string
 */
{template .render}

    <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()`. Here's an example configuration:

   ```
   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. The code below retrieves the example 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'});
   ```

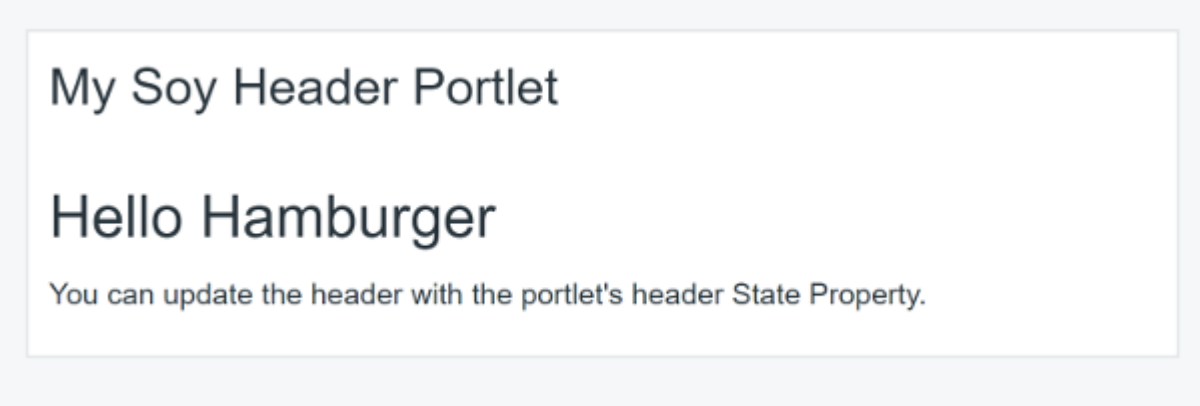


Figure 42.2: You can change the example portlet's header state property on the client side.

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 43

SPRING MVC

Liferay is an open platform in an ecosystem of open platforms. Just because Liferay has its own MVC framework 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?

1. Develop as you normally do using Spring MVC.
2. Configure your application for Liferay.
3. Deploy it to Liferay.

Since you already have your app, you'll start with configuration.

43.1 Configuring a Spring MVC Portlet

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.

Portlet Configuration

In the `portlet.xml` file's `portlet-class` element 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 the `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. Next configure your web application.

Web Application Configuration

If you're letting Liferay generate the WAB for you (this is the recommended approach), the elements are added automatically during auto-deployment.

If you're configuring an OSGi Web Application Bundle (WAB) yourself, the `web.xml` file in your Spring MVC project must be fully ready for deployment. In addition to your Spring MVC configuration, your `web.xml` must include these elements:

- `listener` for `PluginContextListener`
- `servlet` and `servlet-mapping` for `PortletServlet`

The elements look like this:

```
<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>
```

Your application must be able to convert `javax.portlet.PortletRequests` to `javax.servlet.ServletRequests` and back again. Add this to the `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>
```

That's all the configuration that's necessary for `web.xml`. Now you're ready to configure the views.

Views

To configure the Spring view resolver, add a bean to your application context file (`portlet-context.xml` in the previous example):

```
<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, so now it's time to configure controller classes to handle the requests.

Controllers

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.

View Mode Controller

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.

Edit Mode 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

    }
}
```

Make sure to define any controller classes in your application context file by adding a `bean` element 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 must also 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 specifies a new category for the application in Liferay's menu for adding applications:

```
<display>
    <category name="New Category">
        <portlet id="example-portlet" />
    </category>
</display>
```

The descriptor `liferay-portlet.xml` specifies 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 define the Liferay roles used in the portlet.

The `liferay-plugin-package.properties` file describes the Liferay plugin, declares its resources, and specifies its security related parameters. The DTD is 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.

All of Liferay's DTDs are here.

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.

Since you're in the context of a Spring MVC portlet, you can't look up a reference to the services (including Service Builder services) published to the OSGi runtime using Declarative Services. You have to use Service Trackers. There's some boilerplate code involved, but the ability to look up services in the OSGi runtime is worth it.

Next consider how to package and deploy your Spring MVC portlet.

43.2 Deploying a Spring MVC Portlet

Developers creating portlets for Liferay DXP 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 Liferay:

- 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 a `WEB-INF/liferay-plugin-package.properties` file for the WAB.

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 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's Spring Framework JARs and excludes yours from the generated WAB. Understanding Excluded JARs explains how to detect Liferay DXP's Spring Framework version.

Once you've packaged your Spring MVC portlet as a WAR, you can deploy it by copying it to `[LIFERAY_HOME]/deploy`.

Congratulations on deploying your Spring MVC portlet!

CHAPTER 44

JSF PORTLETS WITH LIFERAY FACES

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.

44.1 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` file, 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.

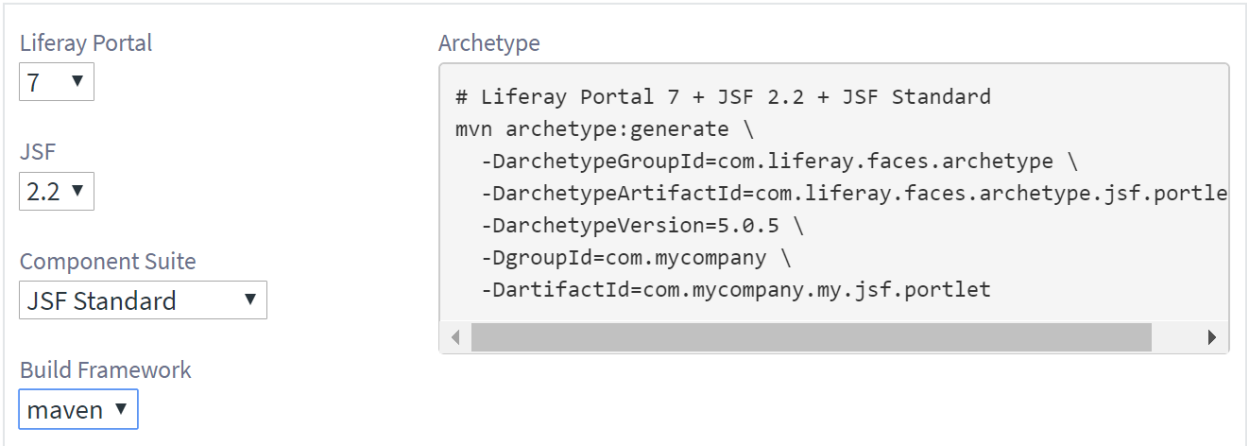


Figure 44.1: You can select the Liferay Portal version, JSF version, and component suite for your archetype generation command.

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 Dev Studio DXP 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:

```
../gradlew 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

44.2 Generating a JSF Project Using Dev Studio

You can generate a Liferay Faces application without having to create your own folder structure and descriptor files manually using Liferay Dev Studio. 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 Dev Studio. Open your Dev Studio 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

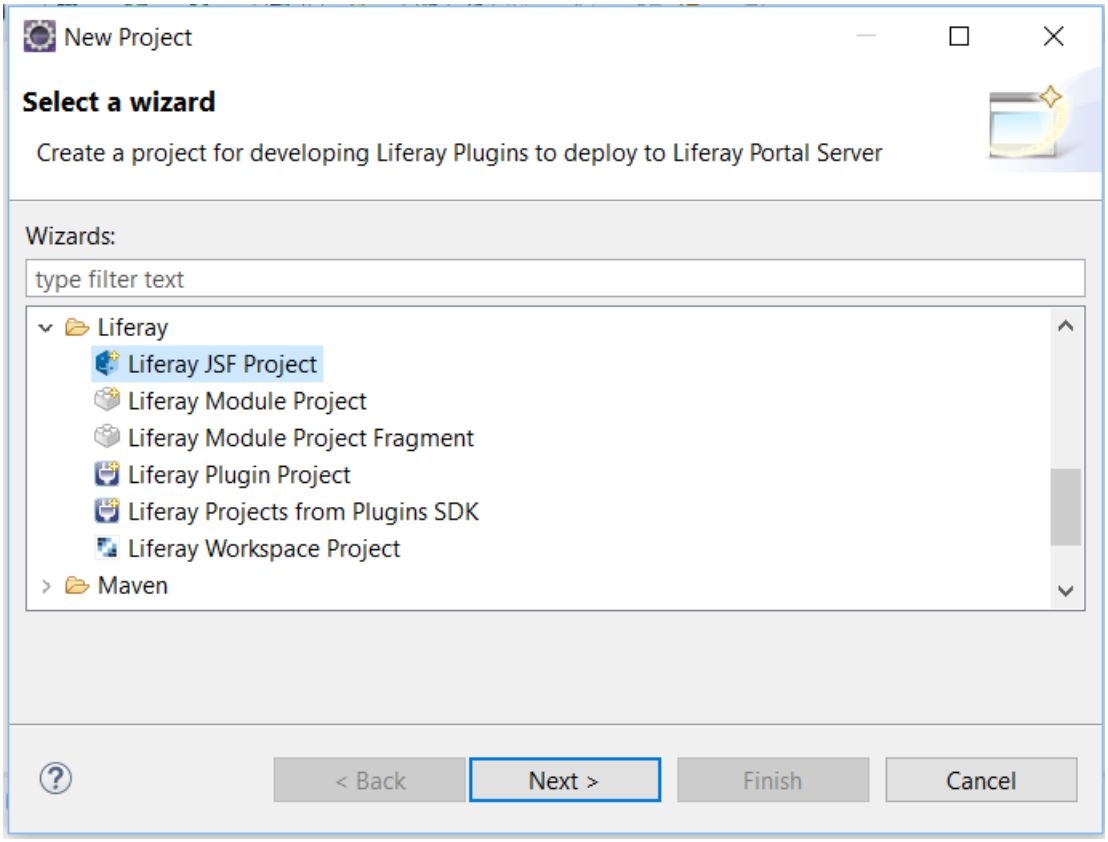


Figure 44.2: Choose the *Liferay JSF Project* option to begin creating a JSF project in Dev Studio.

4. Click *Finish* to generate your Liferay JSF project.

You've generated a Liferay JSF project using Dev Studio! The project you generated contains a simple portlet that you can customize.

Note: There is another option in Dev Studio'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 Dev Studio!

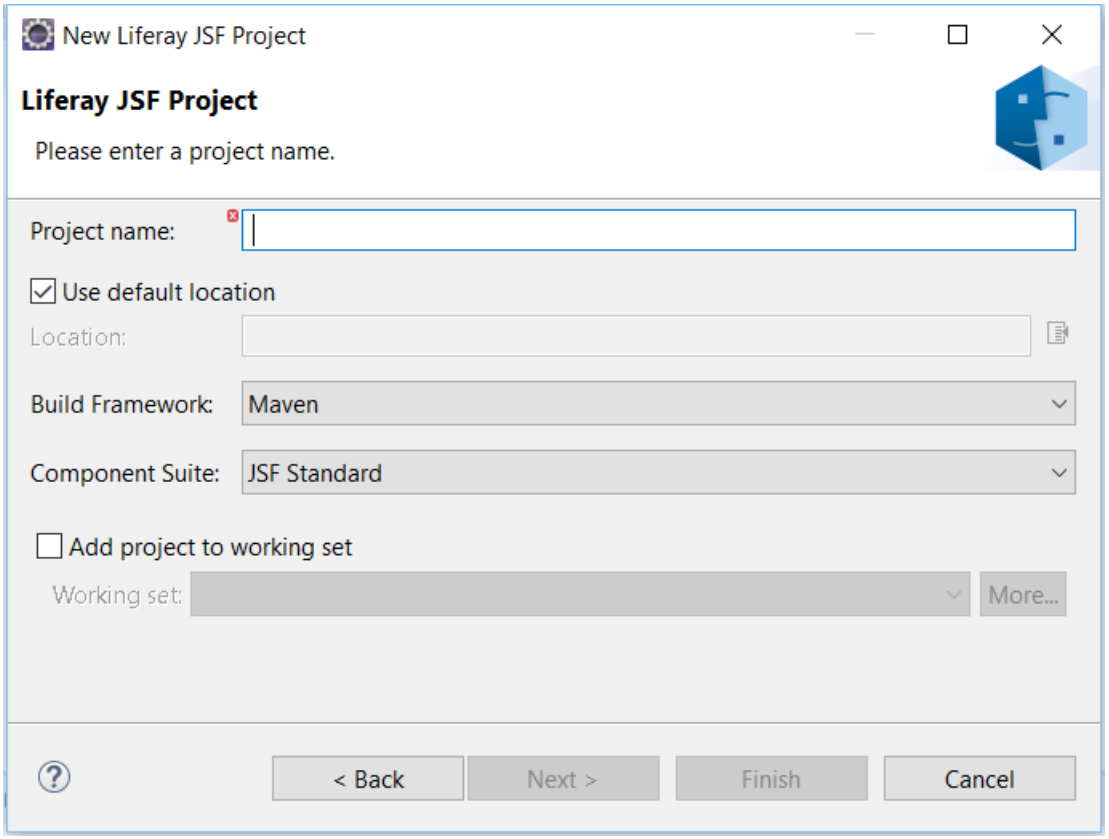


Figure 44.3: Choose your preferred options for your JSF project.

Figure 44.4: The generated JSF portlet project displays basic build information.

CHAPTER 45

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 section of tutorials, you'll step through packaging and creating a JSF application that is deployable as an OSGi module at runtime.

The steps you'll take are outlined below:

- 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 examine the example JSF application solution any time during this section by downloading its ZIP file.

Let's get started!

45.1 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
    * 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.

45.2 Defining a JSF Application's Structure and Dependencies

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>5.0.3</liferay.faces.bridge.ext.version>
        <liferay.faces.bridge.version>4.1.2</liferay.faces.bridge.version>
        <mojarra.version>2.2.18</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 next.

45.3 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'll begin working on your JSF application's resources next.

45.4 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, you'll begin developing the Hello User application's behavior and UI next.

45.5 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.

You can view the finished version of the Hello User JSF application by downloading its ZIP file.

45.6 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.

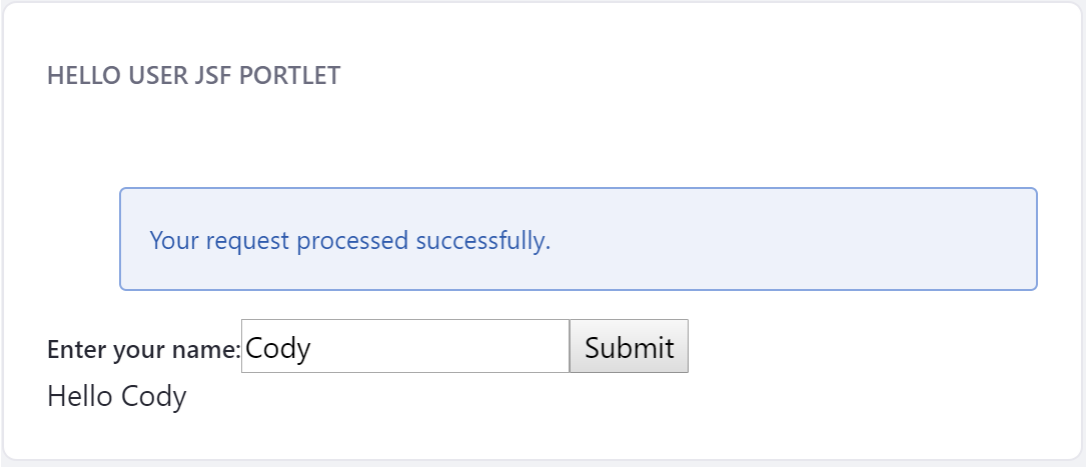


Figure 45.1: After submitting the user's name, it's displayed with a greeting.

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
    Configurable Applications

45.7 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 the parameters he carried. He was happiest when he could tell people he met 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&
```

The other was just, well, friendly. She was less concerned about looking smart and more concerned about those she interacted with, so she shared only the important things about her. 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.

Creating Friendly URL Routes

1. First create a `routes.xml` file in your application's web module. Liferay's pattern puts 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 7.1.0//EN" "http://www.liferay.com/dtd/liferay-friendly-url-
routes_7_1_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 are always the same for the URL. For example, for a render URL, you can be certain that the `p_p_lifecycle` parameter is always `0`. You don't have to define these types of implicit parameters, but it's a best practice because if you don't, they still appear in your URL.

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. This determines the page that's 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
)
```

The DTD file completely defines the `routes.xml` file.

Implementing a Friendly URL Mapper

Once you have your URLs mapped in a `routes.xml` file, you must provide an implementation of the `FriendlyURLMapper` service. Create a component that specifies a `FriendlyURLMapper` service, with two properties:

1. A `com.liferay.portlet.friendly-url-routes` property sets the path to your `routes.xml` file.
2. A `javax.portlet.name` property, which you probably have already, specifies your portlet's name.

```
@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 must only override one method, `getMapping()`. 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 friendly URLs in Blogs begin with the String set here (`blogs`).

Friendly URLs in Action

Let's look at one of these Friendly URLs in action. Add a blog entry and then click on the entry's title. 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>
```

The `urlTitle` is generated from the blog post's title. 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 the render URL 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.

CHAPTER 46

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.

46.1 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:

```
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. Below is an example `package.json` file for a `js-logger` module:

```
{
    "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

46.2 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 your exposed modules in your portlets. Follow the steps below to use your exposed modules in your portlets.

1. Declare the `aui` taglib in your view JSP:

   ```
   <%@ 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.

   The `require` attribute lets you include your exposed modules in your JSP. The AMD Loader fetches the specified module and its dependencies. An example faux Console Logger Portlet's `view.jsp` shown below 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

Customizing JSPs
Web Services

CHAPTER 47

USING NPM IN YOUR PORTLETS

npm is a powerful tool, and almost a necessity for Front-End development. 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.

This section of tutorials covers how to set up npm-based portlet projects.

47.1 Formatting Your npm Modules for AMD

For Liferay DXP to recognize your npm modules, they must be formatted for the Liferay AMD Loader. Luckily, the liferay-npm-bundler handles this for you, you just have to provide the proper configuration and add it to your build script. This tutorial shows how to use the liferay-npm-bundler to set up npm-based portlet projects.

Note: the example and steps covered in this tutorial use the old mode of the liferay-npm-bundler that includes additional build steps in the `build` script before the bundler runs. The new mode of the bundler, like webpack, uses loaders and a set of rules. See Migrating Your Project to Use the New Mode to learn how to migrate your project to use the new mode.

The example structure below is referenced throughout this tutorial. You can download it here if you want to follow along:

- `npm-angular5-portlet-say-hello/`
    - `META-INF/`
        - `resources/`
            - `package.json`
            - name: npm-angular5-portlet-say-hello
            - version: 1.0.0

        - main: js/angular.pre.loader.js
        - scripts:
        - build: tsc && liferay-npm-bundler

        - `tsconfig.json`
        - target: es5
        - moduleResolution: node

        - `.npmbundlerrc`
        - exclude:
        - *: true

        - config:
        - imports:
        - npm-angular5-provider:
        - @angular/animations: ^5.0.0
        - @angular/cdk: ^5.0.0
        - @angular/common: ^5.0.0
        - @angular/compiler: ^5.0.0

        - “” :
        - npm-angular5-provider: ^1.0.0

        - `js/`
        - `indigo-pink.css`
        - `angular.pre.loader.ts` // Bootstrap shims and providers
        - import npm-angular5-provider;

- `npm-angular5-provider`

    - `package.json`

        * name: npm-angular5-provider
        * version: 1.0.0
        * main: bootstrap.js
        * scripts:

            - build: liferay-npm-bundler

        * dependencies:

            - @angular/animations: ^5.0.0
            - @angular/cdk: ^5.0.0
            - @angular/common: ^5.0.0
            - @angular/compiler: ^5.0.0
            - ...

    - `src/main/resources/META-INF/resources/`

        * `bootstrap.js` // This file includes polyfills needed by Angular and must be loaded before the app

- require('core-js/es6/reflect');
- require('core-js/es7/reflect');
- require('zone-js/dist/zone');

Follow these steps to configure your project to use the liferay-npm-bundler:

1. Install NodeJS >= v6.11.0 if you don't have it installed.

2. Navigate to your portlet's project folder and initialize a `package.json` file if it's not present yet.

   If you don't have a portlet already, create an empty MVC portlet project. For convenience, you can use Blade CLI to create an empty portlet with the mvc portlet blade template.

   If you don't have a `package.json` file, you can run `npm init -y` to create an empty one based on the project directory's name.

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.
```

---

4. Add the `liferay-npm-bundler` to your `package.json`'s build script to pack the needed npm packages and transform them to AMD:

   ```
   "scripts": {
         "build": "[... && ] liferay-npm-bundler"
   }
   ```

   The `[...&&]` refers to any previous steps you need to perform (for example, transpiling your sources with Babel, compiling SOY templates, transpiling Typescript, etc.). The example includes the Typescript compiler (`tsc`) in its build script because Angular requires it for transpiling code to ES5:

   ```
   "build": "tsc && liferay-npm-bundler"
   ```

```
**Note:** You can use any languages you like as long as they can be
transpiled to ECMAscript 5 or higher. The only requirements are:

- That Babel can convert them to an AST to be able to process it
- That your browser can execute it.
- That modules are loaded using `require()` calls (this requirement can be
  relaxed by using customized plugins, but is mandatory for the default
  out-of-the-box configuration).

  When you deploy your portlet using Gradle, the build script is called as
  part of the process.
```

5. Configure your project for the bundler, using the `.npmbundlerrc` file (create this file in your project's root folder if it doesn't exist). You can specify packages to exclude from the output JAR, imports for shared dependencies, and more. See the Configuring liferay-npm-bundler reference for more information on the available options.

   The example excludes every dependency (using the wildcard (*) symbol) of the `npm-angular5-portlet-say-hello` widget to prevent Angular from appearing in its JAR, making the build process faster and optimizing deployment. Note that `npm-angular5-provider` is also imported with no namespace ("") because one of its modules is going to be invoked to bootstrap Angular shims: see the `angular.pre.loader.ts` file, where `npm-angular5-provider` is imported. That import, in turn, loads `npm-angular5-provider`'s main file (`bootstrap.js`):

   ```
   {
       ...
       "exclude": {
           "*": true
       },
       "config": {
           "imports": {
               "npm-angular5-provider": {
                   "@angular/animations": "^5.0.0",
                   "@angular/cdk": "^5.0.0",
                   "@angular/common": "^5.0.0",
                   "@angular/compiler": "^5.0.0",
                   ...
               },
               "": {
                   "npm-angular5-provider": "^1.0.0"
               }
           }
       }
   }
   ```

6. Run `npm install` to install the required dependencies.

7. Run the build script to bundle your dependencies with the liferay-npm-bundler:

   ```
   npm run-script build
   ```

The bundler copies the project and `node_modules`' JS files to the output and wraps them inside a `Liferay.Loader.define()` call so that the Liferay AMD Loader knows how to handle them. It also namespaces the module names in `require()` calls and inside the `Liferay.Loader.define()` call with the project's name prefix (`npm-angular5-provider$` in the example) to achieve dependency isolation. the bundler injects the dependencies in the `package.json` pertaining to `npm-angular5-provider` to make them available at runtime. The resulting build for the example widget is shown below:

- `npm-angular5-portlet-say-hello/`
  - `build/`
    - `resources/main/META-INF/resources`
      - `package.json`

            - dependencies:
            - @npm-angular5-provider$angular/animations: ^5.0.0
            - @npm-angular5-provider$angular/cdk: ^5.0.0
            - @npm-angular5-provider$angular/common: ^5.0.0
            - @npm-angular5-provider$angular/compiler: ^5.0.0

            - `js/`
            - `angular.loader.js`
            - Liferay.Loader.define("npm-angular5-portlet-say-hello@1.0.0/js/angular.loader"
            - ['module', 'exports', 'require', '@npm-angular5-provider$angular/platform-browser-dynamic', ...]

- `npm-angular5-provider`

    - `package.json`

        * name: npm-angular5-provider
        * version: 1.0.0
        * main: bootstrap.js
        * dependencies:

            - @npm-angular5-provider$angular/animations: ^5.0.0
            - @npm-angular5-provider$angular/cdk: ^5.0.0
            - @npm-angular5-provider$angular/common: ^5.0.0
            - @npm-angular5-provider$angular/compiler: ^5.0.0
            - ...

    - `bootstrap.js`

        * Liferay.Loader.define('npm-angular5-provider@1.0.0/bootstrap'
        * ['module', 'exports', 'require', 'npm-angular5-provider$core-js/es6/reflect', 'npm-angular5-provider$zone.js/dist/zone']

    - `src/main/resources/META-INF/resources/`

        * `bootstrap.js` // This file includes polyfills needed by Angular and must be loaded before the app

            - require('core-js/es6/reflect');
            - require('core-js/es7/reflect');
            - require('zone-js/dist/zone');

Note: By default, the AMD Loader times out in seven seconds. Since Liferay DXP Fix Pack 3 and Liferay Portal 7.1 CE GA 2, you can configure this value through System Settings. Open the Control Panel and navigate to *Configuration → System Settings → PLATFORM → Infrastructure*, and select *JavaScript Loader*. Set the *Module Definition Timeout* configuration to the time you want and click *Save*.

Now you know how to use the liferay-npm-bundler to bundle your npm-based portlets for the Liferay AMD Loader!

Related Topics

Preparing Your JavaScript Files for ES2015+

47.2 Migrating a liferay-npm-bundler Project from 1.x to 2.x

You should use the latest 2.x version of the liferay-npm-bundler. It offers more stability and includes more features out-of-the-box. If you already created a project using the 1.x version, don't worry. Follow these steps to migrate your project to 2.x:

1. Update the `liferay-npm-bundler` dependency in your `package.json` to version 2.x:

```
{
  "devDependencies": {
    ...
    "liferay-npm-bundler": "^2.0.0",
    ...
  },
  ...
}
```

2. Remove all `liferay-npm-bundler-preset-*` dependencies from your `package.json` because liferay-npm-bundler 2.x includes these by default.

3. Remove any bundler presets you configured in your `.npmbundlerrc` file. liferay-npm-bundler 2.x includes one smart preset that handles all frameworks automatically.

These are the standard requirements that all projects have in common. The remaining steps depend on your project's framework. Follow the instructions in the corresponding section to finish migrating your project.

Migrating a Plain JavaScript, Billboard JS, JQuery, Metal JS, React, or Vue JS Project

After following the steps covered in the beginning, follow these remaining steps to migrate the framework projects shown below to 2.x:

- plain JS project
- Billboard.js project
- JQuery project
- Metal.js project
- React project
- Vue.js project

While Babel is required to transpile your source files, you must remove any Babel preset used for transformations from your project that bundler 1.x imposed. liferay-npm-bundler 2.x handles these transformations by default:

1. Remove the *liferay-project* preset from your project's `.babelrc` file. All that should remain is the `es2015` preset shown below:

```
{
  "presets": ["es2015"]
}
```

If your project uses React, make sure the `react` preset remains as well:

```
{
  "presets": ["es2015", "react"]
}
```

2. Remove the `babel-preset-liferay-project` dependency from your `package.json`.

If you're migrating an Angular project, follow the steps in the next section.

Migrating an Angular Project

After following the steps covered in the beginning, follow these remaining steps to migrate your Angular project to 2.x. While liferay-npm-bundler 1.x relied on Babel to perform some transformation steps, these transformations are now automatically applied in version 2.x. Therefore, you should remove Babel from your project:

1. Open your `tsconfig.json` file and replace the `"module": "amd"` compiler option with the configuration shown below to produce CommonJS modules:

```
{
  "compilerOptions": {
    ...
    "module": "commonjs",
    ...
  }
}
```

2. Delete the `.babelrc` file to remove the Babel configuration.

3. Remove Babel from your `package.json` build process so it matches the configuration below:

```
{
  "scripts": {
    "build": "tsc && liferay-npm-bundler"
  },
  ...
}
```

4. Remove the following Babel dependencies from your `package.json` *devDependencies*:

```
"babel-cli": "6.26.0",
"babel-preset-liferay-amd": "1.2.2"
```

Related Topics

Formatting Your npm Modules for AMD
Using the NPMResolver API in Your Portlets
What Changed between liferay-npm-bundler 1.x and 2.x

47.3 Migrating Your Project to Use liferay-npm-bundler's New Mode

In the previous version of the liferay-npm-bundler, before the bundler ran, the build did some preprocessing, then the bundler modified the output from the preprocessed files, as shown in the example build script below:

```
{
  "scripts":{
    "build": "babel --source-maps -d build src && liferay-npm-bundler"
  }
}
```

In the new mode, the liferay-npm-bundler is in charge of the whole process, like webpack, and configured via a set of rules. The build script is condensed, as shown below:

```
{
  "scripts":{
    "build": "liferay-npm-bundler"
  }
}
```

Follow these steps to migrate your project to use the new configuration mode:

1. Open the project's `package.json` file and update the `build` script to only use the liferay-npm-bundler:

   ```
   {
     "scripts":{
       "build": "liferay-npm-bundler"
     }
   }
   ```

2. Define the rules for the bundler to use (e.g. running babel to transpile files) in the project's `.npmbundlerrc` file. The example configuration below defines rules for using the `babel-loader` to transpile JavaScript files. See the Default Loaders reference for the full list of default loaders. Follow the steps in Creating Custom Loaders for the Bundler to create a custom loader. The liferay-npm-bundler processes the `*.js` files in `/src/` with babel and writes the results in the default `/build/` folder:

   ```
   {
     "sources": ["src"],
     "rules": [
       {
         "test": "\\.js$",
         "exclude": "node_modules",
         "use": [
           {
             "loader": "babel-loader",
             "options": {
               "presets": ["env"]
             }
           }
         ]
       }
     ]
   }
   ```

```
**Note:** The new mode of the liferay-npm-bundler acts very much
 like webpack, but because webpack creates a single JS bundle file and
 liferay-npm-bundler targets an AMD loader, they are not compatible.
```

Related Topics

- Default liferay-npm-bundler Loaders
- Understanding liferay-npm-bundler's Loaders

47.4 Creating Custom Loaders for the liferay-npm-bundler

Since webpack creates JS bundles and the liferay-npm-bundler targets AMD loader, webpack's loaders aren't compatible with the liferay-npm-bundler. So, if you want to use a loader that isn't available by default, you must create a custom loader.

A loader, in terms of the liferay-npm-bundler, is defined as an npm package that has a main module which exports a default function with this signature:

```
function(context, options){
}
```

The arguments are defined as follows:

- `context`: an object that contains these fields
  - `content`: A string with the contents of the processed file (the main input of the loader)
  - `filepath`: the project-relative path to the file to process with the loader
  - `extraArtifacts`: an object with project-relative paths as keys and strings as values of properties which may be used to output extra files along with the one being processed (for example, you can use it to generate source maps).
  - `log`: a logger that writes execution information to the bundler's report file (see the PluginLogger class for information on its structure and API).
- `options`: an object taken from the `options` field of the loader's configuration (See Understanding liferay-npm-bundler's loaders and rules for more information).

Note: the function may return nothing or modified content. If something is returned, it is copied on top of the `context.content` field ans used to feed the next loader or write the output file. This is the equivalent to `context.content` = 'something'. If your loader does not return a file, but instead it only filters files to prevent them from being generated, you must explicitly set `context.content = 'undefined'`.

Follow these steps to write a new loader. These steps use the babel loader as an example:

1. If your loader requires configuration, like babel, you may define a rule configuration like the one shown below so you can specify options for the loader:

```
{
  "rules": [
    {
      "test": "\\.js$",
      "exclude": "node_modules",
      "use": [
        {
          "loader": "babel-loader",
          "options": {
            "presets": ["env", "react"]
          }
        }
      ]
    }
  ]
}
```

2. Create an `index.js` file and write a function that takes the input content, passes it through the loader, and writes the result and the source map file to the output folder. The loader function below takes the passed content (JS files), run it through babel, and writes the result and source map to the default `/build/` output folder:

```
export default function(context, options) {
  // Get input parameters
  const { content, filePath, log, sourceMap } = context;

  // Run babel on content
  const result = babel.transform(content, options);

  // Create an extra .map file with source map next to source .js file
  context.extraArtifacts[`${filePath}.map`] = JSON.stringify(result.map);

  // Tell the user what we have done
  log.info("babel-loader", "Transpiled file");

  // Return the modified content
  return result.code;
}
```

3. Place the `index.js` file in an npm package and publish it.

4. Include the npm package you just created as a devDependency in the project's `package.json`:

```
"devDependencies": {
  "liferay-npm-bundler": "2.12.0",
  "liferay-npm-build-support": "2.12.0",
  "liferay-npm-bundler-loader-babel-loader": "2.12.0",
  ...
}
```

5. Configure the loader's name in the `rules` section of the project's `.npmbundlerrc` file:

```
{
  "sources": ["src"],
  ...
  "rules": [
    {
      "test": "\\.js$",
      "exclude": "node_modules",
      "use": [
        {
```

```
          "loader": "babel-loader",
          "options": {
            "presets": ["env", "react"]
          }
        }
      ]
    }
  ],
  ...
}
```

Related Topics

- Default liferay-npm-bundler Loaders
- Understanding liferay-npm-bundler's Loaders

CHAPTER 48

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. The `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.

48.1 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

48.2 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

Referencing an npm Module's Package

The Structure of OSGi Bundles Containing npm Packages

How Liferay DXP Publishes npm Packages

CHAPTER 49

APPLYING CLAY STYLES TO YOUR APP

It's important to have a consistent user experience across your apps. Portal's built-in apps achieve this through Liferay's Lexicon Experience Language and its web implementation, Clay.

Clay provides a consistent, user-friendly UI and is included in all themes that are based on the `_styled` base theme, making all the components documented on the Clay site accessible.

This means you can use Clay markup and components in your apps. These tutorials explain how to apply Clay's design patterns to achieve the same look and feel as Portal's built-in apps.

The tutorials in this section cover the following topics:

- Applying Clay to navigation
- Implementing the Management Toolbar

49.1 Applying Clay Patterns to Navigation

This tutorial covers how to leverage Clay patterns in your app's navigation to make it more user-friendly. Updating your app's navigation bar to use Clay is easy, thanks to the `<clay:navigation-bar />` tag. Follow these steps to update your app:

1. Add the required imports to your app's `init.jsp`:

```
// Import the clay tld file to be able to use the new tag
<%@ taglib uri="http://liferay.com/tld/clay" prefix="clay" %>

// Import the NavigationItem utility class to create the items model
<%@ page import="com.liferay.frontend.taglib.clay.servlet.taglib.util.JSPNavigationItemList" %>
```

2. Add the `frontend-taglib-clay` and `frontend.taglib.soy` module dependencies to your app's `build.gradle` file:

```
compileOnly group: "com.liferay", name: "com.liferay.frontend.taglib.soy",
version: "1.0.10"

compileOnly group: "com.liferay", name: "com.liferay.frontend.taglib.clay",
version: "1.0.0"
```

3. Inside your JSP view, add a java scriplet to retrieve the navigation variable and portlet URL. An example configuration is shown below:

```
<%
final String navigation = ParamUtil.getString(request, "navigation",
"entries");

PortletURL portletURL = renderResponse.createRenderURL();

portletURL.setParameter("mvcRenderCommandName", "/blogs/view");
portletURL.setParameter("navigation", navigation);
%>
```

4. Add the `<clay:navigation-bar />` tag to your app, and use the `items` attribute to specify the navigation items. The navigation bar should be dark if your app is intended for Admin use. To do this, set the `inverted` attribute to `true`. If your app is intended for an instance on a live site, keep the navigation bar light by setting the `inverted` attribute to `false`. An example configuration for an admin app is shown below:

```
<clay:navigation-bar
    inverted="<%= true %>"
    navigationItems="<%=
        new JSPNavigationItemList(pageContext) {
            {
                add(
                navigationItem -> {
                    navigationItem.setActive(navigation.equals("entries"));
                    navigationItem.setHref(renderResponse.createRenderURL());
                    navigationItem.setLabel(LanguageUtil.get(request, "entries"));
                });

                add(
                navigationItem -> {
                    navigationItem.setActive(navigation.equals("images"));
                    navigationItem.setHref(renderResponse.createRenderURL(),
          "navigation", "images");
                    navigationItem.setLabel(LanguageUtil.get(request, "images"));
                });
            }
        }
    %>"
/>
```

5. Add a conditional block to display the proper JSP for the selected navigation item. An example configuration for the Blogs Admin portlet is shown below:

```
<c:choose>
    <c:when test='<%= navigation.equals("entries") %>'>
        <liferay-util:include page="/blogs_admin/view_entries.jsp"
    servletContext="<%= application %>" />
    </c:when>
    <c:otherwise>
        <liferay-util:include page="/blogs_admin/view_images.jsp"
    servletContext="<%= application %>" />
    </c:otherwise>
</c:choose>
```

Live site navigation bar:
Admin app navigation bar:
Sweet! Now you know how to style a navigation bar with Clay.

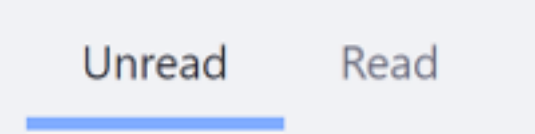


Figure 49.1: The navigation bar should be light for apps on the live site.

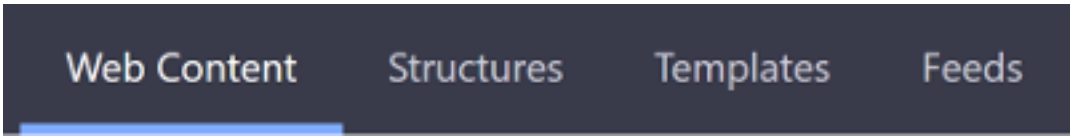


Figure 49.2: The navigation bar should be dark (inverted) in admin apps.

Related topics

Implementing the Management Toolbar

49.2 Configuring Your Application's Title and Back Link

For 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 49.3: Adding a new blog entry displays the portlet title at the top, along with a back link.

The Blogs Admin application is used as an example throughout this article. 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. The example below 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.

```
renderResponse.setTitle((entry != null) ? entry.getTitle() :
LanguageUtil.get(request, "new-blog-entry"));
%>
```

4. Update any back links in the view to use the `redirect`. The Blog Admin app's `edit_entry.jsp` form's cancel button is shown below as an example:

```
<aui:button cssClass="btn-lg" href="<%= redirect %>" name="cancelButton"
type="cancel" />
```

Great! Now you know how to configure your app's title and back URL.

Related Topics

- Applying Clay Patterns to Your Navigation Bar
- Setting Empty Results Messages

49.3 Setting Empty Results Messages

If you've toured the UI, you've probably noticed messages or possibly even animations in the search containers when no results are found.

No web content was found.

Figure 49.4: This is a still frame from the Web Content portlet's empty results animation.

You can configure your apps to use empty results messages and animations too, thanks to the `liferay-frontend:empty-results-message` tag.

Follow these steps:

1. Add the `liferay-frontend` taglib declaration into your portlet's `init.jsp`:

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

2. Add an `empty-result-message` tag to your portlet's view:

   ```
   <liferay-frontend:empty-result-message
   />
   ```

3. Configure the tag's attributes to define your search container's empty results message, with or without an animation or image. The attributes are described in the table below:

```
 Attribute | Description |
 --- | --- |
 `actionDropdownItems` | Specifies the action or actions to display for the empty results in either a dropdown menu, a link, or a button, depending o
 `animationType` | The CSS class for the animation. Four values are available by default with these CSS classes: `EmptyResultMessageKeys.AnimationTyp
empty-state`), `EmptyResultMessageKeys.AnimationType.SEARCH` (`taglib-search-state`), `EmptyResultMessageKeys.AnimationType.SUCCESS` (`taglib-
success-state`), and `EmptyResultMessageKeys.AnimationType.NONE`. You can also specify a custom CSS class if you prefer. |
 `componentId` | Specifies the ID for the `actionDropdownItems` component (dropdown menu, link, or button)|
 `description` | The descriptive text to display beneath the main message. |
 `elementType` | The type of element to replace the `x` parameter in the main message's language key `no-x-yet`. |
```

An example configuration is shown below:

```
```html
<liferay-frontend:empty-result-message
  actionDropdownItems="<%= FragmentPermission.contains(permissionChecker, scopeGroupId,
  FragmentActionKeys.MANAGE_FRAGMENT_ENTRIES) ? fragmentDisplayContext.getActionDropdownItems() : null %>"
  animationType="<%= EmptyResultMessageKeys.AnimationType.NONE %>"
  componentId="actionsComponent"
  description='<%= LanguageUtil.get(request, "collections-are-needed-to-create-fragments") %>'
  elementType='<%= LanguageUtil.get(request, "collections") %>'
/>
```
```

```
 **Note:** You can replace the available default animations with your own
 by overriding the `taglib-empty-state`, `taglib-search-state`, and
 `taglib-success-state` CSS classes provided by
 [_empty_result_message.scss](https://github.com/liferay/liferay-portal/blob/7.1.x/modules/apps/frontend-css/frontend-css-
web/src/main/resources/META-INF/resources/taglib/_empty_result_message.scss),
 or by replacing the existing animations in the `@theme_image_path@/states/`
 folder. Alternatively, you can also provide a new CSS class that defines
 the animation and use it for the `animationType` attribute's value.
```

Figure 49.5: Use the empty state animation to signify there are no entries to search.

Figure 49.6: Use the search state animation to signify no search results were found.

Figure 49.7: Use the success state animation to signify search results were found.

---

empty_state.gif:
search_state.gif:
success_state.gif:

Note: Empty results messages can also contain static images if you prefer. Just use a valid image type instead. All animations must be of type `GIF` though.

Great! Now you know how to configure your app to display an empty results message.

Related Topics

- Using the Liferay Front-End Taglib
- Applying the Add Button Pattern

CHAPTER 50

IMPLEMENTING THE MANAGEMENT TOOLBAR

The Management Toolbar is a combination of search, filters, sorting options, and display options that let you manage data. For admin apps, we recommend that you add a management toolbar to manage your search container results. The Clay Management Toolbar tutorial covers how to use the Clay taglibs to create the Management Toolbar. This section of tutorials covers how to create the features below for the Management Toolbar:

- Implementing View Types
- Sorting and Filtering Items

50.1 Implementing the View Types

The Management Toolbar has three predefined view types for your app's search container results. Each style offers a slightly different look and feel. To provide these view types in your app, you must make some updates to your search result columns. Start by defining the view types you want to provide.

Defining the View Types

The Management Toolbar has three view types:

- **Cards:** displays search result columns on a horizontal or vertical card
- **List:** displays a detailed description along with summarized details for the search result columns
- **Table:** the default view, which list the search result columns from left to right

Follow these steps to define the view types for your management toolbar:

1. Import the `ViewTypeItemList` utility class to create the action items model:

```
<%@ page import="com.liferay.frontend.taglib.clay.servlet.taglib.util.JSPViewTypeItemList" %>
```

2. Add the `frontend.taglib.clay` and `frontend.taglib.soy` module dependencies to your app's `build.gradle` file:

```
compileOnly group: "com.liferay", name: "com.liferay.frontend.taglib.soy",
version: "1.0.10"

compileOnly group: "com.liferay", name: "com.liferay.frontend.taglib.clay",
version: "1.0.0"
```

3. In your app's main view, retrieve the `displayStyle` for reference. Each view type corresponds to a display style. this is used to determine the proper content configuration to display for the selected view type:

```
<%
String displayStyle = ParamUtil.getString(request, "displayStyle");
%>
```

4. Add the management toolbar to your app's main view and configure the display buttons as shown below. Note that while this example implements all three view types, only one view type is required. The default or active view type is set by adding `viewTypeItem.setActive(true)` to the view type:

```
<clay:management-toolbar
    disabled=<%= assetTagsDisplayContext.isDisabledTagsManagementBar() %>
    namespace="<%= renderResponse.getNamespace() %>"
    searchContainerId="assetTags"
    selectable="<%= true %>"
    viewTypes="<%=
        new JSPViewTypeItemList(pageContext, baseURL, selectedType) {
            {
                addCardViewTypeItem(
                    viewTypeItem -> {
                        viewTypeItem.setActive(true);
                        viewTypeItem.setLabel("Card");
                    });

                addListViewTypeItem(
                    viewTypeItem -> {
                        viewTypeItem.setLabel("List");
                    });

                addTableViewTypeItem(
                    viewTypeItem -> {
                        viewTypeItem.setLabel("Table");
                    });
            }
        }
    %>"
/>
```

   `viewTypes`: The available view types

   `portletURL`: The current URL, with the view type parameter included.

5. Create a conditional block to check for the view types. If you only have one view type, you can skip this step.

```
<c:choose>
    <%-- view type configuration goes here --%>
</c:choose>
```

Now that the view types are defined, you can configure them.

Implementing the Card View

The card view displays the entry's information in a vertical or horizontal card with an image, user profile, or an icon representing the content's type, along with details about the content, such as its name, workflow status, and a condensed description.

See the Liferay Frontend Cards tutorial for examples and use cases of each card.

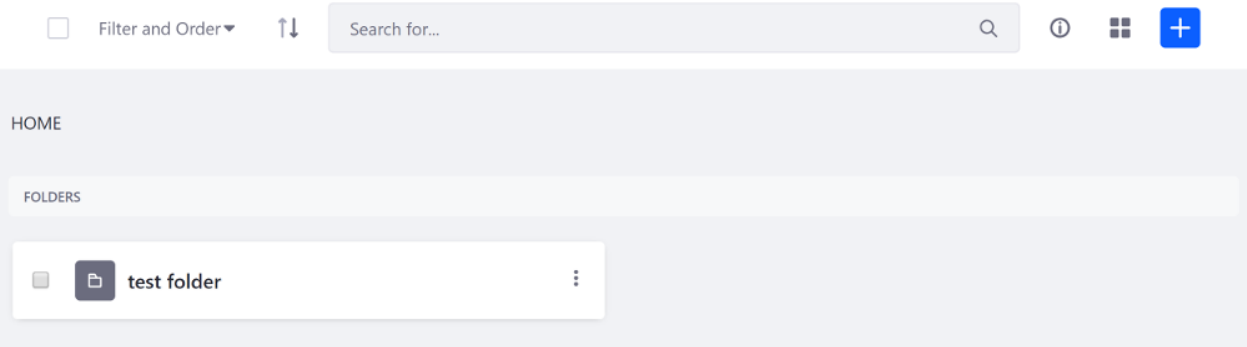


Figure 50.1: The Management Toolbar's card view gives a quick summary of the content's description and status.

Follow the steps below to create your card view:

1. Inside the `<c:choose>` conditional block, add a condition for the icon display style (Card view type):

```
<c:when test='<%= Objects.equals(displayStyle, "icon") %>'>
    <%-- card view type configuration goes here --%>
</c:when>
```

2. Add the proper java scriplet to make the card view responsive to different devices:

   Use the pattern below for vertical cards:

```
<%
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");
%>
```

3. Add the search container column text containing your card. The card should include the actions for the entry(if applicable), as well as an image, icon or user profile, and the entry's title. An example configuration is shown below:

```
<liferay-frontend:icon-vertical-card
  actionJsp='<%= dlPortletInstanceSettingsHelper.isShowActions() ?
  "/image_gallery_display/image_action.jsp" : StringPool.BLANK %>'
  actionJspServletContext="<%= application %>"
  cssClass="entry-display-style"
```

```
        icon="documents-and-media"
        resultRow="<%= row %>"
        title="<%= dlPortletInstanceSettingsHelper.isShowActions() ?
        fileEntry.getTitle() : StringPool.BLANK %>"
    />
```

Implementing the List View

The list view displays the entry's complete description, along with a small icon for the content type, and its name.

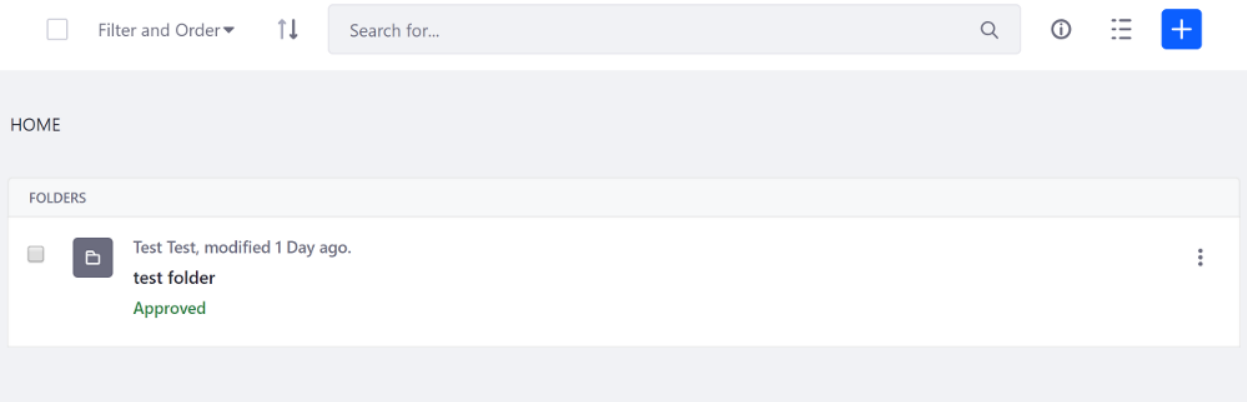


Figure 50.2: The Management Toolbar's list view gives the content's full description.

Inside the `<c:choose>` conditional block, add a condition for the descriptive display style (list view type):

```
<c:when test='<%= Objects.equals(displayStyle, "descriptive") %>'>
    <%-- list view type configuration goes here --%>
</c:when>
```

Your list view should have three columns with the content shown in the table below:

Column | Content Options | Example 1 | icon | <liferay-ui:search-container-column-icon/> | image | <liferay-ui:search-container-column-image/> | user portrait | <liferay-ui:search-container-column-text> <liferay-ui:user-portrait/></liferay-ui:search-container-column-text> 2 | description | <liferay-ui:search-container-column-text colspan="<%=2%>" > <h5><%= userGroup.getName() %></h5> <h6 class="text-default"> <%= userGroup.getDescription() %> </h6> <h6 class="text-default"> " key="x-users"/> </h6> 3 | actions | <liferay-ui:search-container-column-jsp path="/edit_team_assignments_user_groups_action.jsp"/>

Implementing the Table View

The table view list the search container columns from left to right.

Inside the `<c:choose>` conditional block, add a condition for the list display style (table view type):

```
<c:when test='<%= Objects.equals(displayStyle, "list") %>'>
    <%-- table view type configuration goes here --%>
</c:when>
```

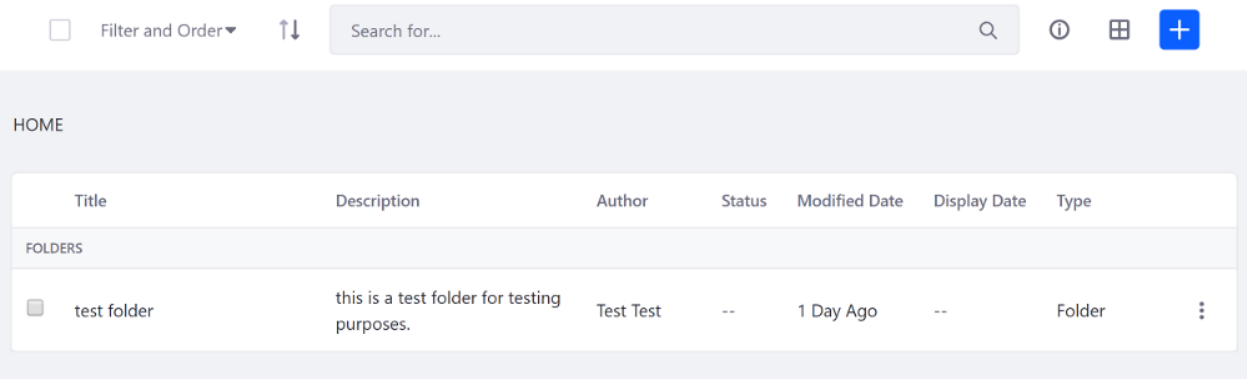


Figure 50.3: The Management Toolbar's table view list the content's information in individual columns.

The columns should at least contain the information shown in the table below:

Column | Content Options | Example 1 | name | <liferay-ui:search-container-column-text cssClass="content-column name-column title-column" name="name" truncate="<%= true %>" value="<%= rule.getName(locale) %>" /> 2 | description | <liferay-ui:search-container-column-text cssClass="content-column description-column" name="description" truncate="<%= true %>" value="<%= rule.getDescription(locale) %>" /> 3 | create date | <liferay-ui:search-container-column-date cssClass="create-date-column text-column" name="create-date" property="createDate" /> 4 | actions | <liferay-ui:search-container-column-jsp cssClass="entry-action-column" path="/rule_actions.jsp" />

Updating the Search Iterator

Once the view type's display styles are defined, you must update the search iterator to show the selected view type. If your management toolbar only has one view type, you can set the `displayStyle` attribute to the style you defined, otherwise follow the pattern below:

```
<liferay-ui:search-iterator
    displayStyle="<%= displayStyle %>"
    markupView="lexicon"
    searchContainer="<%= searchContainer %>"
/>
```

The `displayStyle`'s value is set to the current view type.

Related Topics

Configuring the Clay Management Toolbar Taglib

Filtering and Sorting Items with the Management Toolbar

50.2 Filtering and Sorting Items with the Management Toolbar

The Management Toolbar lets you filter and sort your search container results. While you can configure the toolbar's filters in the JSP, this can quickly crowd the JSP. We recommend instead

that you move this functionality to a separate java class, which we refer to as a Display Context throughout this tutorial.

There are two main types of filters: navigation and order. Both of these are contained within the same dropdown menu. Follow the steps below to create your filters.

1. Depending on your needs, there are two classes that you can extend for your management toolbar Display Context. These base classes provide the required methods to create your navigation and order filters:

    - `BaseManagementToolbarDisplayContext`: for apps without a search container
    - `SearchContainerManagementToolbarDisplayContext`: for apps with a search container (extends `BaseManagementToolbarDisplayContext` and provides additional logic for search containers)

    An example configuration for each is shown below:

    `BaseManagementToolbarDisplayContext` example:

    ```
    public class MyManagementToolbarDisplayContext
      extends BaseManagementToolbarDisplayContext {

      public MyManagementToolbarDisplayContext(
        LiferayPortletRequest liferayPortletRequest,
        LiferayPortletResponse liferayPortletResponse,
        HttpServletRequest request) {

        super(liferayPortletRequest, liferayPortletResponse, request);
      }
      ...
    }
    ```

    `SearchContainerManagementToolbarDisplayContext` example:

    ```
    public class MyManagementToolbarDisplayContext
      extends SearchContainerManagementToolbarDisplayContext {

      public MyManagementToolbarDisplayContext(
        LiferayPortletRequest liferayPortletRequest,
        LiferayPortletResponse liferayPortletResponse,
        HttpServletRequest request, SearchContainer searchContainer) {

        super(
          liferayPortletRequest, liferayPortletResponse, request,
          searchContainer);
      }
    }
    ```

2. Override the `getNavigationKeys()` method to return the navigation filter dropdown item(s). If your app doesn't require any navigation filters, you can just provide the *all* filter to display everything. An example configuration is shown below:

    ```
    @Override
    protected String[] getNavigationKeys() {
      return new String[] {"all", "pending", "done"};
    }
    ```

3. override the `getOrderByKeys()` method to return the columns to order. An example configuration is shown below:

```
@Override
protected String[] getOrderByKeys() {
  return new String[] {"name", "items", "status"};
}
```

4. Open the JSP view that contains the Clay Management Toolbar and set its `displayContext` attribute to the Display Context you created. An example configuration is shown below:

```
<clay:management-toolbar
    displayContext="<%= myManagementToolbarDisplayContext %>"
/>
```

Now you know how to configure the Management Toolbar's filters via a Display Context.

Related Topics

Configuring Filtering and Sorting Management Toolbar Attributes
Implementing the View Types

50.3 Applying the Add Button Pattern

Clay'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. Follow these steps to add a plus button to your app:

1. Add the `liferay-frontend` taglib declaration to your portlet's `init.jsp`:

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

2. Add an `add-menu` tag to your portlet's view:

```
<liferay-frontend:add-menu>
</liferay-frontend:add-menu>
```

3. Nest 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>
```

If there's only one item, the plus icon acts as a button that triggers the item. If there's multiple items, clicking the plus icon displays a menu containing them.

The `com.liferay.mobile.device.rules.web` module's add menu is shown below:

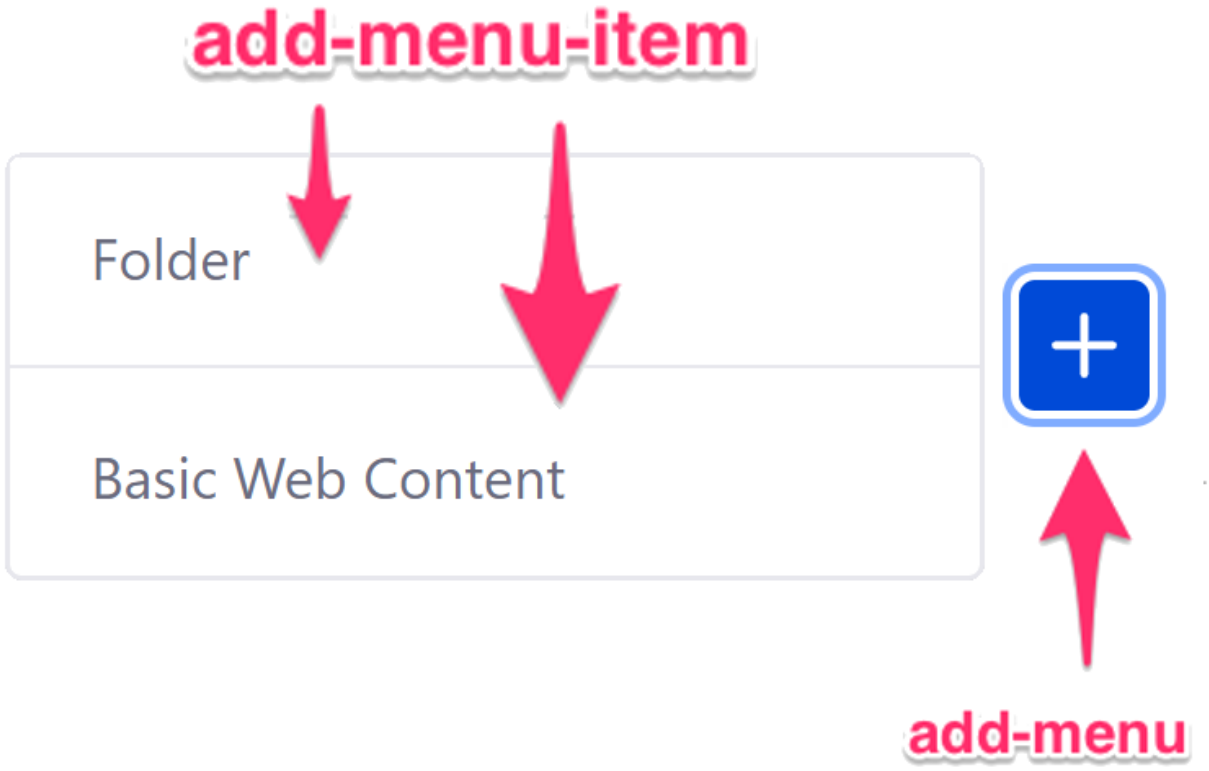


Figure 50.4: The add button pattern consists of an `add-menu` tag and at least one `add-menu-item` tag.

```
<liferay-frontend:add-menu
  inline="<%= true %>"
>
  <liferay-frontend:add-menu-item
    title='<%= LanguageUtil.get(resourceBundle, "add-device-family") %>'
    url="<%= addRuleGroupURL %>"
  />
</liferay-frontend:add-menu>
```

There you have it! Now you know how to use the add button pattern.

Related Topics

- Setting Empty Results Messages
- Implementing the Management Toolbar

50.4 Configuring Your Admin App's Actions Menu

Rather then have a series of buttons or menus with actions in the different views of the app, you can 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. For example, the web content application has the actions menu shown below:

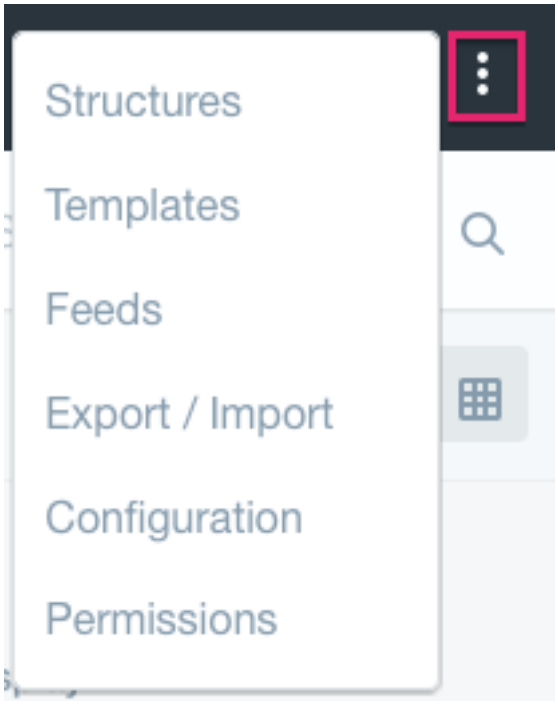


Figure 50.5: The upper right ellipsis menu contains most of the actions for the app.

Follow these steps to configure the actions menu in your admin app:

1. Create a `*ConfigurationIcon` Component class for the action that extends the `BasePortletConfigurationIcon` class and implements the `PortletConfigurationIcon` service:

```
@Component(
    immediate = true,
    service = PortletConfigurationIcon.class
)
public class MyConfigurationIcon extends BasePortletConfigurationIcon {
  ...
}
```

2. Override the `getMessage()` method to specify the action's label:

```
@Override
public String getMessage(PortletRequest portletRequest) {
  ThemeDisplay themeDisplay = (ThemeDisplay)portletRequest.getAttribute(
    WebKeys.THEME_DISPLAY);

  ResourceBundle resourceBundle = ResourceBundleUtil.getBundle(
    themeDisplay.getLocale(), ExportAllConfigurationIcon.class);

  return LanguageUtil.get(resourceBundle, "export-all-settings");
}
```

3. Override the `get()` method to specify whether the action is invoked with the `GET` or `POST` method:

```
@Override
public String getMethod() {
    return "GET";
}
```

4. Override the `getURL()` method to specify the URL (or `onClick` JavaScript method) to invoke when the action is clicked:

```
@Override
public String getURL(
  PortletRequest portletRequest, PortletResponse portletResponse) {

  LiferayPortletURL liferayPortletURL =
    (LiferayPortletURL)_portal.getControlPanelPortletURL(
      portletRequest, ConfigurationAdminPortletKeys.SYSTEM_SETTINGS,
      PortletRequest.RESOURCE_PHASE);

  liferayPortletURL.setResourceID("export");

  return liferayPortletURL.toString();
}
```

5. Override the `getWeight()` method to specify the order that the action should appear in the list:

```
@Override
public double getWeight() {
  return 1;
}
```

6. Override the `isShow()` method to specify the context in which the action should be displayed:

```
@Override
public boolean isShow(PortletRequest portletRequest) {
    return true;
}
```

7. Define the view where you want to display the configuration options. 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.

8. If the portlet can be added to a page and you want to always include the configuration options, add this initialization parameter to the portlet's properties:

```
javax.portlet.init-param.always-display-default-configuration-icons=true
```

In this example, the action appears in the System Settings portlet. To make it appear in a secondary screen, you can use the `path` property as shown below. 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):

```
@Component(
    immediate = true,
    property = {
        "javax.portlet.name=" + ConfigurationAdminPortletKeys.SYSTEM_SETTINGS,
        "path=/view_factory_instances"
    },
    service = PortletConfigurationIcon.class
)
public class ExportFactoryInstancesIcon extends BasePortletConfigurationIcon {

    @Override
    public String getMessage(PortletRequest portletRequest) {
        ThemeDisplay themeDisplay = (ThemeDisplay)portletRequest.getAttribute(
            WebKeys.THEME_DISPLAY);

        ResourceBundle resourceBundle = ResourceBundleUtil.getBundle(
            themeDisplay.getLocale(), ExportFactoryInstancesIcon.class);

        return LanguageUtil.get(resourceBundle, "export-entries");
    }

    @Override
    public String getMethod() {
        return "GET";
    }

    @Override
    public String getURL(
        PortletRequest portletRequest, PortletResponse portletResponse) {

        LiferayPortletURL liferayPortletURL =
            (LiferayPortletURL)_portal.getControlPanelPortletURL(
                portletRequest, ConfigurationAdminPortletKeys.SYSTEM_SETTINGS,
                PortletRequest.RESOURCE_PHASE);

        ConfigurationModel factoryConfigurationModel =
            (ConfigurationModel)portletRequest.getAttribute(
                ConfigurationAdminWebKeys.FACTORY_CONFIGURATION_MODEL);

        liferayPortletURL.setParameter(
            "factoryPid", factoryConfigurationModel.getFactoryPid());

        liferayPortletURL.setResourceID("export");

        return liferayPortletURL.toString();
    }

    @Override
    public double getWeight() {
        return 1;
    }

    @Override
    public boolean isShow(PortletRequest portletRequest) {
        ConfigurationModelIterator configurationModelIterator =
            (ConfigurationModelIterator)portletRequest.getAttribute(
                ConfigurationAdminWebKeys.CONFIGURATION_MODEL_ITERATOR);
```

```
            if (configurationModelIterator.getTotal() > 0) {
                return true;
            }

            return false;
        }

        @Reference
        private Portal _portal;

    }
```

This covers some of the available methods. See the Javadoc for a complete list of the available methods.

Great! Now you know how to configure your admin app's actions.

Related Topics

- Applying Clay Patterns to Your Navigation Bar
- Configuring Your Application's Title and Back Link

50.5 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) avoids this issue. Single Page Applications drastically cut down on load times by loading only a single HTML page that's dynamically updated as the user interacts and navigates through the site. This provides a more seamless app experience by eliminating page reloads. **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. All right! 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. The page history didn't update as you navigated through the app; Only the main app URL was saved.

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 see examples and read more about SennaJS at its website.

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. Since this module is included by default, you shouldn't have to do anything. If you've removed it, deploy `com.liferay.frontend.js.spa.web-[version]` module and enable it, and you're all set to use SPA.

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 are 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 *Infrastructure* under the *PLATFORM* heading.

3. Click *Frontend SPA Infrastructure*.

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.

Navigation Exception Selectors: Defines a CSS selector that SPA should ignore.

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 Timeout: 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 across an entire Liferay DXP instance, 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 `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 you are navigating 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.

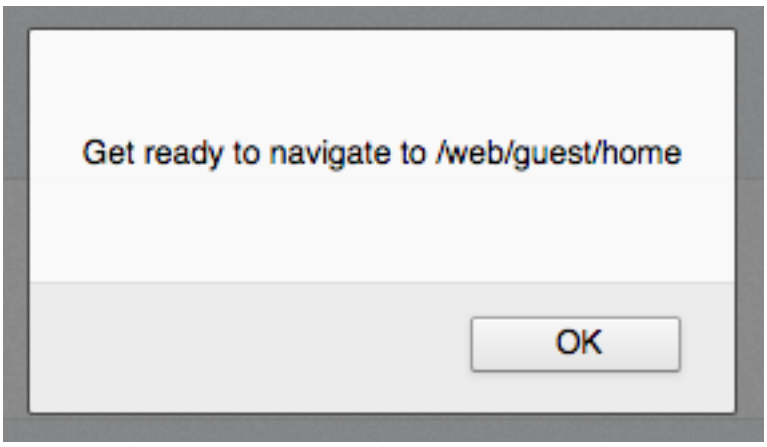


Figure 50.6: You can leverage SPA lifecycle events in your apps.

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`, `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

Preparing your JavaScript Files for ES2015+

Using ES2015+ Modules in Your Portlet

50.6 Creating Layouts inside Custom Portlets

Layout templates specify how your portlets and content are organized on your site pages. What if, instead, you want to organize your portlet's content? `<aui>` tags let you create layouts using Bootstrap within your portlets. This tutorial explains this process.

Follow these steps:

1. Open your portlet's JSP and include the AUI taglib declaration if it's not already included:

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

2. Wrap your portlet's content in `<aui:container>` tags. If you wish to only have a portion of your portlet's content in a layout, wrap that portion with a `<aui:container>` tags.

3. In between the `<aui:container>...</aui:container>` tags, add a set of `<aui:row>` tags for each row that you want in your portlet's layout.

4. Add a set of `<aui:col>` tags for each column that you want in the row. Repeat this step for each row in the layout.

A complete example is shown below:

```
<aui:container>
  <aui:row>
    <aui:col md="12">
      <h1>My Custom Layout Portlet</h1>
    </aui:col>
  </aui:row>
  <aui:row>
    <aui:col md="4" sm="6">
      <h2>Column One</h2>
      <p>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Integer eget pulvinar quam. Vivamus ornare leo libero, sed
        mollis urna aliquam ac. Duis porta sapien non felis varius, in
        iaculis orci fermentum. Etiam quis molestie elit, in tempor
        risus. Morbi varius elementum lectus at feugiat. Quisque
        dapibus orci ac dui eleifend, ut ullamcorper nulla sagittis.
        Ut ac scelerisque sem.
      </p>
    </aui:col>
    <aui:col md="8" sm="6">
    <h2>Column Two</h2>
      <p>
        Aliquam hendrerit augue sed nisl ullamcorper pulvinar. Donec
        tristique congue erat ac condimentum. Suspendisse vehicula
        nunc vel velit imperdiet dapibus. In hac habitasse platea
        dictumst. Morbi eleifend arcu sit amet magna faucibus, vitae
        posuere erat finibus. Sed hendrerit convallis ipsum id luctus.
        Aliquam aliquam consequat turpis eu vulputate. Nulla vitae
        libero lorem. Proin nec lacus et nunc laoreet posuere.
        Vestibulum euismod vestibulum faucibus. Vivamus dolor justo,
        malesuada ac libero ac, feugiat varius leo. Integer viverra
        nisi vel fringilla aliquam.
      </p>
      <p>
        Suspendisse potenti. Mauris neque nisl, hendrerit a sem at,
        rutrum dictum arcu. Ut aliquet tortor vel tortor interdum
        dictum. Sed non sapien quam. Nunc aliquet in massa elementum
        aliquam. Cras convallis tristique ante ut ultrices. Aenean
        quis congue nulla. Integer in lacus lectus. Mauris maximus,
        nibh sit amet pharetra laoreet, sem dolor eleifend metus, non
        semper sem justo vel mauris. Praesent tristique quis risus
        vulputate faucibus. Nullam feugiat diam vel elit pharetra, id
        porta velit fringilla. Pellentesque metus justo, dictum et
        dolor venenatis, pretium egestas massa. Donec risus nisi,
        elementum in lectus id, imperdiet blandit mauris.
      </p>
    </aui:col>
  </aui:row>
</aui:container>
```

The columns in the second row take advantage of Bootstrap's grid classes to create responsive layouts. On medium sized view ports, `column-1` is 33.33% width and `column-2` is 66.66% width, but on small sized view ports both `column-1` and `column-2` are 50% width.

AUI Layout Tag Attributes

This section contains a list of the available attributes for each tag along with a brief description of its purpose.

AUI Container

The `<aui:container>` tag creates a container `<div>` tag to wrap `<aui:row>` components and offer additional styling.

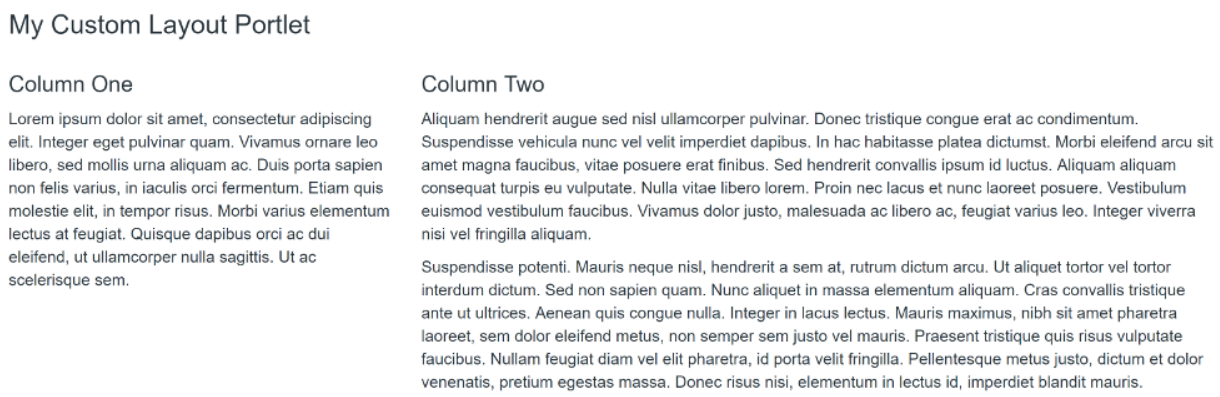


Figure 50.7: Custom layouts in your portlets let you organize your portlet's content with the user in mind.

Figure 50.8: You can take advantage of Bootstrap's grid classes to create responsive layouts within your custom portlets.

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-`

Attribute	Type	Description
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.

Now you know how to create layouts inside your portlets!

Related Topics

Layout Templates with the Liferay Theme Generator

Chapter 51

Customizing

Portlets are the main application platform, and themes let you style your sites. You can also modify existing behavior, globally and in the installed applications, as well as specify your own look and feel.

The *Customizing* tutorials below show you how to affect your site in the following ways:

- Add, modify, or remove content
- Modify behavior
- Perform actions that respond to events

CHAPTER 52

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.

52.1 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.

52.2 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. Since overriding a JSP using an OSGi fragment or a Custom JSP Bag is 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. These approaches should only be used as a last resort.

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	Completely overrides a module's JSP using an OSGi fragment	Changes to the original JSP or module can cause runtime errors.
Custom JSP bag	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!

52.3 Customizing JSPs with Dynamic Includes

The `liferay-util:dynamic-include` tag is placeholder into which you can inject content. Every JSP's 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.

Blogs entries contain a good example of how dynamic includes work. 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 an `@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 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 JSP dynamically includes your content. Congratulations on injecting dynamic content into a JSP!

52.4 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 the content sent back to the client.

This demonstration uses a portlet filter to modify content in Liferay's Blogs portlet. For reference, you can download the example module.

1. 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"
   }
   ```

2. 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);
            }
        }
    }

}
```

3. 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
)
```

4. 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 let you work with all of a JSP's content.

52.5 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 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).
3. Liferay cannot guarantee that JSPs overridden by fragments can be upgraded.

Using OSGi fragments to override JSPs is a bad practice, equivalent to using Ext plugins to customize Liferay DXP. They should only be used as a last resort. Liferay's API based approaches

to overriding JSPs (i.e., Dynamic Includes and Portlet Filters), on the other hand, provide more stability as they customize specific parts of JSPs that are safe to override. Also, the API based approaches don't limit your override to a specific host module version. If you are 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:

- The host module's symbolic name and version in the OSGi header `Fragment-Host` declaration.
- 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 manifest 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 must 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 Module JSP Override sample project.

Related Topics

Upgrading App JSP Hooks

52.6 JSP Overrides Using Custom JSP Bag

Liferay's API based approaches to overriding JSPs (i.e., Dynamic Includes and Portlet Filters) 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. Using Custom JSP Bags to override JSPs is a bad practice, equivalent to using Ext plugins to customize Liferay DXP. If you're maintaining existing Custom JSP Bags, however, this tutorial explains how they work.

Important: Liferay cannot guarantee that JSPs overridden using Custom JSP Bag can be upgraded.

A Custom JSP Bag module must satisfy these criteria:

- Provides and specifies a custom JSP for the JSP you're extending.
- Includes a `CustomJspBag` implementation for serving the custom JSPs.

The module provides transportation for this code into Liferay's OSGi runtime. After you create your new module, continue with providing your custom JSP.

Providing a Custom JSP

Create your JSPs to override Liferay DXP core JSPs. If you're using the Maven Standard Directory Layout, place your JSPs under `src/main/resources/META-INF/jsps`. For example, if you're overriding

```
portal-web/docroot/html/common/themes/bottom-ext.jsp
```

place your custom JSP at

```
[your module]/src/main/resources/META-INF/jsps/html/common/themes/bottom-ext.jsp
```

Note: If you place custom JSPs somewhere other than `src/main/resources/META-INF/jsps` in your module, assign that location to a `-includeresource: META-INF/jsps=` directive in your module's `bnd.bnd` file. For example, if you place custom JSPs in a folder `src/META-INF/custom_jsps` in your module, specify this in your `bnd.bnd`:

```
-includeresource: META-INF/jsps=src/META-INF/custom_jsps
```

Implement a Custom JSP Bag

Liferay DXP (specifically the `CustomJspBagRegistryUtil` class) loads JSPs from `CustomJspBag` services. The following steps implement a custom JSP bag.

1. In your module, create a class that implements `CustomJspBag`.
2. Register your class as an OSGi service by adding an `@Component` annotation to it, like this:

```
@Component(
    immediate = true,
    property = {
        "context.id=BladeCustomJspBag",
        "context.name=Test Custom JSP Bag",
        "service.ranking:Integer=100"
    }
)
```

   - **`immediate = true`:** Makes the service available on module activation.
   - **`context.id`:** Your custom JSP bag class name. Replace `BladeCustomJspBag` with your class name.
   - **`context.name`:** A more human readable name for your service. Replace it with a name of your own.
   - **`service.ranking:Integer`:** A priority for your implementation. The container chooses the implementation with the highest priority.

3. Implement the `getCustomJspDir` method to return the folder path in your module's JAR where the JSPs reside (for example, `META-INF/jsps`).

```
@Override
public String getCustomJspDir() {
    return "META-INF/jsps/";
}
```

4. Create an `activate` method and the following fields. The method adds the URL paths of all your custom JSPs to a list when the module is activated.

```
@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 Bundle _bundle;
private List<String> _customJsps;
```

5. Implement the `getCustomJsps` method to return the list of this module's custom JSP URL paths.

```
@Override
public List<String> getCustomJsps() {
    return _customJsps;
}
```

6. Implement the `getURLContainer` method to return a new `com.liferay.portal.kernel.url.URLContainer`. Instantiate the URL container and override its `getResources` and `getResource` methods. The `getResources` method looks up all the paths to resources in the container by a given path. It returns a `HashSet` of `Strings` for the matching custom JSP paths. The `getResource` method returns one specific resource by its name (the path included).

```
@Override
public URLContainer getURLContainer() {
    return _urlContainer;
}

private final URLContainer _urlContainer = new URLContainer() {

    @Override
    public URL getResource(String name) {
        return _bundle.getEntry(name);
    }

    @Override
    public Set<String> getResources(String path) {
        Set<String> paths = new HashSet<>();

        for (String entry : _customJsps) {
```

```
            if (entry.startsWith(path)) {
               paths.add(entry);
            }
        }

        return paths;
    }

};
```

7. Implement the `isCustomJspGlobal` method to return `true`.

```
@Override
public boolean isCustomJspGlobal() {
    return true;
}
```

Now your module provides custom JSPs and a custom JSP bag implementation. When you deploy it, Liferay DXP uses its custom JSPs in place of the core JSPs they override.

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 keeps 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 must add something to `bottom.jsp`, override `bottom-ext.jsp`.

Since Liferay DXP 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 is a last resort. Your override may 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 Liferay 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 Liferay DXP 7.0, the use case for this has shrunk considerably. If you must 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

52.7 Overriding Inline Content Using JSPs

Some Liferay DXP core content, such as tag library tags, can only be overridden using JSPs ending in `.readme`. The suffix `.readme` facilitates finding them. The code from these JSPs is now inlined (brought into Liferay DXP Java source files) to improve performance. Liferay DXP ignores JSP files with the `.readme` suffix. If you add code to a JSP `.readme` file and remove the `.readme` suffix, Liferay DXP uses that JSP instead of the core inline content. This tutorial shows you how to make these customizations.

Important: This type of customization is a last resort. Your override may break due to the nature of this implementation, and core functionality can go down with it. Liferay cannot guarantee that content overridden using JSP `.readme` files can be upgraded.

Warning: Modifying a Liferay DXP tag library tag affects all uses of that tag in your Liferay DXP installation.

Here's how to override inline content using JSPs:

1. Create a Custom JSP Bag for deploying your JSP. Note the module folder you're storing the JSPs in: the default folder is `[your module]/src/main/resources/META-INF/jsps/`

```
**Note:** you can develop your JSP anywhere, but a Custom JSP Bag module
provides a straightforward way to build and deploy it.
```

---

2. Download the Liferay DXP source code or browse the source code on GitHub (Liferay Portal CE).

3. Search the source code for a `.jsp.readme` file that overrides the tag you're customizing.

```
**Note:** Files ending in `-ext.jsp.readme` let you prepend or
append new content to existing content. Examples include the
`bottom-test.jsp.readme`, `bottom-ext.jsp.readme`,
`body_top-ext.jsp.readme`, and `body_bottom-ext.jsp.readme` files in
the Liferay DXP application's `portal-web/docroot/html/common/themes` folder.
```

---

4. Copy the `.jsp.readme` file into your project and drop the `.readme` suffix. Use the same relative file path Liferay DXP uses for the `.jsp.readme` file. For example, if the file in Liferay DXP is

```
portal-web/docroot/html/taglib/aui/fieldset/start.jsp.readme
```

use file path

```
[your module]/src/main/resources/META-INF/jsps/html/taglib/aui/fieldset/start.jsp
```

5. Familiarize yourself with the current UI content and logic, so you can override it appropriately. Tag library tag content logic, for example, is in the respective `*Tag.java` file under `util-taglib/src/com/liferay/taglib/[tag library]/`.
6. Develop your new logic, keeping in mind the current inline logic you're replacing.
7. Deploy your JSP.

Liferay DXP uses your JSP in place of the current inline logic. If you want to walk through an example override, continue with this tutorial. Otherwise, congratulations on a modified `.jsp.readme` file to override core inline content!

Example: Overriding the fieldset Taglib Tag

This example demonstrates changing the `liferay:aui` tag library's `fieldset` tag. Browsing the Liferay DXP web application or the source code at `portal-web/docroot/html/taglib/aui/fieldset` reveals these files:

- `start.jsp.readme`
- `end.jsp.readme`

They can override the logic that creates the start and end of the `fieldset` tag. The `FieldsetTag.java` class's `processStart` and `processEnd` methods implement the current inline content. Here's the `processStart` method:

```
@Override
protected int processStartTag() throws Exception {
    JspWriter jspWriter = pageContext.getOut();

    jspWriter.write("<fieldset class=\"fieldset ");
    jspWriter.write(GetterUtil.getString(getCssClass()));
    jspWriter.write("\" ");

    String id = getId();

    if (id ≠ null) {
        jspWriter.write("id=\"");
        jspWriter.write(id);
        jspWriter.write("\" ");
    }

    jspWriter.write(
        InlineUtil.buildDynamicAttributes(getDynamicAttributes()));

    jspWriter.write(StringPool.GREATER_THAN);

    String lable = getLabel();

    if (lable ≠ null) {
        jspWriter.write(
            "<legend class=\"fieldset-legend\"><span class=\"legend\">");
```

```
        MessageTag messageTag = new MessageTag();

        messageTag.setKey(lable);
        messageTag.setLocalizeKey(getLocalizeLabel());

        messageTag.doTag(pageContext);

        String helpMessage = getHelpMessage();

        if (helpMessage != null) {
            IconHelpTag iconHelpTag = new IconHelpTag();

            iconHelpTag.setMessage(helpMessage);

            iconHelpTag.doTag(pageContext);
        }

        jspWriter.write("</span></legend>");
    }

    if (getColumn()) {
        jspWriter.write("<div class=\"row\">");
    }
    else {
        jspWriter.write("<div class=\"\">");
    }

    return EVAL_BODY_INCLUDE;
}
```

The code above does this:

1. Write `<fieldset class=\"fieldset`starting tag.
2. Write the CSS class name attribute.
3. If the tag has an ID, add the `id` as an attribute.
4. Write the tag's dynamic attribute (map).
5. Close the starting `fieldset` tag.
6. Get the tag's `label` attribute.
7. Write the starting `legend` element.
8. Use `getLocalizeLabel()` to add the localized label in the `legend`.
9. If there's a help message (retrieved from `getHelpMessage()`), write it in an `icon-help-tag`.
10. Write the closing `legend` tag.
11. If there's a column attribute, write `<div class=\"row\">`; else write `<div class=\"\">`.

Replicating the current logic in your custom JSP helps you set up the tag properly for customizing. The `init.jsp` for `fieldset` initializes all the variables required to create the starting tag. You can use the variables in the `start.jsp`. The logic from `FieldsetTag`'s `processStart` method converted to JSP code for `start.jsp` (renamed from `start.jsp.readme`) would look like this:

```
<%@ include file="/html/taglib/aui/fieldset/init.jsp" %>

<fieldset class="fieldset <%= cssClass %>" <%= Validator.isNotNull(id) ? "id=\"" + id + "\"" : StringPool.BLANK %> <%= InlineUtil.buildDynamicAttribu
    <c:if test="<%= Validator.isNotNull(label) %>">
        <legend class="fieldset-legend">
            <span class="legend">
                <liferay-ui:message key="<%= label %>" localizeKey="<%= localizeLabel %>" />

                <c:if test="<%= Validator.isNotNull(helpMessage) %>">
                    <liferay-ui:icon-help message="<%= helpMessage %>" />
                </c:if>
            </span>
        </legend>
    </c:if>

    <div class="<%= column ? "row" : StringPool.BLANK %>">
```

Tip: A *Tag.java file's history might reveal original JSP code that was inlined. For example, the logic from `fieldset` tag's `start.jsp` was inlined in this commit.

On deploying the `start.jsp`, the `fieldset` tags render the same as they did before. This is expected because it uses the same logic as `FieldsetTag`'s `processStart` method.

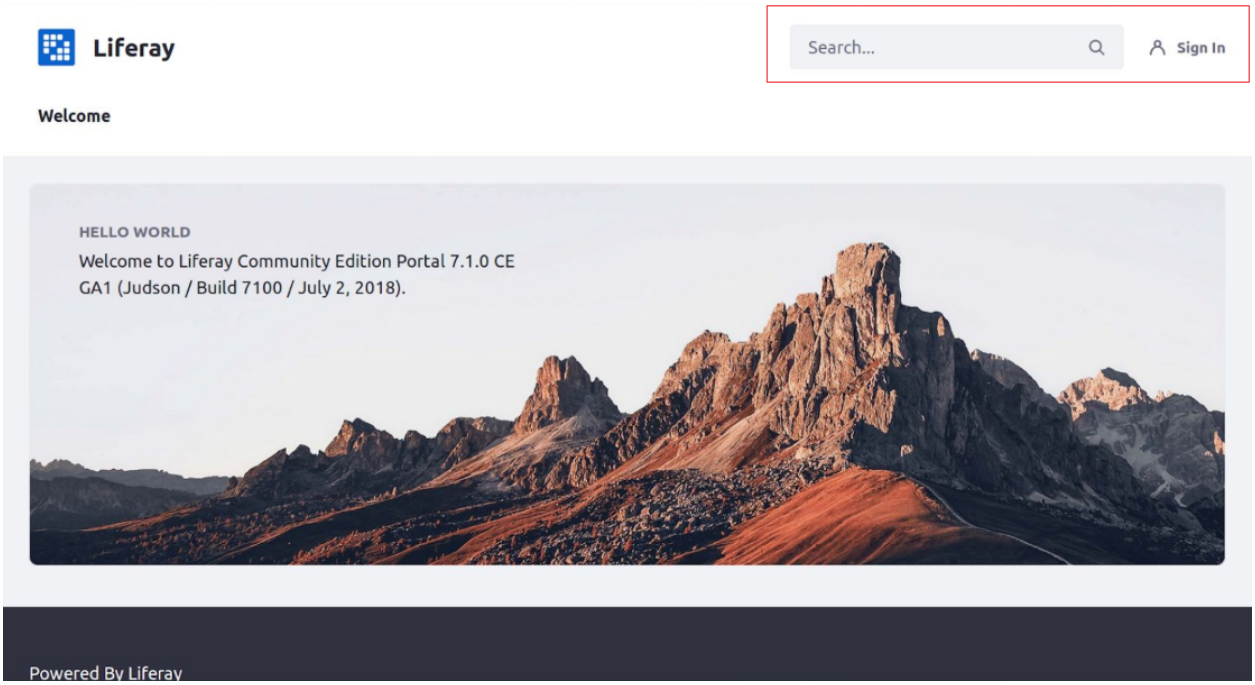


Figure 52.1: Liferay DXP's home page's search and sign in components are in a `fieldset`.

The `fieldset` starting logic is ready for customization. To test that this works, you'll print the word *test* surrounded by asterisks before the end of the `fieldset` tag's starting logic. Insert this line before the `start.jsp`'s last `div` tag:

```
<c:out value="**********test**********"/>
```

Redeploy the JSP and refresh the page to see the text printed above the `fieldset`'s fields.

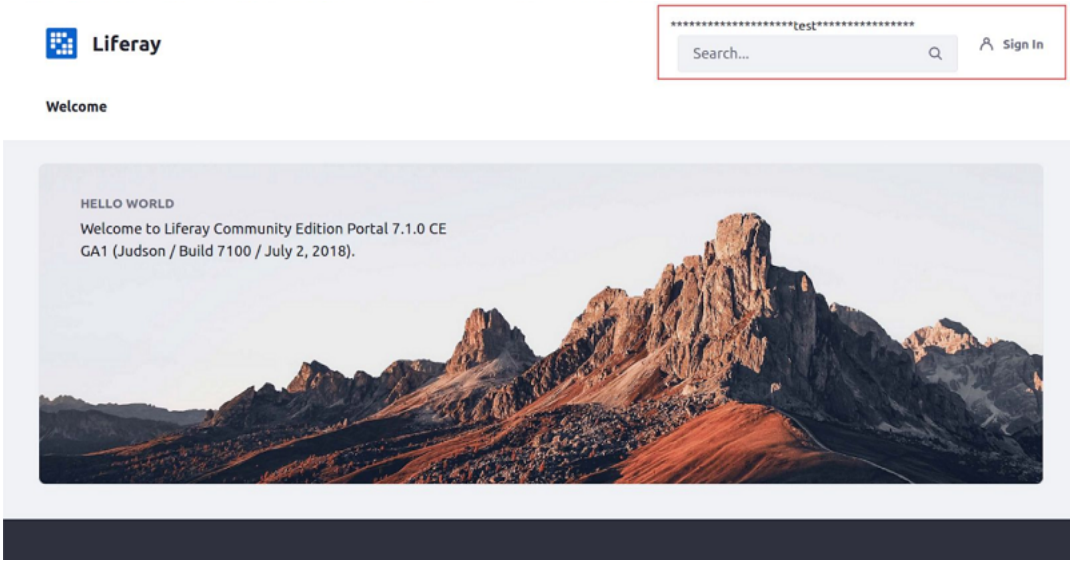


Figure 52.2: Before the `fieldset`'s nested fields, it prints *test* surrounded by asterisks.

You know how to override specific Liferay DXP core inline content using Liferay's `.jsp.readme` files.

Related Topics

Customizing JSPs with Dynamic Includes

JSP Overrides Using Portlet Filters

CHAPTER 53

Overriding Liferay Services (Service Wrappers)

Why might you need to customize Liferay services? Perhaps you've added a new 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 Liferay APIs or other services built using Service Builder. 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 execute a few print statements that before executing the original service implementations.

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);
}
```

Build and deploy your module. Congratulations! You've created and deployed a Liferay service wrapper!

53.1 Related Topics

Upgrading Service Wrappers

Installing Blade CLI

Creating Projects with Blade CLI

CHAPTER 54

OVERRIDING OSGI SERVICES

Components register as services with the OSGi service registry. A service component's availability, ranking, and attributes determine whether components referring to the service type bind to that particular service. Liferay DXP's OSGI container is a dynamic environment in which services come and go and can be overridden, which means that if there's a service whose behavior you want to change, you can override it. Here are the steps for overriding a service:

1. Get the service and service reference details
2. Create a custom service
3. Configure components to use your custom service

Note: The Service Builder services in portal-impl are Spring beans that Liferay makes available as OSGi services.

Start with examining the service you want to override.

54.1 Examining an OSGi Service to Override

Creating and injecting a custom service in place of an existing service requires three things:

- Understanding the service interface
- The existing service
- The references to the service

Your custom service must implement the service interface, match references you want, and might need to invoke the existing service.

Getting components to adopt your custom service immediately can require reconfiguring their references to the service. Here you'll flesh out service details to make these decisions.

Gathering Information on a Service

1. Since component service references are extension points, start with following the tutorial Finding Extension Points to determine the service you want to override and components that use that service.

2. Once you know the service and components that use it, use Gogo Shell's Service Component Runtime (SCR) to inspect the components and get the service and reference details. The Gogo Shell command `scr:info [componentName]` lists the component's attributes and service references.

Here's an example `scr:info` command and results (abbreviated with `...`) that describe component `override.my.service.reference.OverrideMyServiceReference` (from sample module override-my-service-reference) and its reference to a service of type `override.my.service.reference.service.api.SomeService`:

```
> scr:info override.my.service.reference.OverrideMyServiceReference

...
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: 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 `scr:info` results, like the ones above, contain information relevant to injecting a custom service. Here's what you'll do with the information:

1. Copy the service interface name

2. Copy the existing service name

3. Gather reference configuration details (if reconfiguration is necessary)

Start with the service interface.

Step 1: Copy the Service Interface Name

The reference's *Interface Name* is the service interface's fully qualified name.

```
...
Reference: _someService
    Interface Name: override.my.service.reference.service.api.SomeService
    ...
```

Copy and save the interface name, because it's the type your custom service must implement.

Javadocs for Liferay DXP service interfaces are at these locations:

- Liferay DXP core Javadocs
- Liferay DXP app Javadocs
- MVNRepository and Maven Central (for Liferay and non-Liferay artifact Javadocs).

Step 2: Copy the Existing Service Name

If you want to invoke the existing service along with your custom service, get the existing service name.

The `src:info` result's Component Configuration section lists the existing service's fully qualified name. For example, the `OverrideMyServiceReferencePortlet` component's references `_someService` is bound to a service component whose fully qualified name is `override.my.service.reference.service.impl.SomeServiceImpl`.

```
Component Configuration:
...
SatisfiedReference: _someService
  ...
  Bound to:        6840
      Properties:
        ...
        component.name = override.my.service.reference.service.impl.SomeServiceImpl
```

Copy the `component.name` so you can reference the service in your custom service.
Here's an example of referencing the service above.

```
@Reference  (
    target = "(component.name=override.my.service.reference.service.impl.SomeServiceImpl)"
)
private SomeService _defaultService;
```

Step 3: Gather Reference Configuration Details (if reconfiguration is needed)

The service reference's policy and policy option determine a component's conditions for adopting a particular service.

- If the reference's policy option is `greedy`, it binds to the matching, highest ranking service right away. The reference need not be reconfigured to adopt your service.
- If policy is `static` and its policy option is `reluctant`, however, the component requires one of the following conditions to switch from using the existing service it's referencing to using the matching, highest ranking service (i.e., you'll rank your custom service highest):

    1. The component is reactivated

2. The component's existing referenced service is unavailable
3. The component's reference is modified so that it does not match the existing service but matches your service

Reconfiguring the reference can be the quickest way for the component to adopt a new service.
Gather these details:

- *Component name:* Find this at *Component Description* → *Name*. For example,

```
Component Description:
    Name: override.my.service.reference.portlet.OverrideMyServiceReferencePortlet
    ...
```

- *Reference name:* The *Reference* value (e.g., `Reference: _someService`).
- *Cardinality:* Number of service instances the reference can bind to.

Note: Declarative Services makes all components configurable through OSGi Configuration Admin. 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 `@Reference` is on a field called `_someService`, for example, then the reference name is `_someService`.
- If the `@Reference` is on a method, then heuristics derive the reference name. Method name suffix is used and 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`.

After creating your custom service (next), you'll use the details you collected here to configure the component to use your custom service.

Congratulations on getting the details required for overriding the OSGi service!

Related Topics

OSGi Services and Dependency Injection with Declarative Services
Finding Extension Points
Gogo Shell

54.2 Creating a Custom OSGi Service

It's time to implement your OSGi service. Make sure to examine the service and service reference details, if you haven't done so already. Here you'll create a custom service that implements the service interface, declares it an OSGi service of that type, and makes it the best match for binding with other components.

The example custom service `CustomServiceImpl` (from sample module `overriding-service-reference`) implements service interface `SomeService`, declares itself an OSGi service of the `SomeService` service type, and even delegates work to the existing service. Examine this example code as you follow the steps for creating your custom service.

```
@Component(
    property = {
        "service.ranking:Integer=100"
    },
    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  (
        target = "(component.name=override.my.service.reference.service.impl.SomeServiceImpl)"
    )
    private SomeService _defaultService;
}
```

Here are the steps to create a custom OSGi service:

1. Create a module.
2. Create your custom service class so that it `implements` the service interface you want. In the example above, `CustomServiceImpl implements SomeService`. Step 5 (later) demonstrates implementing the interface methods.
3. Make your class a Declarative Services component that is the best match for references to the service interface:

    - Use an `@Component` annotation and `service` attribute to make your classes a Declarative Services (DS) component. This declares your class to be an OSGi service that can be made available in the OSGi service registry. The example class above is a DS service component of service type `SomeService.class`.
    - Use a `service.ranking:Integer` component property to rank your service higher than existing services. The `"service.ranking:Integer=100"` property above sets the example's ranking to `100`.

4. If you want to invoke the existing service implementation, declare a field that uses a Declarative Services reference to the existing service. Use the `component.name` you copied when you examined the service to target the existing service. The example above refers to an existing service like this:

    ```
    @Reference  (
        target = "(component.name=override.my.service.reference.service.impl.SomeServiceImpl)"
    )
    private SomeService _defaultService;
    ```

    The field lets you invoke the existing service in your custom service.

5. Override the interface's methods. Optionally, delegate work to the existing service implementation (see previous step).

   The example custom service's `doSomething` method delegates work to the original service implementation.

6. Register your custom service with the OSGi runtime framework by deploying your module.

Components that reference the service type you implemented and whose reference policy option is `greedy` bind to your custom service immediately. Components bound to an existing service and whose reference policy option is `reluctant` can be dynamically reconfigured to use your service. That's demonstrated next.

Related Topics

OSGi Services and Dependency Injection with Declarative Services

54.3 Reconfiguring Components to Use Your OSGi Service

In many cases, assigning your custom service (service) a higher ranking convinces components to unbind from their current service and bind to yours. In other cases, components keep using their current service. Why is that? And how do you make components adopt your service? The component's service reference policy option is the key to determining the service.

Here are the policy options:

`greedy`: The component uses the matching, highest ranking service as soon as it's available.

`reluctant`: The component uses the matching, highest ranking service available in the following events:

- the component is (re)activated
- the component's existing referenced service becomes unavailable
- the component's reference is modified so that it no longer matches the existing bound service

In short, references with greedy policy options adopt your higher ranking service right away, while ones with reluctant policy options require particular events. What's great is that Liferay DXP's Configuration Admin lets you use configuration files (config files) or the API to swap in service reference changes on the fly. Here you'll use a config file to reconfigure a service reference to use your custom service immediately.

This tutorial uses example modules `override-my-service-reference` and `overriding-service-reference` to demonstrate reconfiguring a service reference, binding the component to a different service. You can download the modules 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 `OverrideMyServiceReferencePort` 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`.

You're ready to reconfigure a component's service reference to target your custom service.

Reconfiguring the Service Reference

Liferay DXP's Configuration Admin lets you use configuration files to swap in service references on the fly.

1. Create a system configuration file named after the referencing component. Follow the name convention `[component].config`, replacing `[component]` with the component name. The configuration file name for the example component `override.my.service.reference.portlet.OverrideMyServiceReferen` is:

   ```
   override.my.service.reference.portlet.OverrideMyServiceReferencePortlet.config
   ```

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.

   This example filters on the `component.name` service property:

   ```
   _someService.target="(component.name\=overriding.service.reference.service.CustomServiceImpl)"
   ```

   This example filters on the `service.vendor` service property:

   ```
   _someService.target="(service.vendor\=Acme, Inc.)"
   ```

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 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 the custom service component `overriding.service.reference.service.CustomServiceImpl`. `CustomServiceImpl` references default service `SomeServiceImpl` to delegate work to it.

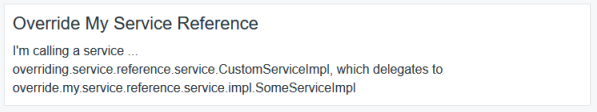


Figure 54.1: Because the example component's service reference is overridden by the configuration file deployment, the portlet indicates it's calling the custom service.

Liferay DXP processed the configuration file and injected the service reference, which in turn bound the custom service to the referencing component!

Related Topics

OSGi Services and Dependency Injection with Declarative Services
    Finding Extension Points
    Using Felix Gogo Shell

CHAPTER 55

OVERRIDING LANGUAGE KEYS

Core and portlet module `Language*.properties` files implement site internationalization. They're fully customizable, too. These tutorials demonstrate this in the following topics:

- Overriding Liferay's Language Keys
- Overriding a Module's Language Keys

55.1 Overriding 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's modules. The process for overriding a module's language keys is different from the process for overriding Liferay's 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 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`
```

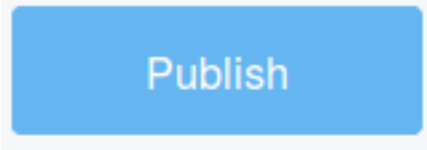


Figure 55.1: Messages displayed in Liferay's user interface can be customized.

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.

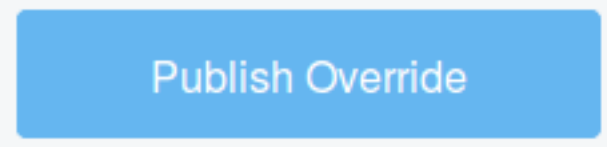


Figure 55.2: This button uses the overridden `publish` key.

That's all there is to overriding Liferay's language keys.

Related Topics

Resource Bundle Override Sample Project
Upgrading Core Language Key Hooks
Internationalization

55.2 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 Liferay's 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.l
servlet.context.name=blogs-web
```

The example `Provide-Capability` header has two parts:

1. `liferay.resource.bundle;resource.bundle.base.name="content.Language"` declares that the module provides a resource bundle with the base name `content.language`.
2. The `liferay.resource.bundle;resource.bundle.aggregate:String=...` directive specifies the list of bundles with resource bundles to aggregate, 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.w` 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 Liferay's language keys too.

Related Topics

Resource Bundle Override Sample Project
Upgrading Core Language Key Hooks
Internationalization

CHAPTER 56

OVERRIDING MVC COMMANDS

MVC Commands are used to break up the controller layer of Liferay MVC applications 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 customization of the component, give it a higher service ranking, and deploy it.

All existing components that reference the original MVC command service component (using a greedy reference policy) switch to reference your new one. Any existing reluctant references to the original command must be configured to reference the new one. Once they're configured with the new service component, their JSP's command URLs invoke the new custom MVC command.

Here are the customization options available for each Liferay MVC Command type:

- MVCActionCommand: Add logic
- MVCRenderCommand:
  - Add logic
  - Redirect to a different JSP
- MVCResourceCommand: Add logic

These tutorials demonstrate each MVC command customization option. Since the steps for adding logic are generally the same across MVC command types, start with adding logic.

56.1 Adding Logic to MVC Commands

You can completely override MVC commands, or any OSGi service for that matter, but *adding logic* to the commands is the better option. Discarding necessary logic is bad. Conversely any logic you copy from the original might not work in new versions of the portlet. Adding custom logic while continuing to invoke the original logic decouples the custom class from the original implementation. Keeping the new logic separate form the original logic keeps the code clean, maintainable, and easy to understand.

Here are the steps for adding logic to MVC commands.

1. Implement the interface
2. Publish as a component
3. Refer to the original implementation
4. Add the logic, and call the original

Step 1: Implement the interface

Implement the respective MVC Command interface either directly or by extending an existing base class that implements it. Extending a base class for the interface relieves you from implementing logic that should typically be a part of most command implementations. For example, to add logic to the Blogs portlet's `EditEntryMVCActionCommand`, you would extend base class `BaseMVCActionCommand`.

```
public class CustomBlogsMVCActionCommand extends BaseMVCActionCommand
```

Check the MVC command interfaces for existing base classes:

- `MVCActionCommand`
- `MVCRenderCommand`
- `MVCResourceCommand`

Next make your class a service component.

Step 2: Publish as a component

The Declarative Services `@Component` annotation facilitates customizing MVC commands. All the customization options require publishing your MVC command class as a component. For example, this `@Component` annotation declares an `MVCActionCommand` service.

```
@Component(
    immediate = true,
    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 {
    ...
}
```

It publishes `CustomBlogsMVCActionCommand` as a service component for the `MVCActionCommand` class. Upon resolving, it's activated immediately because `immediate = true`. The component is invoked in the Blogs Admin portlet by the command URL `/blogs/edit_entry`. Its service ranking of `100` prioritizes it ahead of the original service component, whose ranking is `0`.

Here's what you need to specify in an `@Component` annotation for your custom MVC command:

- `javax.portlet.name`: for each portlet you want the customization to affect. JSPs in these portlets can invoke the MVC command via applicable command URL tags. You can specify the same portlets as the original MVC command or a subset of those portlets.
- `mvc.command.name`: this property declares the command URL that maps to this custom MVC command component.

- `service.ranking:Integer`: set this property to a higher integer than the original service implementation's ranking. The ranking tells the OSGi runtime which service to use, in cases where multiple components register the same service, with the same properties. The higher the integer you specify here, the more weight your component carries. Liferay's service implementations typically have a `0` ranking.
- `service`: this attribute specifies the service (interface) to override .
- `immediate`: set this attribute to `true` to activate your component immediately upon resolution.

You can refer back to this list as you add `@Component` annotations to your custom MVC commands. Next reference the original implementation.

Step 3: Refer to the original implementation

Use a field annotated with `@Reference` to fetch a reference to the original MVC command component. If there are no additional customizations on the original component, this reference will be for the original MVC command type. For example, this field references the original MVC command component `EditEntryMVCActionCommand`.

```
@Reference(
    target = "(component.name=com.liferay.blogs.web.internal.portlet.action.EditEntryMVCActionCommand)")
protected MVCActionCommand mvcActionCommand;
```

Here's how to add the reference:

1. Declare the field as the MVC command interface type that it is. For example, the `mvcActionCommand` field is type `MVCActionCommand`.
2. Add the `@Reference` annotation.
3. In the annotation, define a `target` attribute that filters on a `component.name` equal to the default service implementation class's fully qualified name.

When your custom component resolves, the OSGi runtime assigns the targeted service to your field. It's time to add your custom logic.

Step 4: Add the logic

Adding the logic involves overriding the primary method of the base class you're extending or the interface you're implementing. In your method override, add your new logic AND then invoke the original implementation. For example, the following method overrides `BaseMVCActionCommand`'s method `doProcessAction`.

```
@Override
protected void doProcessAction(
    ActionRequest actionRequest, ActionResponse actionResponse)
throws Exception {
    // Add custom logic here
    ...

    // Call the original service implementation
    mvcActionCommand.processAction(actionRequest, actionResponse);
}
```

The method above defines custom logic and then invokes the original service it referenced in the previous step.

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.

Congratulations on adding logic to your existing MVC command.

56.2 Overriding MVCRenderCommands

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 is 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. MVC render commands can be customized for these purposes:

- Adding Logic to an Existing MVC Render Command
- Redirecting to a new JSP

Start by exploring how to add logic to an existing MVC render command.

Adding Logic to an Existing MVC Render Command

You can add logic to an MVC render command following the general steps for MVC commands. Specifically for MVC render commands, you must directly implement the `MVCRenderCommand` interface and override its `render` method.

For example, this custom MVC render command has a placeholder (i.e., at comment `//Do something here`) for adding logic to the `render` method.:

```
public CustomEditEntryRenderCommand implements MVCRenderCommand {
    @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;
}
```

The example references an `EditEntryMVCRenderCommand` implementation of `MVCRenderCommand`. In the `render` method, you'd replace the placeholder with new logic and then invoke the original implementation's logic by calling its `render` method.

Sometimes, you might need to redirect the request to an entirely new JSP. You can do that from a custom MVC render command module too.

Redirecting to a New JSP

`MVCRenderCommand`'s `render` method returns a JSP path as a String. By default, the JSP must live in the original module, so you cannot simply specify a path to a custom JSP in your override module. To redirect it to a JSP in your new module, you must make the method skip dispatching to the original JSP altogether, by using the constant `MVCRenderConstants.MVC_PATH_VALUE_SKIP_DISPATCH` 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;
}
```

The servlet context provides access to 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
```

Follow these steps to fetch the portlet's servlet context in your custom MVC render command:

1. Add a `ServletContext` field.

   ```
   protected ServletContext servletContext;
   ```

2. Add the `@Reference` annotation to the field and set the annotation to filter on the portlet's module. By convention, Liferay puts portlets in modules whose symbolic names end in `.web`. For example, this servlet context reference filters on a module whose symbolic name is `com.custom.code.web`.

   ```
   @Reference(target = "(osgi.web.symbolicname=com.custom.code.web)")
   protected ServletContext servletContext;
   ```

Implement your `render` method this way:

1. Get a request dispatcher to your module's custom JSP.

   ```
   RequestDispatcher requestDispatcher =
       servletContext.getRequestDispatcher("/custom_edit_entry.jsp");
   ```

2. Include the HTTP servlet request and response in the request dispatcher.

   ```
   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);
        }
```

3. Return the request dispatcher via the constant `MVC_PATH_VALUE_SKIP_DISPATCH`.

```
return MVCRenderConstants.MVC_PATH_VALUE_SKIP_DISPATCH;
```

After deploying your module, the portlets targeted by your custom `MVCRenderCommand` component render your new JSP.

Related Topics

MVC Render Command
    Adding Logic to MVC Commands
    Converting StrutsActionWrappers to MVCCommands

56.3 Overriding MVCActionCommands

In case you want add to a Liferay MVC action command, you can. The OSGi framework lets you override MVC action commands if you follow the instructions for adding logic to MVC commands. It involves registering your custom MVC action command as an OSGi component with the same properties as the original, but with a higher service ranking.

Custom MVC action commands typically extend the `BaseMVCActionCommand` class, and override its `doProcessAction` method, which returns `void`. 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. For example, this `MVCActionCommand` override 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;

}
```

Adding MVC action command logic before existing logic is straightforward and maintains loose coupling between new and old code.

Related Topics

MVC Action Command
Adding Logic to MVC Commands
Overriding MVCRenderCommands
Converting StrutsActionWrappers to MVCCommands

56.4 Overriding MVCResourceCommands

If you need to add functionality to a Liferay MVC resource command, you can. The Liferay MVC command framework supports customizing MVC resource commands. It follows the process for adding logic to MVC commands and it is similar to the ones 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. Implement the `MVCResourceCommand` 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.CaptchaMVCResourceComma` from the Liferay's Login portlet's `login-web` module. It simply prints a line in the console and 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.

Related Topics

MVC Resource Command
    Adding Logic to MVC Commands
    Overriding MVCRenderCommands

56.5 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. To do this, you must 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`.

3. Apply your modifications and save the file.

4. Next, 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

Customizing JSPs

56.6 Overriding lpkg files

Applications are delivered through Liferay Marketplace as *lpkg* files. This is a simple compressed file format that contains .jar files for deploying to Liferay DXP. If you want to examine an application from Marketplace, all you have to do is unzip its .lpkg file to reveal its .jar files.

After examining an application, you may want to customize one of its .jars. Make your customization in a copy of the .jar, but don't deploy it the way you'd normally deploy an application. By overriding the .lpkg file, you can update application modules without modifying the original .lpgk file. Here are the steps:

1. Shut down Liferay DXP.

2. Create a folder called `override` in the `[Liferay Home]/osgi/marketplace` folder](/docs/7-0/deploy/-/knowledge_base/d/installing-product#liferay-home).

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.). To customize or patch any of these .jar files, follow this process:

1. Make your customization and package it in a .jar file.

2. Name your .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. Copy the .jar into the `[Liferay Home]/osgi/static` folder.

Now start Liferay DXP. Note that any time you add and remove .jars this way, Liferay DXP must be shut down and then restarted for the changes to take effect.

If you must roll back your customizations, delete the overriding .jar files: Liferay DXP uses the original .jar on its next startup.

56.7 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 are 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 gets 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 completes.

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 the `BaseModelListener` class.

```
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) tells the listener's `ServiceTrackerCustomizer` which model class to register the listener against.

Register the Model Listener Service

Register the service with Liferay's OSGi runtime for immediate activation. 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 source file `portal-kernel/src/com/liferay/portal/kernel/service/persistence/impl/BasePersiste` 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
Service Builder
Service Builder Persistence

CHAPTER 57

DYNAMIC INCLUDES

Dynamic includes expose extension points in JSPs for injecting additional HTML, adding resources, modifying editors, and more. Several dynamic includes are available. Once you know the dynamic include's key, you can use it to create a module to inject your content.

This section of tutorials lists the available dynamic include keys, along with a description of their use cases and a code example.

The following extension points are covered in this section of tutorials:

Extension Point	Purpose
bottom	Load additional HTML or scripts in the bottom of the theme's body
top_head	Load additional links in the theme's head
top_js	Load additional JS files in the theme's head
WYSIWYG	Add resources to the editor, listen to events, update the configuration, etc.

57.1 WYSIWYG Editor Dynamic Includes

All WYSIWYG editors share the same dynamic include extension points for these things:

- Adding resources, plugins, etc. to the editor:
  com.liferay.frontend.editor.`editorType`.web#`editorName`#additionalResources
- Accessing the editor instance to listen to events, configure it, etc:
  com.liferay.frontend.editor.`editorType`.web#`editorName`#onEditorCreate

The table below shows the `editorType`, variable, and `editorNames` for each editor:

editorType	variable	editorName
alloyeditor	alloyEditor	alloyeditor

editorType	variable	editorName
ckeditor	ckEditor	alloyeditor_bbcode alloyeditor_creole ckeditor ckeditor_bbcode ckeditor_creole
tinymce	tinyMCEEditor	tinymce tinymce_simple

The example below alerts the user when he/she pastes content into the CKEditor. *`DynamicInclude` Java Class:

```
@Component(immediate = true, service = DynamicInclude.class)
public class CKEditorOnEditorCreateDynamicInclude implements DynamicInclude {

    @Override
    public void include(
            HttpServletRequest request, HttpServletResponse response,
            String key)
        throws IOException {

        Bundle bundle = _bundleContext.getBundle();

        URL entryURL = bundle.getEntry(
            "/META-INF/resources/ckeditor/extension/ckeditor_alert.js");

        StreamUtil.transfer(
            entryURL.openStream(), response.getOutputStream(), false);
    }

    @Override
    public void register(
        DynamicInclude.DynamicIncludeRegistry dynamicIncludeRegistry) {

        dynamicIncludeRegistry.register(
            "com.liferay.frontend.editor.ckeditor.web#ckeditor#onEditorCreate");
    }

    @Activate
    protected void activate(BundleContext bundleContext) {
        _bundleContext = bundleContext;
    }

    private BundleContext _bundleContext;

}
```

Example JavaScript:

```
// ckEditor variable is already available in the execution context
ckEditor.on(
    'paste',
    function(event) {
        event.stop();

        alert('Please, do not paste code here!');
    }
);
```

Now you know how to use the WYSIWYG editor dynamic includes.

Related Topics

Adding New Behavior to an Editor
    Bottom JSP Dynamic Includes
    Top Head JSP Dynamic Includes
    Top JS Dynamic Include

57.2 Top Head JSP Dynamic Includes

The `top_head.jsp` dynamic includes load additional links in the theme's head. It uses the following keys:

Load additional links in the theme's head before the existing ones:

```
/html/common/themes/top_head.jsp#pre
```

Alternatively, you can load additional links in the theme's head, after the existing ones:

```
/html/common/themes/top_head.jsp#post
```

The example below injects a link into the top of the `top_head.jsp`:

```
@Component(immediate = true, service = DynamicInclude.class)
public class CssTopHeadDynamicInclude extends BaseDynamicInclude {

    @Override
    public void include(
            HttpServletRequest request, HttpServletResponse response,
            String key)
        throws IOException {

        PrintWriter printWriter = response.getWriter();

        String content =
    "<link href=\"http://localhost:8080/o/my-custom-dynamic-include/css/mentions.css\"
    rel=\"stylesheet\"
    type = \"text/css\" />";

        printWriter.println(content);
    }

    @Override
    public void register(DynamicIncludeRegistry dynamicIncludeRegistry) {
        dynamicIncludeRegistry.register("/html/common/themes/top_head.jsp#pre");
    }

}
```

Note that the link's `href` attribute's value `/o/my-custom-dynamic-include/` is provided by the OSGi module's `Web-ContextPath` (`/my-custom-dynamic-include` in the example).

Now you know how to use the `top_head.jsp` dynamic includes.

Related Topics

Bottom JSP Dynamic Includes
    Top JS Dynamic Include
    WYSIWYG Editor Dynamic Includes

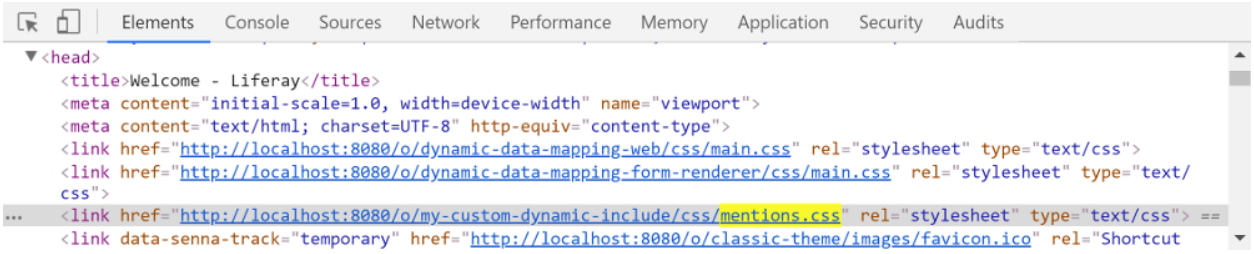


Figure 57.1: The top_head pre key loads additional CSS and HTML resources in the head of the theme.

57.3 Top JS Dynamic Include

The `top_js.jspf` dynamic include adds additional JavaScript files to the theme's head. For example, you can use this extension point to include a JS library that you need present in the theme's head:

```
/html/common/themes/top_js.jspf#resources
```

The example below injects a JavaScript file into the top of the `top_js.jspf`:
*`DynamicInclude` Java Class:

```
@Component(immediate = true, service = DynamicInclude.class)
public class JSTopHeadDynamicInclude extends BaseDynamicInclude {

  @Override
    public void include(
            HttpServletRequest request, HttpServletResponse response,
            String key)
        throws IOException {

    PrintWriter printWriter = response.getWriter();

    String content = "<script charset=\"utf-8\" src=\"/o/my-custom-dynamic-include/my_example_javascript.js\" async />";

    printWriter.println(content);
    }

    @Override
    public void register(
        DynamicInclude.DynamicIncludeRegistry dynamicIncludeRegistry) {

        dynamicIncludeRegistry.register(
      "/html/common/themes/top_js.jspf#resources"
    );
    }
}
```

Note that the JavaScript `src` attribute's value `/o/my-custom-dynamic-include/...` is provided by the OSGi module's `Web-ContextPath` (`/my-custom-dynamic-include` in the example).

Now you know how to use the `top_js.jspf` dynamic include.

Related Topics

Bottom JSP Dynamic Includes
Top Head JSP Dynamic Includes
WYSIWYG Editor Dynamic Includes

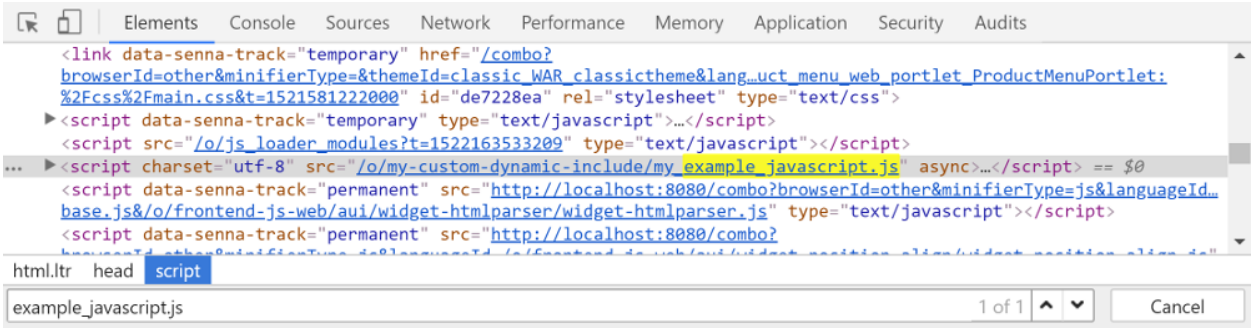


Figure 57.2: The Top JS dynamic include lets you load additional scripts in the theme's head.

57.4 Bottom JSP Dynamic Includes

The `bottom.jsp` dynamic includes load additional HTML or scripts in the bottom of the theme's body. The following keys are available:

Load additional HTML or scripts in the bottom of the theme's body, before the existing ones:

```
/html/common/themes/bottom.jsp#pre
```

Alternatively, load HTML or scripts in the bottom of the theme's body, after the existing ones:

```
/html/common/themes/bottom.jsp#post
```

The example below includes an additional script for the Simulation panel in the bottom of the theme's body, after the existing ones.

`SimulationDeviceDynamicInclude` Java class:

```
@Component(immediate = true, service = DynamicInclude.class)
public class SimulationDeviceDynamicInclude extends BaseDynamicInclude {

    @Override
    public void include(
            HttpServletRequest request, HttpServletResponse response,
            String key)
        throws IOException {

        PrintWriter printWriter = response.getWriter();

        printWriter.print(_TMPL_CONTENT);
    }

    @Override
    public void register(DynamicIncludeRegistry dynamicIncludeRegistry) {
        dynamicIncludeRegistry.register("/html/common/themes/bottom.jsp#post");
    }

    private static final String _TMPL_CONTENT = StringUtil.read(
        SimulationDeviceDynamicInclude.class,
        "/META-INF/resources/simulation_device_dynamic_include.tmpl");

}
```

`simulation_device_dynamic_include.tmpl`:

```
<script type="text/javascript">
    // <![CDATA[
        AUI().use(
            'aui-base',
            function(A) {
                var frameElement = window.frameElement;

                if (frameElement && frameElement.getAttribute('id') === 'simulationDeviceIframe') {
                    A.getBody().addClass('lfr-has-simulation-panel');
                }
            }
        );
    // ]]>
</script>
```

When the Simulation panel is open, the script adds the `lfr-has-simulation-panel` class to the theme's body.

Figure 57.3: You can use the bottom JSP dynamic include to inject scripts.

Now you know how to use the `bottom.jsp` dynamic includes.

Related Topics

Top Head JSP Dynamic Includes
    Top JS Dynamic Include
    WYSIWYG Editor Dynamic Includes

CHAPTER 58

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 Service Builder to implement your app's business logic.

These tutorials shows you how to use Service Builder to generate your persistence framework and implement your business logic. They also demonstrate using Spring in your app.

58.1 What is Service Builder?

Liferay Service Builder is a model-driven code generation tool that lets you 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. This tutorial explains some of the main benefits of using Service Builder:

- Integration with Liferay DXP
- 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 DXP uses Service Builder to generate all of its internal database persistence code. In fact, the services, both local and remote, are generated by Service Builder. Additionally, the service modules are generated by Service Builder. These things demonstrate Service Builder to be a robust and reliable tool. It is easy to use and can save *lots* of development time. Although the number of files Service Builder generates can seem intimidating at first, you only need to work with a few files to customize to your applications and add business logic.

Note: You don't have to use Service Builder to develop applications on Liferay DXP. It's entirely possible to develop them 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 DXP itself.

Each entity Service Builder generates 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 code that makes the remote services accessible. The remote services Service Builder generates 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 to make 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 you, Service Builder hides the complexities of using these technologies. You can take advantage of Dependency Injection (DI), Aspect Oriented Programming (AOP), and Object-Relational Mapping (ORM) in your 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 generates *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 let you, for example, 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 supports generating these kinds of simple finder methods but also 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 Liferay Clustering 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 lets you define custom SQL queries in an XML file and implement custom finder methods to run the queries. This is 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.

Note: Liferay includes a Service Builder library and build tool-specific plugins such as the Gradle Service Builder plugin, which includes Liferay's Service Builder library as a dependency. Service Builder supports Liferay 7's modular application development style of keeping API and implementation code in separate modules. The Service Builder sample apps demonstrate this.

In summary, we encourage you to use Service Builder for application development because it's a proven solution used throughout Liferay DXP and Liferay DXP applications. It generates distinct model, persistence, and service layers, local and remote services, Spring and Hibernate configurations, and related infrastructure without requiring any manual intervention. It also lets you generate basic SQL queries and finder methods and methods that filter results, taking Liferay permissions into account. Service Builder also supports entity and query caching. Each of these features saves 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 lets you add custom SQL queries and finder methods and it also supports dynamic query.

CHAPTER 59

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 Service Builder to quickly define your entities and generate the layer instantaneously.

Here's what these tutorials cover:

- Defining an object-relational map and generating your persistence layer from that map
- Running Service Builder
- Understanding and using local and remote services Service Builder generates
- Using the `ServiceContext` class
- Customizing model entities with Model Hints
- SQL queries
- Using Hibernate's criteria API
- Configuring `service.properties`
- Connecting Service Builder to external data sources

Start with defining an object-relational map.

CHAPTER 60

DEFINING AN OBJECT-RELATIONAL MAP WITH SERVICE BUILDER

In this tutorial, you'll learn how to define an object relational map so your application can persist data. As an example, you'll examine the existing Liferay Bookmarks application that uses Service Builder.

The Bookmarks application bookmarks 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.

The Bookmarks application's source code resides 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.

This tutorial assumes your application has these types of modules :

- `*-api`: Service interfaces
- `*-service`: Service implementations
- `*-web`: Portlet and controller

The parent folder of these modules is the *application folder*. The Service Builder project template is available for creating the `*-api` and `*-service` modules. Client UI project templates such as the MVCPortlet template are available for creating the `*-web` module. You can create projects from both templates using either Dev Studio DXP or Blade.

The first step in using Service Builder is to define your model classes and their attributes in a `service.xml` file. This file typically resides in the `*-service` module's root folder, although you can configure your build tool to recognize it in other folders. Service Builder terminology calls model classes *entities*. For example, the Bookmarks application has two entities: `BookmarksEntry` and `BookmarksFolder`. The requirements for each 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 Dev Studio DXP makes it easy to define entities in your application's `service.xml` file. Follow these steps:

1. Create the `service.xml` file.
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.

Each step is explained in detail. Start with creating a `service.xml` file.

60.1 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 liferay-service-builder_7_1_0.dtd specifies the format and requirements of the XML to use. You can create your `service.xml` file manually, or you can use Liferay Dev Studio DXP. build the `service.xml` file piece-by-piece, taking the guesswork out of creating XML that adheres to the DTD.

If you created your project from the Blade or Dev Studio DXP template, you have a `service.xml` file in your `*-service` module's root folder with an `entity` element named *Foo*. Remove the entire `<entity name="Foo" ...> ... </entity>` element: it's just 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's root folder and open the file. Liferay Dev Studio DXP provides a Diagram mode and a Source mode to give you different perspectives of the service information in your `service.xml` file.

- **Diagram mode** facilitates 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.

Next, you'll specify your service's global information.

60.2 Defining Global Service Information

A service's global information applies to all its entities, so it's a good place to start. In Liferay Dev Studio DXP, 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 to enter your service's global information. The fields include the service's

- Package path

- Namespace options
- Multiversion concurrency control
- Author

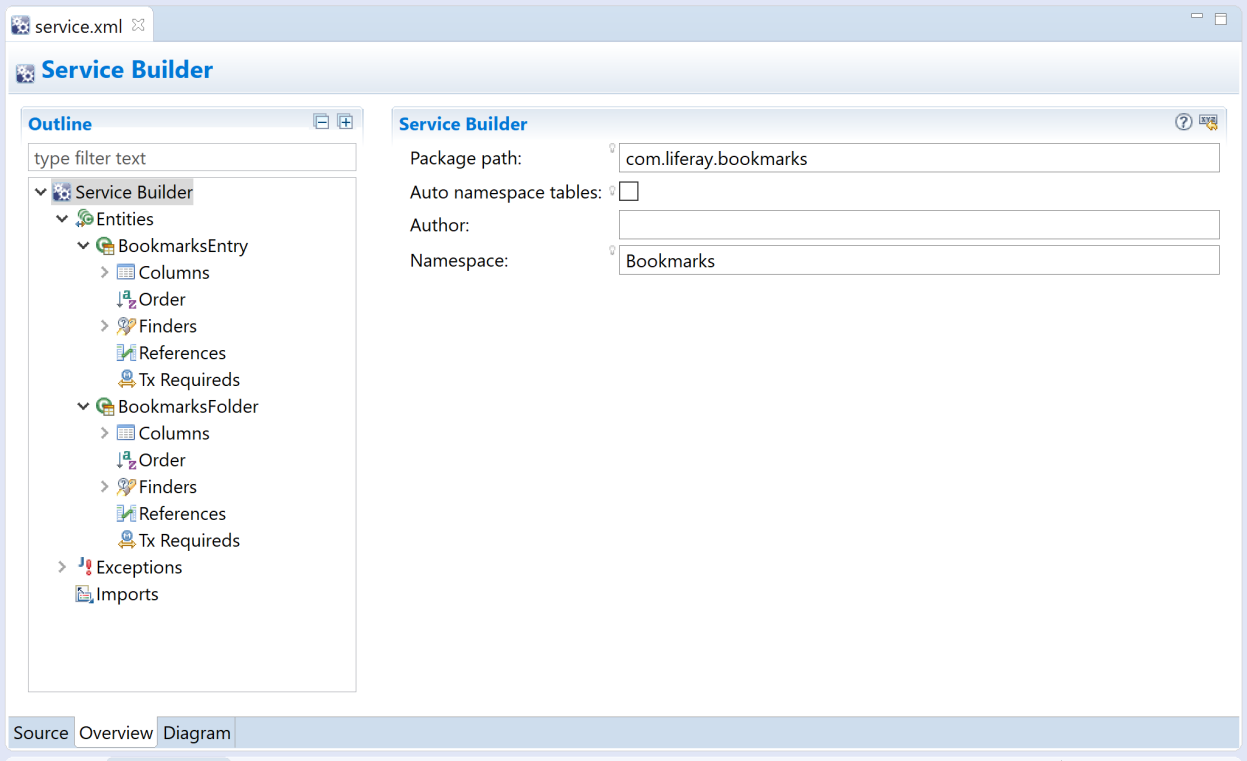


Figure 60.1: This is the Service Builder form from the Bookmarks application's `service.xml`.

Package Path

The package path specifies the package in which the service and persistence classes are generated. The package path for Bookmarks ensures that the `*-api` module's service classes are generated in the `com.liferay.bookmarks` package. The persistence classes are generated in a package of the same name in the `*-service` module. For example, examine the Bookmarks application's bookmarks-api and bookmarks-service modules to see how these are automatically generated for you. A later tutorial describes the package content.

Namespace Options

Service Builder uses the service namespace in naming the database tables it generates for the service. For example, *Bookmarks* could serve as the namespace for a Bookmarks application service.

```
<namespace>Bookmarks</namespace>
```

Service Builder uses the namespace in the following SQL scripts it generates in your `src/main/resources/sql` folder:

- indexes.sql
- sequences.sql
- tables.sql

Note: The generated SQL script folder location is configurable. For example, if you're using Gradle, you can define the sqlDir setting in the project's Gradle build.gradle file or Maven pom.xml file, the same way the databaseNameMaxLength setting is applied in the examples below.

Service Builder uses the SQL scripts to create database tables for all the entities the service.xml defines. The database table names have the namespace prepended when they are created. Since the example namespace value is Bookmarks, the database table names created for the entities start with Bookmarks_ as their prefix. Each Service Builder project's namespace must be unique. Separate plugins should use separate namespaces and should not use a namespace already used by Liferay entities (such as Users or Groups). Check the table names in Liferay's database to see the namespaces 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>
```

Multiversion concurrency control (MVCC)

The service-builder element's mvcc-enabled attribute is false by default. Setting mvcc-enabled="true" (hint: edit service.xml in *Source* view) enables multiversion concurrency control (MVCC) for all of the service's entities. In systems, concurrent updates are common. Without MVCC people may read or overwrite data from an invalid state unknowingly. With MVCC, each modification is made upon a given base version number. When Hibernate receives the update, it generates an update SQL statement that uses a where clause to make sure the current data version is the version you expect.

If the current data version

- **matches the expected version**, your data operation is based on up-to-date data and is accepted.
- **doesn't match the expected version**, the data you're operating on is outdated. Liferay DXP rejects your data operation and throws an exception, which you can catch to help the user handle the exception (e.g., suggest retrying the operation).

Important: Enable MVCC for all your services by setting `mvcc-enabled="true"` in your `<service-builder/>` element. When invoking service entity updates (e.g., `fooService.update(object)`), make sure to do so in transactions. Propagate rejected transactions to the UI for the user to handle.

Author

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 the Java classes and interfaces it generates. Save your `service.xml` file. Next, you'll add entities for your services.

60.3 Defining Service Entities

Entities are the heart and soul of a service. They represent the map between the model objects in Java and your database fields and tables. Service Builder maps the entities you define 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 `BookmarksEntry` entity information:

- **Name:** *BookmarksEntry*
- **Local service:** *yes*
- **Remote service:** *yes*

And here's what is used for the `BookmarksFolder` entity:

- **Name:** *BookmarksFolder*
- **Local service:** *yes*
- **Remote service:** *yes*

Here are steps to create entities using Liferay Dev Studio DXP:

1. In the outline on the left side of the `service.xml` editor in Overview mode, select the *Entities* node under the Service Builder node. In the main part of the view, notice that the Entities table is empty.

2. Create an entity by clicking on the *Add Entity* icon ( ) to the right of the table.

3. Name your entity and mark whether to generate local and remote services for it.

Add as many entities as you need.

The entity's database table name includes the entity name prefixed with the namespace. The Bookmarks example creates one database table named `Bookmarks_BookmarksEntry` and another named `Bookmarks_BookmarksFolder`.

Setting *Local Service* (the `local-service` attribute) to `true` instructs Service Builder to generate local interfaces for the entity's services. Local services are set to `false` by default. Local services can only be invoked from the Liferay server on which they're deployed.

Setting *Remote Service* (the `remote-service` attribute) to `true` instructs Service Builder to generate remote interfaces for the service. Local services are set to `true` by default. You can build a fully-functional application without generating remote services. In that case, you could set your entity

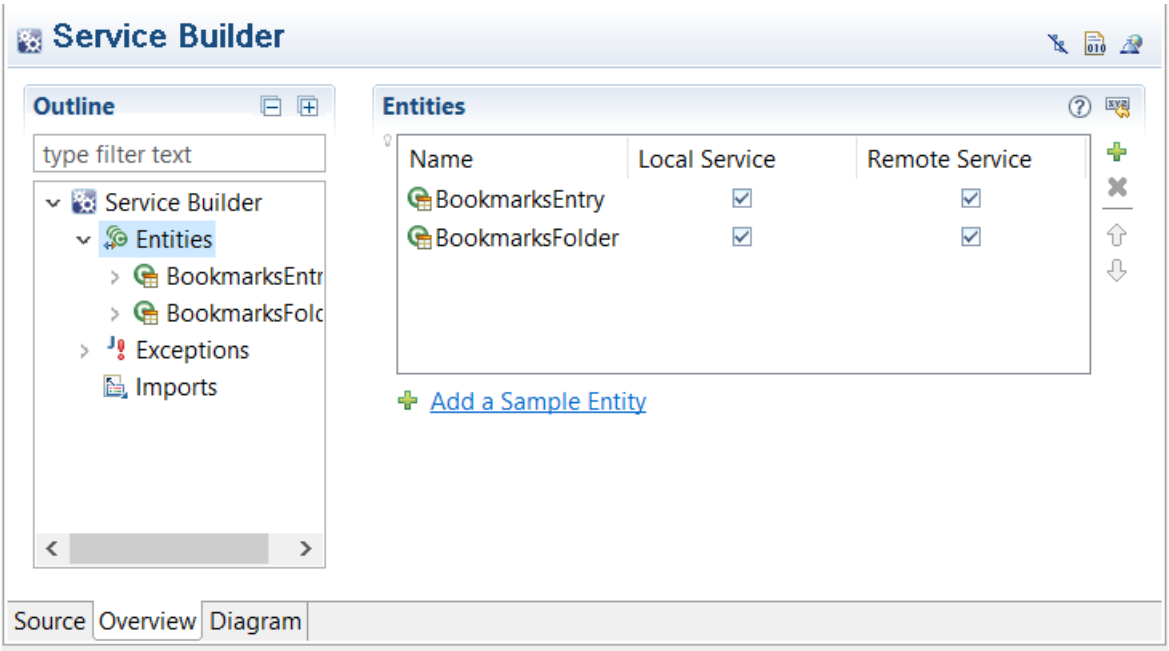


Figure 60.2: Adding service entities in your `service.xml` file is easy with Liferay Dev Studio DXP's *Overview* mode.

local services to `true` and remote services to `false`. If, however, you want to enable remote access to your application's services, set both local service and remote service to `true`.

Tip: Suppose you have an existing Data Access Object (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*.

60.4 Defining the Columns (Attributes) for Each Service Entity

An entity's columns represent its attributes. These attributes map table fields to Java object fields. To add attributes for your entity, 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 Dev Studio DXP 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

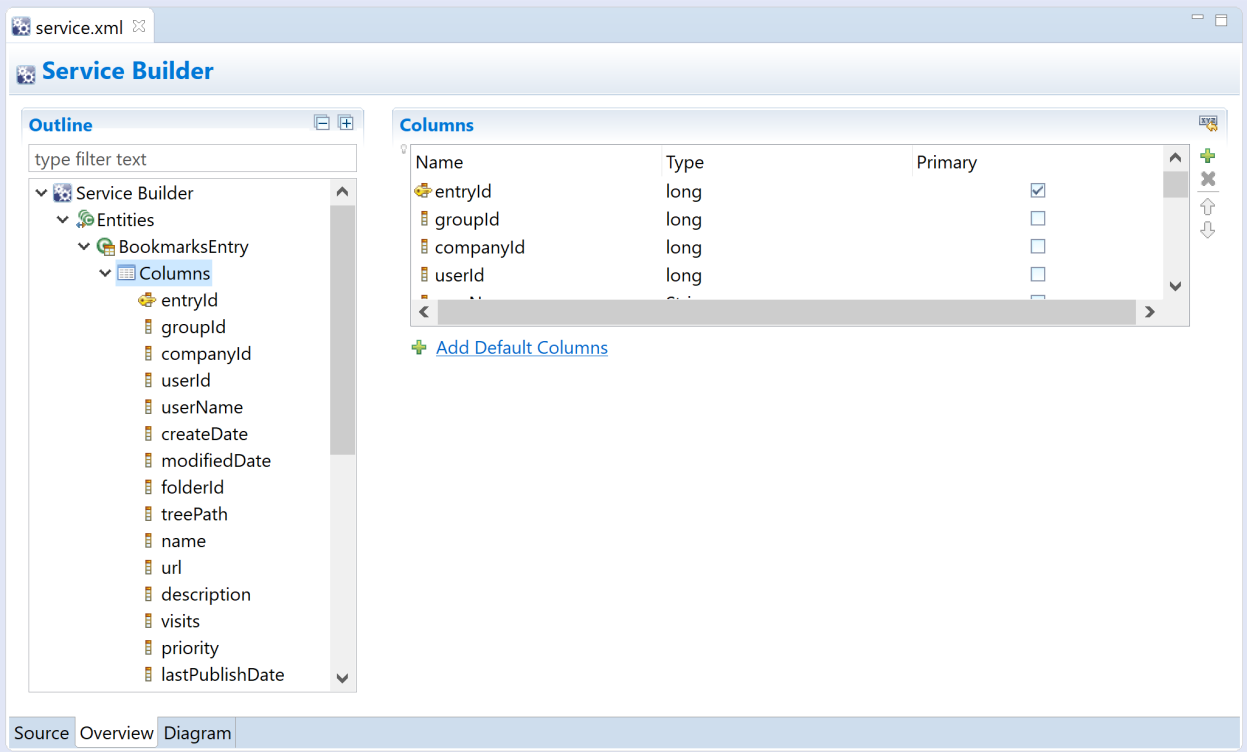


Figure 60.3: Liferay Dev Studio DXP facilitates defining table columns for entities.

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 uniquely identifies 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 Bookmarks application. 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: The Implementing an Add Method article demonstrates how to generate unique primary keys for entity instances.

Create Entity Columns

Similar to the way you used the form table for adding entities, add attribute columns for each of your entities.

1. Create each attribute by clicking on the Add icon ( ).
2. Fill in the attribute's name
3. Select the attribute's type. 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.
4. Specify whether the attribute is a primary key for the entity.

Note: On deploying a *`service` module, Service Builder automatically generates indexes for all entity primary keys.

Create a column for each attribute of your entity or entities.

Support Multi-tenancy

In addition to columns for your entity's primary key and attributes, add portal instance ID and site ID columns. They let your portlet support Liferay's multi-tenancy features, 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`. 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

Track Ownership

To track each entity instance's owner, add a column called `userId` of type `long`.

User column

Name	Type	Primary
`userId`	long	no

Audit Entities

Lastly, you can add columns to help audit your entities. For example, you could create a column named `createDate` of type `Date` to note an entity instance's creation date. 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 have 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.

60.5 Defining Relationships Between Service Entities

Referencing one type of entity in the context of another entity is a common requirement. This is also known as *relating* 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` facilitates relating entities.

1. Select Diagram mode for the `service.xml` file.
2. 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.
3. Click your first entity and move your cursor over to the entity you'd like to relate it with. Liferay Dev Studio DXP draws a dashed line from your selected entity to the cursor.
4. Click the second entity to complete drawing the relationship. Liferay Dev Studio DXP turns the dashed line into a solid line, with an arrow pointing to the second entity.
5. Save the `service.xml` file.

Congratulations! You've related two entities. Their relationship shows in Diagram mode and looks similar to the relationship in the figure below.

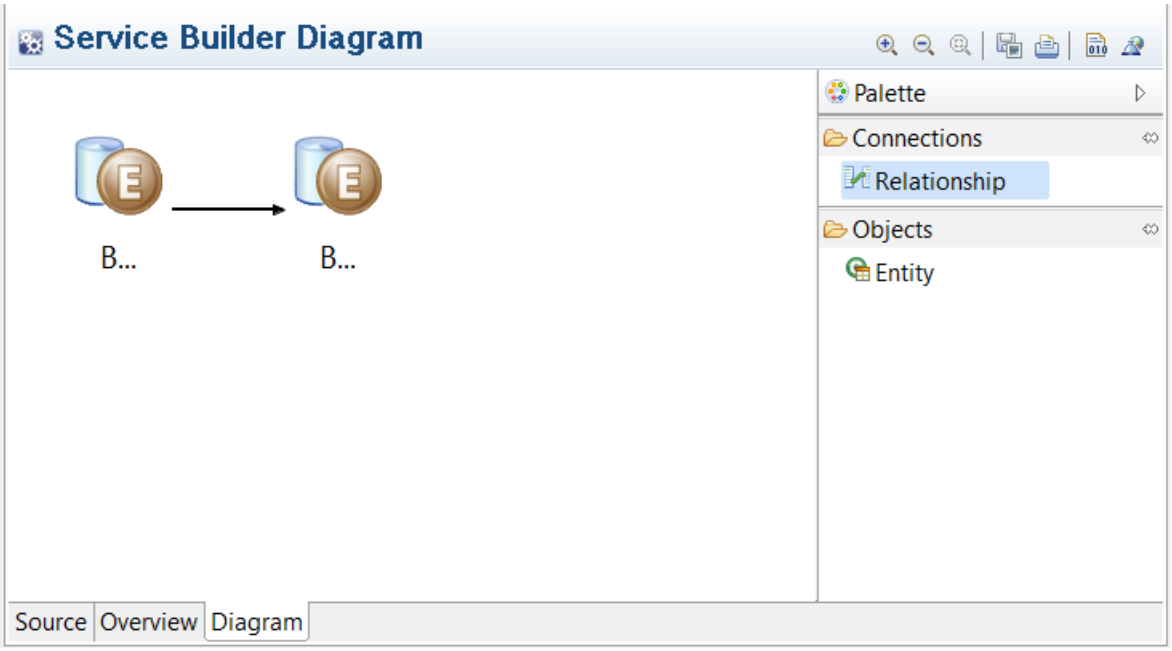


Figure 60.4: Relating entities is a snap in Liferay Dev Studio DXP's *Diagram* mode for `service.xml`.

Switch to *Source* mode in the editor for your `service.xml` file and note that Liferay Dev Studio DXP created a column element in the first selected entity to hold the ID of the corresponding entity instance reference. For example, the `BookmarksEntry` entity uses this column to relate to a `BookmarksFolder` entity :

```
<column name="folderId" type="long" />
```

Now that your entity columns are in place and entity relationships are established, you can specify the default order in which the entity instances are retrieved from the database.

60.6 Defining Ordering of Service Entity Instances

Often, you want to retrieve multiple instances of a given entity and list them in a particular order. The `service.xml` file lets you specify the default order of your entities.

Suppose you want to return `BookmarksEntry` entities alphabetically by name. It's easy to specify these default orderings using Liferay Dev Studio DXP.

1. Switch back to *Overview* mode in the `service.xml` file editor.
2. 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.
3. Check the *Specify ordering* checkbox to show the form for specifying the ordering.
4. Create an order column by clicking the *Add* icon ( ) to the right of the table.
5. Enter the column name (e.g., *name*, *date*, etc.) to use in ordering the entity.
6. 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.

60.7 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 Dev Studio DXP's Overview mode for the `service.xml` file.

Create Finders

Here are the steps for creating a finder node:

1. 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.
2. Create a new finder by clicking the *Add* icon ( ) to the right of the table.
3. Specify your finder's 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.

Create Finder Columns

Under the new finder node, Dev Studio DXP created a *Finder Columns* node. Here are the steps for creating finder columns:

1. Select the *Finder Columns* node to specify the columns for your finder's parameters.
2. Create a new finder column by clicking the *Add* icon and specifying the column's name. Keep in mind that you can specify multiple finder parameters (columns).
3. Save your `service.xml` file.

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.

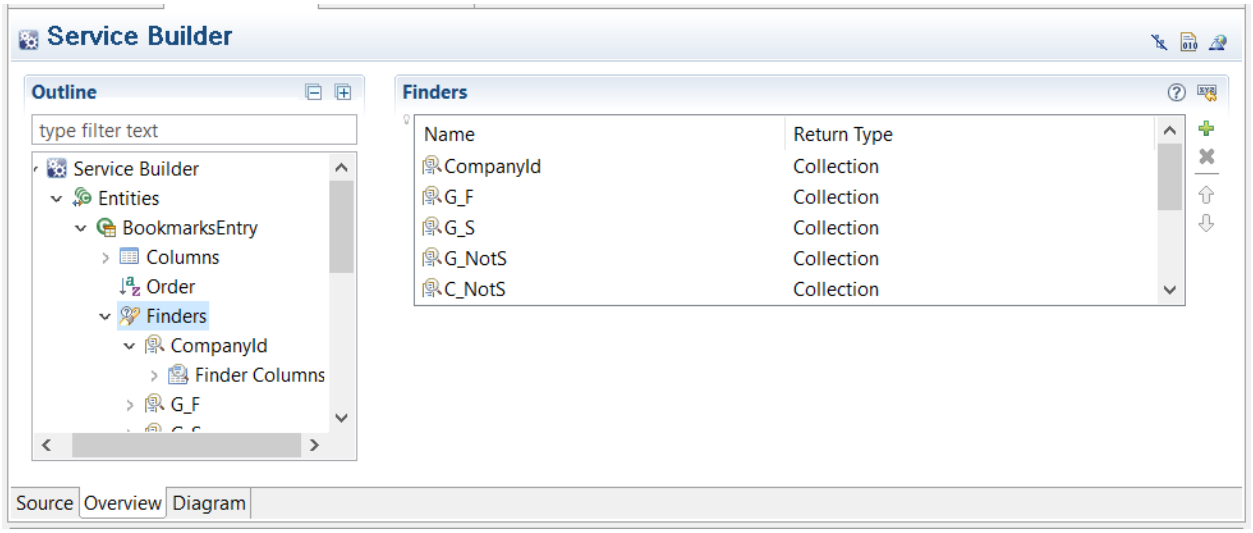


Figure 60.5: Creating Finder entities is easy with Liferay Dev Studio DXP.

If you're creating site-scoped entities (entities whose data should be unique to each site), follow the steps described above to create finders by `groupId` for retrieving your entities.

Service Builder generates finder-related methods (e.g., `fetchByGroupId`, `findByGroupId`, `removeByGroupId`, `countByGroupId`) for the your entities in the `*Persistence` and `*PersistenceImpl` classes. 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. It's time to run Service Builder and examine the code it generates.

60.8 Running Service Builder

This tutorial demonstrates how to run Service Builder. If want to use Service Builder in your application but haven't yet created a `service.xml` file that defines an object-relational map for you application, make sure to do so before proceeding with this tutorial.

There are two ways to build services from a `service.xml` file:

- Liferay Dev Studio
- Command line

Liferay Dev Studio DXP is demonstrated first.

Using Liferay Dev Studio

From the Package Explorer, right-click on your service module and then select *Liferay → build-service*.

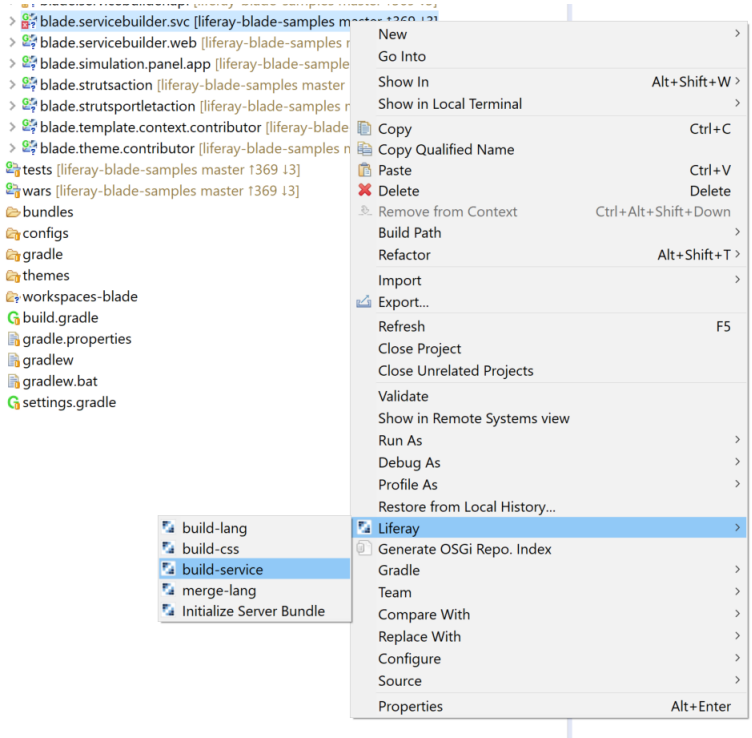


Figure 60.6: Liferay Dev Studio DXP facilitates building Service Builder services via the *build-service* option in the service module's *Liferay* submenu.

Service Builder generates plenty of files. You can run Service Builder from the command line too.

Using the Command Line

Open a command line and navigate to your application folder (the folder that contains your `*-api` and `*-service` modules).

To build your services using Gradle, enter the following command:

```
blade gw buildService
```

or

```
gradlew buildService
```

Blade's `gw` command works in any project that has a Gradle Wrapper available to it–projects generated using Liferay project templates have a Gradle Wrapper.

Note: Liferay Workspace's Gradle Wrapper script is in the workspace root folder. If your application project folder is located at `[workspace]/modules/[application]`, for example, the Gradle Wrapper is available at `../../gradlew`.

If you're using Maven, build the services by running the following command:

```
mvn service-builder:build
```

Important: 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. More information is available on using Maven to run Service Builder.

On successfully building the services, Service Builder prints the message `BUILD SUCCESSFUL`. Many generated files appear in your project. They represent a model layer, service layer, and persistence layer for your entities. Don't worry about the number of generated files–they're explained in the next tutorial. Its time to review the code Service Builder generates for your entities.

60.9 Understanding the Code Generated by Service Builder

Service Builder generates code to support your entities. 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 your application's `*-api` and `*-service` modules in the packages you specified in `service.xml`. For example, here are the package paths for Liferay's Bookmarks application:

- `/bookmarks-api/src/main/java/com/liferay/bookmarks`
- `/bookmarks-service/src/main/java/com/liferay/bookmarks`

The `bookmarks-api` module's interfaces define the Bookmarks application API. The `*-api` module interfaces define the application's persistence layer, service layer, and model layer. Whenever you compile and deploy the `*-api` module, all its classes and interfaces are packaged in a `.jar` file called `PROJECT_NAME-api.jar` in the module's `build/libs` folder. Deploying this JAR to Liferay *defines* the API as OSGi services.

The `bookmarks-service` module classes implement the `bookmarks-api` module interfaces. The `*-service` module provides the OSGi service implementations to deploy to Liferay's OSGi framework.

Next, examine the classes and interfaces generated for the entities you specified. Similar classes are generated for each entity, depending on how each entity is specified in the `service.xml`. Here are the three types of customizable classes:

- `*LocalServiceImpl`
- `*ServiceImpl`
- `*Impl`

The * represents the entity name in the classes listed above.
Here are the persistence, service, and model classes:

- 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 the `[ENTITY_NAME]` class by referencing it with the `@Reference` annotation.

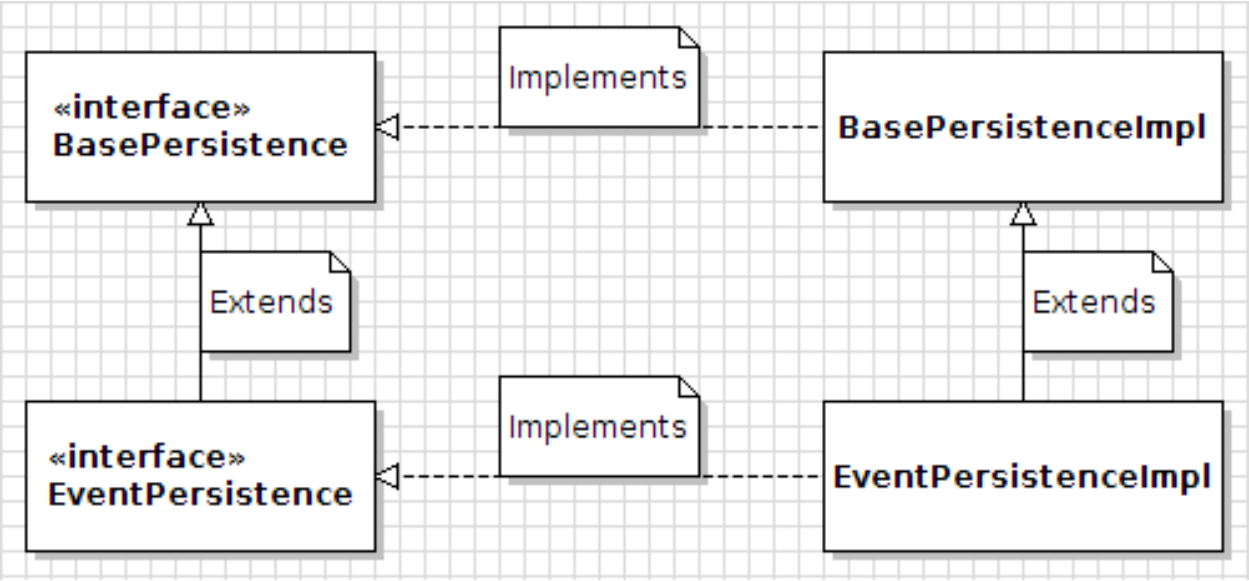


Figure 60.7: Service Builder generates these persistence classes and interfaces for an example entity called *Event*. You shouldn't (and you won't need to) customize any of these classes or interfaces.

- Local Service (generated for an entity only if the 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 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`. This class is generated for backwards compatibility purposes only. Use the `*LocalService` class by referencing it with the `@Reference` annotation.
- `[ENTITY_NAME]LocalServiceWrapper`: Local service wrapper which implements `[ENTITY_NAME]LocalService`. This class is designed to be extended and it lets you customize the entity's local services.

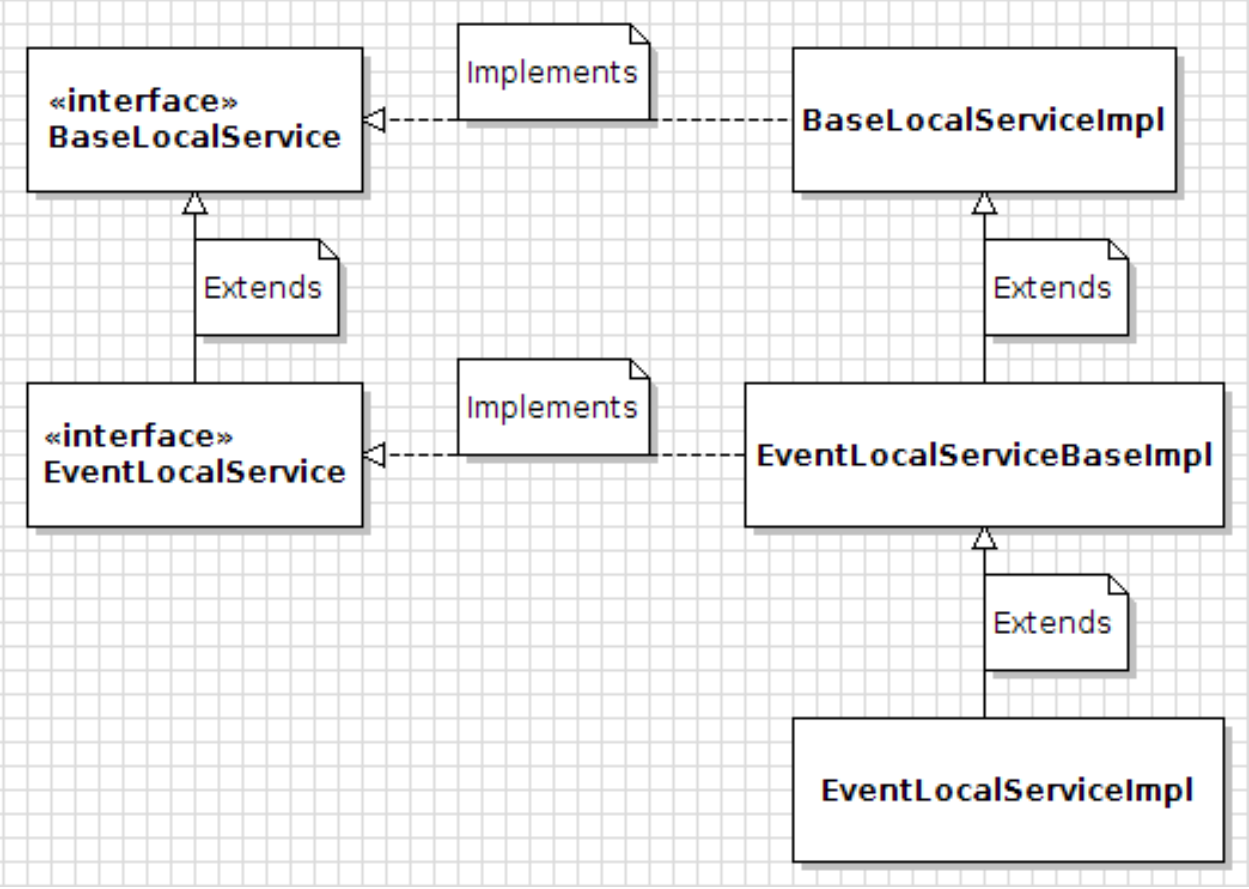


Figure 60.8: Service Builder generates these service classes and interfaces. Only the [ENTITY_NAME]LocalServiceImpl (e.g., EventLocalServiceImpl for the Event entity) allows custom methods to be added to the service layer.

- 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. 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. This class is generated for backwards compatibility purposes only. Use the *Service class by referencing it with the @Reference annotation.
- [ENTITY_NAME]ServiceWrapper: Remote service wrapper which implements [ENTITY_NAME]Service. This class is designed to be extended and it lets you customize the entity's remote 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 Service Builder generates. 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]. This class is designed to be extended and it lets you customize the entity.

Note: *Util classes are generated for backwards compatibility purposes only. Your module applications should avoid calling the util classes. Use the non-util classes instead–you can reference them using the @Reference annotation.

Each file that Service Builder generates is assembled from an associated FreeMarker template. The FreeMarker templates are in the com.liferay.portal.tools.service.builder module's /com/liferay/portal/tools/service/builder/dependencies/ folder. For example, Service Builder uses the service_impl.ftl template to generate the *ServiceImpl.java classes.

You can modify any *Impl class Service Builder generates. The most common are *LocalServiceImpl, *ServiceImpl and *Impl. If you manually modify the other classes, Service Builder overwrites the changes the next time you run it. Whenever you add methods to, remove methods from, or change a method signature of a *LocalServiceImpl class, *ServiceImpl

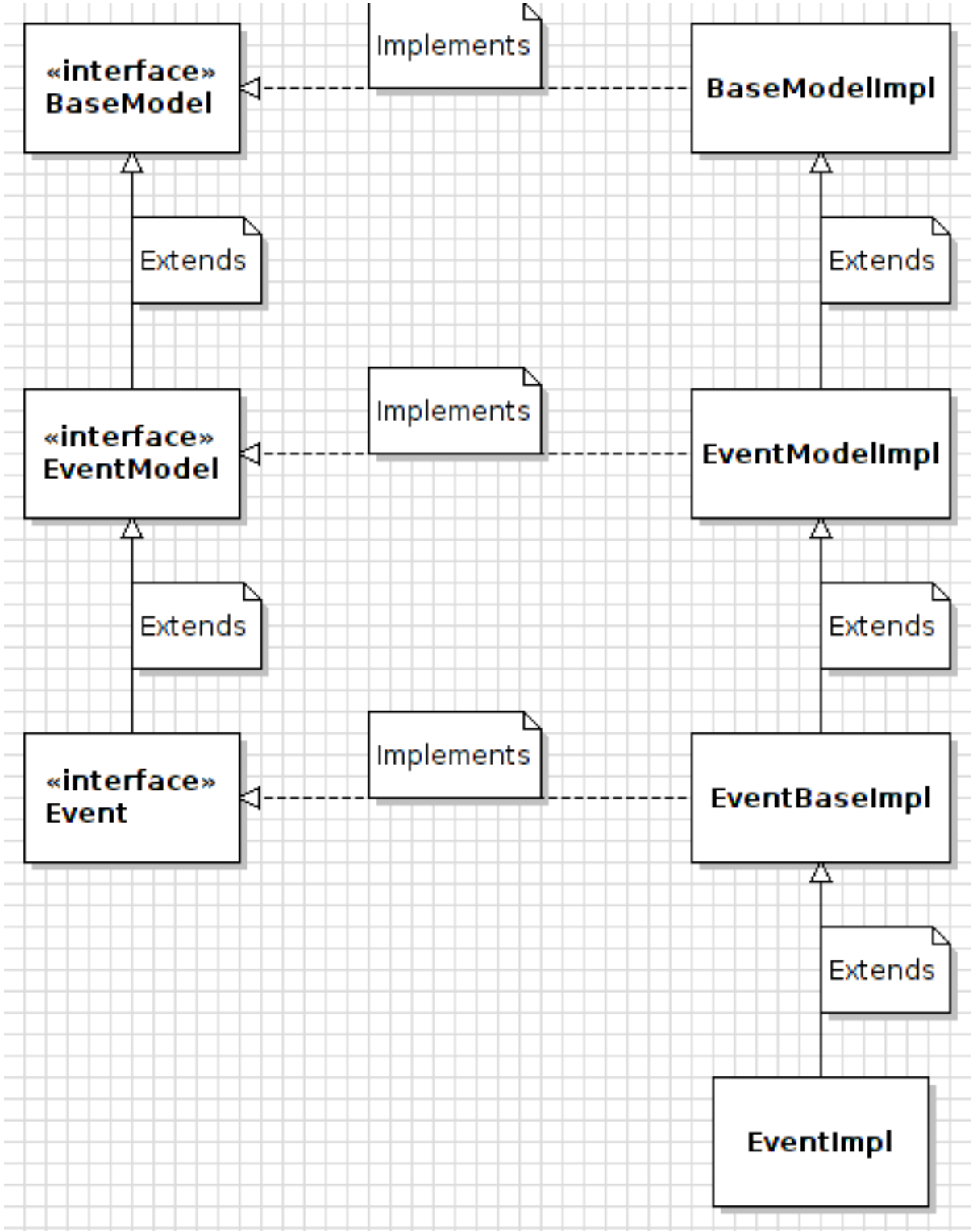


Figure 60.9: Service Builder generates these model classes and interfaces. Only `[ENTITY_NAME]Impl` (e.g., EventImpl for the Event entity) allows custom methods to be added to the service layer.

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
Understanding Service Context
Creating Local Services

60.10 Iterative Development

As you develop an application, you might 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. **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. The Maven DB Support Plugin reference article explains how to run this command from a Maven project.

Use the first option if you have a released application and you must preserve user data. Use the second option if you're adding new columns during development.

Related Topics

Upgrade Processes
Gradle DB Support Plugin
Maven DB Support Plugin

60.11 Understanding ServiceContext

The `ServiceContext` class holds contextual information for a service. It aggregates information necessary for features used throughout Liferay's portlets, such as permissions, tagging, categorization, and more. This tutorial covers the following `ServiceContext` class topics:

- Service Context Fields
- Creating and Populating a Service Context in Java
- Creating and Populating a Service Context in JavaScript
- Accessing Service Context Data

The `ServiceContext` fields are first.

Service Context Fields

The `ServiceContext` class has many fields. The `ServiceContext` class Javadoc describes them. Here's a categorical listing of some commonly used Service Context 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 data with scope must at least specify the scope group ID. Here's an example of creating a `ServiceContext` instance and passing it as a parameter to a Liferay service API:

```
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 class with 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 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.

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.

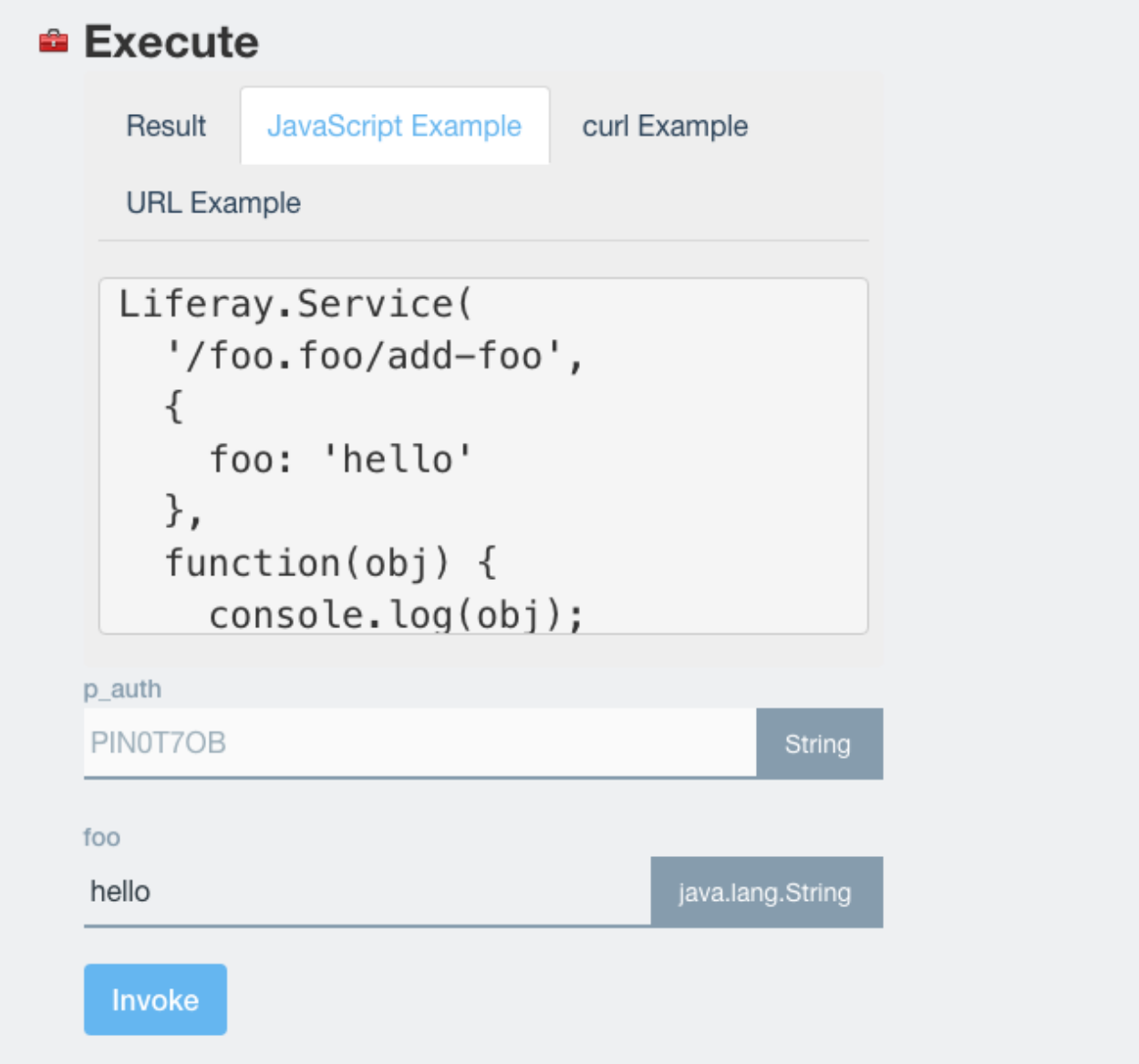


Figure 60.10: When you invoke a service from Liferay's JSON web services page, you can view the result of your service invocation as well as example code for invoking the service via JavaScript, curl, or URL.

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 entities with scope must 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

60.12 Customizing Model Entities With Model Hints

Once you've used Service Builder to define model entities, you may want to further refine how users enter that data. For example, model hints can define a calendar field where only future dates should be selectable. Model hints provide a single place to specify entity data restrictions and other formatting.

You define model hints in a file called portlet-model-hints.xml. If your project has an API module and a service module like the Service Builder project template, portlet-model-hints.xml goes in the service module's src/main/resources/META-INF folder. For example, in Liferay's Bookmarks application, the portlet-model-hints.xml file is in the bookmarks-service/src/main/resources/META-INF/ folder.

Model hints define two things:

1. How entities are presented to users
2. The size of database columns

As Liferay renders your form fields, it customizes the form's input fields based your configuration.

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; changes to them are overwritten when Service Builder runs. You can, however, safely edit the `portlet-model-hints.xml` file.

As an example, consider the Bookmarks app service module'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="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`. 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 class name. Models have `field` elements representing their entity's columns. Lastly, `field` elements must have a name and a type. Each `field` element's name and type maps to the name

and type specified for the entity's column in the service module's `service.xml` file. Service Builder generates all these elements for you, based on the `service.xml`.

To add hints to a field, add a `hint` child element. For example, you can add a `display-width hint` to specify the pixel width to use in displaying 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, run Service Builder again and redeploy your module. 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.

Model Hint Types

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: The aui taglib is fully supported and not related to AlloyUI (the JavaScript library) that's deprecated.

Note: You can use a mix of Clay and aui tags in a form. Model hints, however, affect aui tags only.

Note that Liferay has its own model hints file–`portal-model-hints.xml`. It's in `portal-impl.jar`'s `META-INF` folder. This file contains many hint examples, so you can reference it when creating `portlet-model-hints.xml` files.

Default Hints

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 pixels to each field:

```
<default-hints>
    <hint name="display-width">300</hint>
</default-hints>
```

Hint Collections

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="EMAIL-ADDRESS">
    <hint name="max-length">254</hint>
</hint-collection>
<hint-collection name="HOSTNAME">
    <hint name="max-length">200</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>
```

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!

60.13 Configuring service.properties

This tutorial explains how to use and edit the `service.properties` file. It also tells you about the properties 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 this file's properties to alter your service's database schema. You should not modify this file directly, but rather make any necessary property overrides in a `service-ext.properties` file in that same folder.

Here are some of the properties the `service.properties` file includes:

- `build.namespace`: This is the namespace you defined in your `service.xml`. Liferay distinguishes different modules from each other using their namespaces.
- `build.number`: Liferay distinguishes your module's different Service Builder builds. Each time you deploy a distinct Service Builder build to Liferay, Liferay increments this number.
- `build.date`: This is the time of your module's latest Service Builder build.
- `include-and-override`: The default value of this property defines `service-ext.properties` as an override file for `service.properties`.

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.

The Build Auto Upgrade feature is now different and is set in a global property `schema.module.build.auto.upgrade` in the file `[Liferay_Home]/portal-developer.properties`. To learn more, see the tutorial Upgrading Data Schemas in Development.

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

60.14 Connecting Service Builder to External Databases

Sometimes you want to use a database other than Liferay DXP's. To do this, its data source must be defined in `portal-ext.properties` or configured as a JNDI data source on the app server. This tutorial shows how to connect Service Builder to a data source. Here's how:

1. Specify the database and data source in your `service.xml`.
2. Create the Database Manually
3. Define the Data Source
4. Create a Spring bean that points to the data source.
5. Set your entity's data source to the `liferayDataSource` alias.
6. Run Service Builder.

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.

Step 1: Specify Your Database and a Data Source Name in Your `service.xml`

In your `service.xml` file, specify the same arbitrary data source name for all of the entities, a unique table name for each entity, and a database column name for each column. Here's an example:

```
<?xml version="1.0"?>
<!DOCTYPE service-builder PUBLIC "-//Liferay//DTD Service Builder 7.1.0//EN"
    "http://www.liferay.com/dtd/liferay-service-builder_7_1_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.

Step 2: Create the Database Manually

Create the database per the database specification in your service.xml.

Next, use portal properties to set your data source.

Step 3: Specify the Data Source

If the 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'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
```

Step 4: Create a Spring Bean that Points to the Data Source

To do this, create a parent context extension (e.g.,ext-spring.xml) in your Service Builder module's src/main/resources/META-INF/spring/parent folder or in your traditional portlet's WEB-INF/src/META-INF/parent folder. Create this folder if it doesn't exist already.

Note: Since Liferay DXP 7.1 Fix Pack 3 (included in Service Pack 1) and Liferay Portal 7.1 CE GA2, the Spring extender uses two application contexts for Service Builder *-service modules. This lets Liferay DXP register extender services earlier and separately from the Service Builder services and allows disabling features in the parent application context that may no longer be needed in the future. For details, see LPS-85683.

If you're using a prior version of Liferay DXP 7.1, put your parent context extension (e.g.,ext-spring.xml) 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:

1. 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>
```

2. A Liferay data source bean that refers to the data source factory Spring bean.
3. An alias for the Liferay 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 is in a `portal-ext.properties` file, the bean requires 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.

Step 5: Set Your Entity's Data Source to the `liferayDataSource` Alias

In your `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 7.1.0//EN"
    "http://www.liferay.com/dtd/liferay-service-builder_7_1_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.

Step 6: Run Service Builder

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.

Congratulations! You've connected Service Builder to your external data source.

Related Topics

Connecting to JNDI Data Sources
Service Builder
Running Service Builder and Understanding the Generated Code
Business Logic with Service Builder

60.15 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. `CustomSQL` class (from module `com.liferay.portal.dao.orm.custom.sql.api`) 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.i` 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 have it implement the `*Finder` interface you just generated:

```
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 = _customSQL.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";

@ServiceReference(type=CustomSQL.class)
private CustomSQL _customSQL;
```

The custom finder method opens a new Hibernate session and uses Liferay's `CustomSQL.get(Class<?> clazz, String id)` method to get the custom SQL to use for the database query. The `FIND_BY_ENTRYNAME_ENTRYMESSAGE_GUESTBOOKNAME` 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.

Note: Liferay Portal 6.2 made finder methods accessible via static `*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 a JSP in your web module. For example, to call the finder method in the Guestbook application, just call `entryLocalService.findByEntryNameEntryMessageGuestbookName(...)`!

Congratulations on developing a custom SQL query and custom finder for your application!

Related Topics

Customizing Liferay Services
    Service Builder Web Services

CHAPTER 61

DYNAMIC QUERY

Though you can use custom SQL queries with Service Builder to retrieve data from the database, sometimes it's more convenient to build queries dynamically at runtime. You can do this with Liferay's Dynamic Query API, which wraps Hibernate's Criteria API. The Dynamic Query API lets you build queries without writing any SQL. It helps you think in terms of objects and member variables instead of tables and columns. Complex queries 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 Dynamic Queries, you still must take care to construct efficient queries. For information on Hibernate's Criteria API, please see Hibernate's manual. This tutorial demonstrates creating custom finders for Liferay applications using Service Builder and 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. This tutorial assumes that you're creating a Liferay application consisting 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, follow these steps to call custom finders using the Dynamic Query API:

1. Define a custom finder method.
2. Implement your finder using the Dynamic Query API.
3. Add a method to your `*LocalServiceImpl` class that invokes your finder method.

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, you'll examine these steps in more detail.

61.1 Defining a Custom Finder Method

Dynamic queries belong in finder methods. You implement them and then make them available through an interface. This tutorial demonstrates defining the finder method in an implementation class, generating its interface and tying the implementation to the interface.

An example of this is a Guestbook application with two entities: guestbook and entry. Each entry belongs to a guestbook so the entry entity has a `guestbookId` field as a foreign key. If you need a finder to search for guestbook entries by entry name and guestbook name, you'd add a finder method to `GuestbookFinderImpl` and name it `findByEntryNameGuestbookName`. The full method signature would be `findByEntryNameGuestbookName(String entryName, String guestbookName)`. The steps are below.

1. Create a `[Entity]FinderImpl` class in the `[package path].service.persistence.impl` package of your service module's `src/main/java` folder. Recall that you specify the `[package path]` in your `service.xml` file. Here's an example:

   ```
   <service-builder package-path="com.liferay.docs.guestbook">
       ...
   </service-builder>
   ```

2. Define a `findBy*` finder method in the class you created. Make sure to add any required arguments to your finder method signature.

3. Run Service Builder to generate the appropriate interface in the `[package path].service.persistence` package in the `service` folder 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.GuestbookFinder` is generated.

4. Make the finder class a component (annotated with `@Component`) that implements the finder interface. For example, the class declaration should look like this:

   ```
   @Component(service = GuestbookFinder.class)
   public class GuestbookFinderImpl extends BasePersistenceImpl<Guestbook> implements GuestbookFinder
   ```

Your next step is to implement the query in your finder method using the Dynamic Query API.

61.2 Implementing a Custom Finder Method Using Dynamic Query

Once you've defined your custom finder method, you can use the Dynamic Query API to implement your query in it. Here's what you must do in your finder method:

1. Open a Hibernate Session

2. Create a dynamic query using these Hibernate features:

   - *Restrictions*: Similar to `where` clauses of an SQL query, restrictions limit results based on criteria.
   - *Projections*: Modify the kind of results the query returns.
   - *Orders*: Organize results.

3. Execute the Dynamic Query and return the results

Before implementing a dynamic query in your own finder method, it can be helpful to examine an example. The following example method uses multiple dynamic queries and all the Hibernate features. Instructions for implementing your own finder method follow the example.

Example Finder Method: findByGuestbookNameEntryName

This finder method for the Guestbook application retrieves a list of Guestbook entries that have a specific name and that also belong to a Guestbook of a specific name:

```
public List<Entry> 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<Entry> entries = _entryLocalService.dynamicQuery(entryQuery);

        return entries;
    }
    catch (Exception e) {
        try {
            throw new SystemException(e);
        }
        catch (SystemException se) {
            se.printStackTrace();
        }
    }
    finally {
        closeSession(session);
    }
}
```

The method first opens a Hibernate session. While the session is open in the `try` block, it creates and executes a dynamic query, which returns results (a list of guestbook `Entry` objects) if all goes well.

The finder method has two distinct dynamic queries.

1. The first query retrieves a list of guestbook IDs corresponding to guestbook names that match the `guestbookName` parameter of the finder method.
2. 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 entry query only needs the guestbook IDs.

Next appears an order:

```
Order order = OrderFactoryUtil.desc("modifiedDate");
```

This arranges the results list 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 entryQuery = DynamicQueryFactoryUtil.forClass(Entry.class, classLoader))
    .add(RestrictionsFactoryUtil.eq("name", entryName))
    .add(PropertyFactoryUtil.forName("guestbookId").in(guestbookQuery))
    .addOrder(order);
```

By default, DynamicQueryFactoryUtil.forClass(Entry.class, classLoader)) returns a 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 finder method's entryName parameter. PropertyFactoryUtil is a Liferay utility class whose method forName(String propertyName) 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 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 the dynamic query API to create complex queries, similar to SQL joins.

Lastly, the order defined earlier is applied 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.

Lastly, the dynamic query is invoked on the EntryLocalService instance. It returns a list of Entry objects which are then returned by the finder method.

```
List<Entry> entries = _entryLocalService.dynamicQuery(entryQuery);

return entries;
```

It's time to implement your finder method to use Dynamic Query. Start with opening and managing a Hibernate session.

Using a Hibernate Session

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 structure of your finder method should look like this:

```
public List<Entity> findBy-(...) {

    Session session = null;
    try {
            session = openSession();

            /*
            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
    */

}
```

Next, in the try block, create your dynamic query objects.

Creating Dynamic Queries

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));
```

In your finder method, initialize your dynamic query for your entity class.

Most features of Hibernate's Criteria API, including restrictions, projections, and orders, can be used on Liferay dynamic query objects. Each criteria can be applied to your query. The restriction criteria type is described first.

Restriction Criteria

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.

When you need to add restrictions to a dynamic query, don't call Hibernate's `Restrictions` class directly. Instead, use the `RestrictionsFactoryUtil` service. `RestrictionsFactoryUtil` has the same methods that you're used to from Hibernate's `Restrictions` class: `in`, `between`, `like`, `eq`, `ne`, `gt`, `ge`, `lt`, `le`, etc.

Thus, instead of using the following call to specify that a guestbook must have a certain name,

```
entryCriteria.add(Restrictions.eq("name", guestbookName));
```

you use

```
entryQuery.add(RestrictionsFactoryUtil.eq("name", guestbookName));
```

The restriction above limits the results to guestbook entries whose `name` attribute matches the value of the variable `guestbookName`. Add the restrictions you need to get the results you want.

Projections are the next criteria type. They let you transform the query results to return the field type you desire.

Projection Criteria

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 to return only a list of the values of a certain entity field, or fields. 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.

Similarly, to set projections, 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"));
```

The projection above specifies the `guestbookId` entity field to changes the result set to a list of those field values. If you want to return a specific field type from your entities, add a projection for it.

The last criteria type lets you organize results your way.

Order Criteria

Orders in Hibernate's Criteria API let you control the order of the elements in the list a query returns. You can choose the property or properties to which an order applies as well as whether they're in ascending or descending order.

This code creates an order by the entity's `modifiedDate` attribute:

```
Order order = OrderFactoryUtil.desc("modifiedDate");
```

When you apply this order, the results 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.

Like Hibernate criteria, Liferay's dynamic queries are *chain-able*: 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 a restriction criterion and a projection to a dynamic query object declaration:

```
DynamicQuery guestbookQuery = DynamicQueryFactoryUtil.forClass(Guestbook.class)
    .add(RestrictionsFactoryUtil.eq("name", guestbookName))
    .setProjection(ProjectionFactoryUtil.property("guestbookId"));
```

It's time to execute your dynamic query.

Executing the Dynamic Query

In the previous tutorial, you ran Service Builder after defining your custom finder. Service Builder generated a `dynamicQuery(DynamicQuery dynamicQuery)` method in your `*LocalServiceBaseImpl` class. Using a `*LocalService` instance, invoke `dynamicQuery` method, passing it your dynamic query. Here's an example dynamic query execution.

```
List<Entity> entities = _someLocalService.dynamicQuery(entityQuery);

return entities;
```

The dynamic query execution returns a list of entities and the finder method returns that list.

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 defining custom finder methods and either modify your finder method or create overloaded versions of it 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: `dynamicQuery(entryQuery)`, or `dynamicQuery(entryQuery, start, end)` or `dynamicQuery(entryQuery, start, end, orderByComparator)`.

Great! You've now created a finder method using Liferay's Dynamic Query API. Your last step is to add a service method that calls your finder.

61.3 Accessing Your Custom Finder Method from the Service Layer

So far, you've created a `*FinderImpl` class, defined a `findBy*` finder method in that class, and implemented the finder method using Dynamic Query. Now how do you call your finder method from the service layer?

When you ran Service Builder, the *Finder interface was generated (e.g., GuestbookFinder). For proper separation of concerns, only a local or remote service implementation (i.e., *LocalServiceImpl or *ServiceImpl) in your service module should invoke the *Finder class. 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.

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!

Related Topics

Service Builder Web Services
    Creating Local Service
    Invoking Local Services

61.4 Actionable Dynamic Queries

Suppose you have over a million users, and you want to perform some kind of mass update to some 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, retrieving large numbers of Service Builder entities using dynamic queries requires too much memory and time.

Liferay actionable dynamic queries solve this problem. Actionable dynamic queries use a pagination strategy to load only small numbers of entities into memory at a time and perform processing (i.e., perform 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. This way, you can still process your million users, but only small numbers are loaded into memory at a time.

Here's how to use actionable dynamic query:

1. Get an `ActionableDynamicQuery` from your `*LocalService` by invoking its `getActionableDynamicQuery` method.
2. Add query criteria and constraints, using the query's `setAddCriteriaMethod` and `setAddOrderCriteriaMethod` methods.
3. Set an action to perform on the matching entities, using `setPerformActionMethod`.
4. Execute the action on each matching entity, by invoking the query's `performActions` method.

This method from a sample portlet creates an actionable dynamic query, adds a query restriction and an action, and executes the query:

```
protected void massUpdate() {
    ActionableDynamicQuery adq = _barLocalService.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);
            _barLocalService.updateBar(bar);
        }

    });

    try {
        adq.performActions();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

The example method demonstrates executing an actionable dynamic query on `Bar` entities that match certain criteria.

1. Retrieve an `ActionableDynamicQuery` from local service `BarLocalService`.

```
ActionableDynamicQuery adq = _barLocalService.getActionableDynamicQuery();
```

```
**Note:** Service Builder generates method `getActionableDynamicQuery()` in
each entity's `*LocalService` interface and implements it in each entity's
`*BaseLocalServiceImpl` class.

    @Transactional(propagation = Propagation.SUPPORTS, readOnly = true)
    public ActionableDynamicQuery getActionableDynamicQuery();
```

---

2. Set query criteria to match `field3` values less than `100`.

```
adq.setAddCriteriaMethod(new ActionableDynamicQuery.AddCriteriaMethod() {

     @Override
     public void addCriteria(DynamicQuery dynamicQuery) {
         dynamicQuery.add(RestrictionsFactoryUtil.lt("field3", 100));
     }

 });
```

3. Set an action to perform. The action increments the matching entity's `field3` value.

```
adq.setPerformActionMethod(new ActionableDynamicQuery.PerformActionMethod<Bar>() {

     @Override
     public void performAction(Bar bar) {
         int field3 = bar.getField3();
         field3++;
         bar.setField3(field3);
         _barLocalService.updateBar(bar);
     }

 });
```

4. Execute the action on each matching entity.

```
try {
    adq.performActions();
}
catch (Exception e) {
    e.printStackTrace();
}
```

Actionable dynamic queries let you act on large numbers of entities in smaller groups. It's an efficient and high performing way to update entities.

Related Topics

Service Builder Web Services
    Creating Local Service
    Invoking Local Services

CHAPTER 62

Business Logic with Service Builder

Once you've defined your application's entities and run 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.

CHAPTER 63

CREATING LOCAL SERVICES

The heart of your service is its `*LocalServiceImpl` class. This class is your entity's local service extension point. Local services are invoked from your application or by other applications running on the same instance as your application.

These tutorials walk you through creating local services:

1. Deciding to Create Local and Remote Services.
2. Implementing the `add` Method.
3. Implementing the `update` and `delete` Methods.
4. Implementing Finder and Counter Methods
5. Implementing Other Business Logic
6. Integrating with Liferay's Services.

Start with deciding the service types you need.

63.1 Deciding to Create Local and Remote Services

Defining your object model involves choosing whether to generate local and or remote service interfaces. Local services can only be invoked from the Liferay server on which they're deployed. Remote services are accessible to clients outside of the Liferay server. Before implementing local or remote services, consider the best practices described here:

1. If you plan to have remote services, enable local services too.
2. Implement your business logic in `*LocalServiceImpl`.
3. Create corresponding remote services methods in your `*ServiceImpl`.
4. Use the remote service methods to call the local service, wrapping the calls in permission checks.

5. In your application, call only the remote services. This ensures that your service methods are secured and that you don't have to duplicate permissions code.

If you are turning on local or remote services in your `service.xml` file just now, make sure to run Service Builder again to generate the service interfaces.

63.2 Implementing an Add Method

Your `*LocalServiceImpl` operates on the entities your `service.xml` defines. The first method to implement, therefore, is one that creates entities. Liferay's convention is to implement this in an `add*` method, where the part after `add` is the entity name (or a shortened version of it). Here are the steps for implementing an `add*` method:

1. Declare an `add*` method with parameters for creating the entity.
2. Validate the parameters.
3. Generate a primary key.
4. Create an entity instance.
5. Populate the entity attributes.
6. Persist the entity.
7. Return the entity instance.

This tutorial refers to the Bookmarks application's `addEntry` method from `BookmarksEntryLocalServiceImpl` as an example. To keep things simple, we have excluded the code that integrates with Liferay services, such as assets, social bookmarks, and more.

Here's the Bookmarks application's `addEntry` method:

```
public BookmarksEntry addEntry(
        long userId, long groupId, long folderId, String name, String url,
        String description, ServiceContext serviceContext)
    throws PortalException {

    // Entry

    User user = userLocalService.getUser(userId);

    if (Validator.isNull(name)) {
        name = url;
    }

    validate(url);

    long entryId = counterLocalService.increment();

    BookmarksEntry entry = bookmarksEntryPersistence.create(entryId);

    entry.setUuid(serviceContext.getUuid());
    entry.setGroupId(groupId);
    entry.setCompanyId(user.getCompanyId());
    entry.setUserId(user.getUserId());
    entry.setUserName(user.getFullName());
    entry.setFolderId(folderId);
```

```
    entry.setTreePath(entry.buildTreePath());
    entry.setName(name);
    entry.setUrl(url);
    entry.setDescription(description);
    // More assignments ...

    bookmarksEntryPersistence.update(entry);

    // Integrate with more Liferay services here ...

    return entry;
}
```

This method uses the parameters to create `BookmarksEntry`. It validates the parameters, creates an entry with a generated entry ID (primary key), populates the entry, persists the entry, and returns it. You can refer to this method as you create your own `add*` method.

Step 1: Declare an add method with parameters for creating the entity

Create a public method for *adding* (creating) your application's entity. Make it a public method that returns the entity it creates.

```
public [ENTITY] add[ENTITY](...) {

}
```

For example, here's the `addEntry` method signature:

```
public BookmarksEntry addEntry(
        long userId, long groupId, long folderId, String name, String url,
        String description, ServiceContext serviceContext)
    throws PortalException {
        ...
}
```

This method specifies all the parameters needed to create and populate a `BookmarksEntry`. It throws a `PortalException` in case the parameters are invalid or a processing exception occurs (more on this in a later step).

Your add method must specify parameters that satisfy the entity's attributes specified in your `service.xml` file. Make sure to account for primary keys of other related entities. For example, the `addEntry` method above includes a parameter `long folderId` to associate the new `BookmarksEntry` to a `BookmarksFolder`.

Step 2: Validate the parameters

Validate the parameters as needed. You might need to make sure a parameter is not empty or `null`, or that a parameter value is within a valid range. Throw a `PortalException` or an extension of `PortalException` for any invalid parameters.

For example, the `addEntry` method invokes the following `validate` method to check if the URL parameter is `null`.

```
protected void validate(String url) throws PortalException {
    if (!Validator.isUrl(url)) {
        throw new EntryURLException();
    }
}
```

Next, generate a primary key for the entity instance you're creating.

Step 3: Generate a primary key

Every entity instance needs a unique primary key. Liferay's `CounterLocalService` generates them per entity. Every `*BaseLocalServiceImpl` has a `counterLocalService` field that references a `CounterLocalService` object for the entity. Invoke the counter service's `increment` method to generate a primary key for your entity instance.

```
long id = counterLocalService.increment();
```

Now you have a unique ID for your entity instance.

Step 4: Create an entity instance

The `*Peristence` instance associated with your entity has a `create(long id)` method that constructs an entity instance with the given ID. Every `*BaseLocalServiceImpl` has a `*Persistence` field that references a `*Persistence` object for the entity. For example, `BookmarksEntryLocalServiceImpl` can access `BookmarksEntryLocalServiceBaseImpl`'s field `bookmarksEntryPersistence`, which is a reference to a `BookmarksEntryPersistence` instance.

```
@BeanReference(type = BookmarksEntryPersistence.class)
protected BookmarksEntryPersistence bookmarksEntryPersistence;
```

`BookmarksEntryLocalServiceImpl`'s `addEntry` method creates a `BookmarksEntry` instance using this call:

```
BookmarksEntry entry = bookmarksEntryPersistence.create(entryId);
```

To create an instance of your entity, invoke the `create` method on the `*Persistence` field associated with the entity, making sure to pass in the entity primary key you generated in the previous step.

```
 [ENTITY_NAME] entity = [ENTITY_NAME]Persistence.create(id);
```

It's time to populate the new entity instance.

Step 5: Populate the entity attributes

Use the `add*` method parameter values and the entity's setter methods to populate your entity's attributes. For example, here are the `BookmarksEntry` attribute assignments:

```
entry.setUuid(serviceContext.getUuid());
entry.setGroupId(groupId);
entry.setCompanyId(user.getCompanyId());
entry.setUserId(user.getUserId());
entry.setUserName(user.getFullName());
entry.setFolderId(folderId);
entry.setTreePath(entry.buildTreePath());
entry.setName(name);
entry.setUrl(url);
entry.setDescription(description);
```

Note that the `ServiceContext` is commonly used to carry an entity's UUID and the `User` is associated to a company.

Step 6: Persist the entity

It's time to store the entity. Invoke the *Persistence field's update method, passing in the entity object. For example, here's how the new BookmarksEntry is persisted:

```
bookmarksEntryPersistence.update(entry);
```

Your entity is persisted for the application.

Step 7: Return the entity

Finally, return the entity you just created so the caller can use it.

Run Service Builder to propagate your new service method to the *LocalService interface.

You've implemented your local service's add* method to create and persist your application's entities.

63.3 Implementing update and delete Methods

After you've implementing an add* method for creating service entities, you'll want to create update* and delete* methods for updating and deleting them. They're easy to implement. The main difference between them and the add* method is they must know which entity they're updating or deleting.

Implementing an update method

An update* method for a local service resembles an add* method most because it has parameters for setting entity attribute values. Create an update* method this way:

1. Declare an update* method with parameters for updating the entity.
2. Validate the parameters.
3. Retrieve the entity instance, if necessary.
4. Update the entity attributes.
5. Persist the updated entity.
6. Run Service Builder.

The following code snippets from BookmarksEntryLocalServiceImpl's updateEntry method are helpful to examine.

```
public BookmarksEntry updateEntry(
        long userId, long entryId, long groupId, long folderId, String name,
        String url, String description, ServiceContext serviceContext)
    throws PortalException {

    // Entry

    BookmarksEntry entry = bookmarksEntryPersistence.findByPrimaryKey(
        entryId);

    if (Validator.isNull(name)) {
```

```
        name = url;
    }

    validate(url);

    entry.setFolderId(folderId);
    entry.setTreePath(entry.buildTreePath());
    entry.setName(name);
    entry.setUrl(url);
    entry.setDescription(description);
    //...

    bookmarksEntryPersistence.update(entry);

    // Integrate with more Liferay services here ...

    return entry;
}
```

This method has all the makings of a good `update*` method:

- parameter for looking up the entity instance
- parameters for updating the entity attributes
- parameter validation
- entity attribute updates
- entity persistence
- returns the entity instance

Refer to the example method above as you follow the steps to create your own `update*` method.

Step 1: Declare an update method with parameters for updating the entity

Create a public method for updating your application's entity.

```
public [ENTITY] update[ENTITY](...)
    throws PortalException {

}
```

Replace `[ENTITY]` with your entity's name or nickname. Create a parameter list that satisfies the entity attributes you're updating. Include an entity instance parameter or an ID parameter for fetching the entity instance.

For example, the `BookmarksEntryLocalServiceImpl`'s `updateEntry` method signature has an ID parameter (`entryId`) for fetching the `BookmarksEntry` entity instance. Also it has parameters `folderId`, `name`, `url`, and `description` for updating the `BookmarksEntry`'s respective attributes.

```
public BookmarksEntry updateEntry(
        long userId, long entryId, long groupId, long folderId, String name,
        String url, String description, ServiceContext serviceContext)
    throws PortalException {...}
```

Note, user ID, group ID, and service context parameters are useful for integrating with Liferay's services. More on that later.

Step 2: Validate the parameters

Similar to validating the `add*` method parameters, validate your `update*` parameters. Your `add*` and `update*` methods might be able to use the same validation code. Throw a `PortalException` or an extension of `PortalException` for any invalid parameters.

Step 3: Retrieve the entity instance

If you're passing in an entity instance, you can update it directly. Otherwise, pass in the entity ID (the primary key). The *Persistence class Service Builder injects into *BaseLocalServiceImpl classes has a findByPrimaryKey(long) method that retrieves instances by ID. For example, the BookmarksEntryLocalServiceImpl retrieves the BookmarksEntry that matches the primary key entryId.

```
BookmarksEntry entry = bookmarksEntryPersistence.findByPrimaryKey(
    entryId);
```

Invoke the findByPrimaryKey(long id) method of your *Persistence class to retrieve the entity instance that matches your primary key parameter.

```
[ENTITY] entity = [ENTITY]Persistence.findByPrimaryKey(id);
```

It's time to update the entity attributes.

Step 4: Update the entity attributes

Invoke the entity's setter methods to replace its attribute values.

Step 5: Persist and return the updated entity instance

Persist the updated entity to the database and return the instance to the caller.

```
[ENTITY]Persistence.update(entity);

...

return entity;
```

Step 6: Run Service Builder

Finally, run Service Builder to propagate your new service method to the *LocalService interface.

You've created a service method to update your entity. If you thought that was easy, implementing a delete* method is even easier.

Implementing a delete method

The remove method of an entity's *Persistence class deletes an entity instance from the database. Use it in your local service's delete* method. Here's what a delete* method looks like:

```
public [ENTITY] delete[ENTITY](ENTITY entity) throws PortalException
{
    [ENTITY]Persistence.remove(entity);

    // Clean up related to additional Liferay services goes here ...

    return entity;
}
```

Make sure to replace [ENTITY] with your entity's name or nickname.

For example, here's paraphrased code from BookmarksEntryLocalServiceImpl's deleteEntry method:

```
public BookmarksEntry deleteEntry(BookmarksEntry entry)
    throws PortalException {

    bookmarksEntryPersistence.remove(entry);

    // Clean up related to additional Liferay services goes here ...

    return entry;
}
```

After implementing your `delete*` method, run Service Builder to propagate your new service method to the `*LocalService` interface.

Related Topics

Implementing an add method
    Implementing getter and counter methods
    Integrating with Liferay services

63.4 Implementing Methods to Get and Count Entities

Service Builder generates `findBy*` methods and `countBy*` methods in your `*Persistence` classes based on your `service.xml` file's finders. You can leverage finder methods in your local services to get and count entities.

- Getters: `get*` methods return entity instances matching criteria.
- Counters: `get*Count` methods return the number of instances matching criteria

Start with getting entities that match criteria.

Getter Methods

The `findByPrimaryKey` methods and `findBy*` methods search for and return entity instances based on criteria. Your local service implementation must only wrap calls to the finder methods that get what you want.

Here's how to create a method that gets an entity based on an ID (primary key):

1. Create a method using this format:

    ```
    public [ENTITY] get[ENTITY_NAME](long id) {
        return [ENTITY]Persistence.findByPrimaryKey(id);
    }
    ```

2. Replace `[ENTITY]` and `[ENTITY_NAME]` with the respective entity type and entity name (or nickname).

3. Run Service Builder to propagate the method to your local service interface.

Here's how to get entities based on criteria:

1. Identify the criteria for finding the entity instance(s).

2. If there is no `finder` element for the criteria, create one for it and run Service Builder.

3. Determine the `*Persistence` class `findBy*` method you want to call. Depending on your `finder` element columns, Service Builder might overload the method to include these parameters:

    - `int start` and `int end` parameters for specifying a range of entities.
    - `com.liferay.portal.kernel.util.OrderByComparator orderByComparator` parameter for arranging the matching entities.

4. Specify your `get*` method signature, making sure to account for the `*Persistence` class `findBy*` method parameters you must satisfy. Use this method format:

    ```
    public List<[ENTITY]> get[DESCRIBE_THE_ENTITIES](...) {

    }
    ```

    Replace `[ENTITY]` with the entity type. Replace `[DESCRIBE_THE_ENTITIES]` with a descriptive name for the entities you're getting.

5. Call the `*Persistence` class `findBy*` method and return the list of matching entities.

6. Run Service Builder.

For example, method `getGroupEntries` from `BookmarksEntryLocalServiceImpl` returns a range of `BookmarksEntrys` associated with a `Group` primary key:

```
public List<BookmarksEntry> getGroupEntries(
    long groupId, int start, int end) {

    return bookmarksEntryPersistence.findByG_S(
        groupId, WorkflowConstants.STATUS_APPROVED, start, end,
        new EntryModifiedDateComparator());
}
```

Of the `BookmarksEntrys` associated with workflows, this method only matches approved ones (`WorkflowConstants.STATUS_APPROVED`). It uses a new `EntryModifiedDateComparator` to order the matching `BookmarksEntrys` by modification date.

Now you know how to leverage finder methods to get entities. Methods that count entities are next.

Counter Methods

Counting entities is just as easy as getting them. Your `*Persistence` class `countBy*` methods do all the work. Service Builder generates `countBy*` methods based on each finder and its columns.

1. Identify the criteria for entity instances you're counting and determine the `*Persistence` class `countBy*` method that satisfies the criteria.

2. Create a `get*Count` method signature following this format:

    ```
    public int get[DESCRIBE_THE_ENTITIES]Count(...) {

    }
    ```

Replace [DESCRIBE_THE_ENTITIES] with a descriptive name for the entities you're counting.

3. Call the *Persistence class' countBy method and return the value. For example, the method getEntriesCount from BookmarksEntryLocalServiceImpl returns the number of BookmarksEntrys that have a workflow status of status and that are associated with a group (matching groupId) and a folder (matching folderId).

```
public int getEntriesCount(long groupId, long folderId, int status) {
    return bookmarksEntryPersistence.countByG_F_S(
        groupId, folderId, status);
}
```

Now your local service can get entities matching your criteria and return quick entity counts.

Related Topics

Creating Local Services
- Implementing an add method
- Defining Service Entity Finder Methods
- Understanding the Code Generated by Service Builder

63.5 Implementing Any Other Business Logic

This section's earlier local service tutorials focus on CRUD methods: methods that **c**reate (add), **r**ead (get), **u**pdate, and **d**elete entities. But you might also need methods that provide business logic.

For example, Liferay Bookmarks application users *open* bookmarks (navigate to a URLs) by clicking on them. BookmarksEntryLocalServiceImpl's openEntry method supports this functionality:

```
public BookmarksEntry openEntry(long userId, BookmarksEntry entry) {
    entry.setVisits(entry.getVisits() + 1);

    bookmarksEntryPersistence.update(entry);

    assetEntryLocalService.incrementViewCounter(
        userId, BookmarksEntry.class.getName(), entry.getEntryId(), 1);

    return entry;
}
```

The openEntry method tracks and persists the number of visits to the BookmarksEntry's URL, increments the number of views for the BookmarksEntry as an asset, and returns the BookmarksEntry. This method implements required business logic that compliments the CRUD methods.

Convenience methods might also be appropriate for your app. They're easier to use because they typically have these characteristics:

- Shorter parameter list
- Intuitive name

Short parameter lists are easier to satisfy, and methods that have intuitive names are easier to find in Javadoc.

For example, the Bookmarks application lets users move bookmarks to different folders. Moving a bookmark can be done using the service's `updateEntry(...)` method, but its long parameter list is overkill since all the operation requires is the bookmarks entry and the folder where it's going. Compare the following `update*` method call to a convenience method call.

Update method:

```
bookmarksEntryLocalService.updateEntry(userId, entryId, groupId, folderId, name, url, description, serviceContext);
```

Convenience method:

```
bookmarksEntryLocalService.moveEntry(entryId, folderId);
```

Convenience methods are typically straightforward to write. Here's the `moveEntry` method:

```
public BookmarksEntry moveEntry(long entryId, long parentFolderId)
    throws PortalException {

    BookmarksEntry entry = getBookmarksEntry(entryId);

    entry.setFolderId(parentFolderId);
    entry.setTreePath(entry.buildTreePath());

    bookmarksEntryPersistence.update(entry);

    return entry;
}
```

The `moveEntry` method retrieves the `BookmarksEntry` entity by its ID, assigns it a new parent folder, updates its tree path, persists all the entity's changes, and returns the entity. Convenience methods like this one facilitate updating a subset of the entity's attributes.

After implementing your custom business methods, run Service Builder to propagate them to the interface.

In your local services, you can implement business logic methods that suit your application.

Related Topics

Creating Local Services

Invoking Local Services

Invoking Services from Service Builder Code

63.6 Integrating with Liferay's Frameworks

New car buyers expect certain features to be standard: power windows, cruise control, floor mats (at least the cheap ones), and so on. Similarly, users expect applications to have certain features and expect those features to behave consistently across applications.

For example, a user might expect the app's content can be shared socially on Twitter and Facebook. The user might expect a way to tag and rate the app content, and a way to comment on it. You must meet, and even exceed, these expectations. Liferay's frameworks implement these features users know and love. Integrating with the frameworks is straightforward and the frameworks provide intuitive, consistent user experiences.

Here are some of Liferay's most popular frameworks:

- Permissions: Defines resource for entities and actions for performing on the resources.

- Configurable Applications: Makes applications configurable from within the Control Panel.
- Workflow: Equips entities for reviewing in workflows before publishing.
- Item Selector: Provides a consistent developer experience for browsing and selecting entities.
- Asset Framework: Makes entities more descriptive enabling users to tag, categorize, rate, prioritize, and comment on them. It enables users to relate entities to each other as assets and it allows the entities to be published in the Asset Publisher.

    - Tags and Categories: Enables users to tag entities and categorize the tagged entities.
    - Priority: Users can ascribe numerical priorities to entities.
    - Related Assets: Users can associate one entity with another as an asset.
    - Asset Renderer: Enables displaying entities in the Asset Publisher.
    - Comments: Lets users comment on entities.
    - Ratings: Enables rating systems, such as five stars or thumbs up/down, on entities.
    - Flags: Users can flag entity content as inappropriate.
    - *Social Bookmarks*: Users can share entity content on Twitter, Facebook, and more. (Tutorials are coming soon.)

- Export/Import: Lets users export entity data to and import entity data from files (`.lpkg` files). Exported data can be imported to another portal instance or saved for later use.
- Staging: Lets users change entities behind the scenes without affecting the live site.
- *Search*: Enables entities to be found and shown in Liferay DXP search results. (Tutorials are coming soon.)
- *Recycle Bin*: Entities can be moved from the application and put them into the Recycle Bin. Entities can be restored from the Recycle Bin or deleted permanently (manually or per a schedule). (Tutorials are coming soon.)

Liferay's frameworks are rich with features users expect in applications. Click on the framework tutorial links to start leveraging the frameworks. Next are tutorials on invoking local services.

Related Topics:

Internationalization
JavaScript Module Loaders
Front-End Taglibs
Data Upgrades

63.7 Invoking Local Services

Once you deploy your services module, those services are available in the container. Service Builder generates local and remote service classes as OSGi Declarative Services (DS) components. These components are accessible to other DS components, so you can invoke them from other components, such as your web application. Here's how:

1. Add a reference to the local service component.

2. Call the component's methods.

There's a Blade sample called Basic Service Builder. Its `basic-web` module has a `Portlet` service component that demonstrates referencing a local service component. This module also has JSPs that invoke the component's methods. Your first step is to add a reference to the local service component object.

Step 1: Reference the Local Service Component

Your application's Service Builder-generated local services are DS components that you can inject into your application's other DS components (classes annotated with `@Component`) using the `@Reference` annotation. The `basic-web` module's `JSPPortlet` class is a `Portlet` service component that references the `FooLocalService` local service as a DS component.

```
@Reference
private volatile FooLocalService _fooLocalService;
```

The OSGi service registry wires the service implementation object to your class that references it. The `JSPPortlet` sample class declares the `_fooLocalService` field to be volatile, but making a field volatile is completely optional.

Note: Service Builder generates `*LocalServiceImpl`, `*ServiceImpl`, `*PersistenceImpl`, and `[ENTITY_NAME]Impl` classes for your entities as Service Builder Spring Beans—not OSGi Declarative Services. Service Builder Spring Beans must use means other than the `@Reference` annotation to reference Liferay services and OSGi services.

Important: You should never invoke `*LocalServiceImpl` objects directly. You should only invoke them indirectly through their `*LocalService` service interface. The OSGi service registry wires the service implementation object to your class.

You can make a service object available to JSPs by associating it with a `RenderRequest` attribute. For example, the `JSPPortlet`'s `render` method associates the `FooLocalService` object with an attribute called `fooLocalService`.

```
@Override
public void render(RenderRequest request, RenderResponse response)
    throws IOException, PortletException {

    //set service bean
    request.setAttribute("fooLocalService", getFooLocalService());

    super.render(request, response);
}

public FooLocalService getFooLocalService() {
    return _fooLocalService;
}
```

If your JSP declares the `<portlet:defineObjects />` tag, it can retrieve the service object from the `RenderRequest` attribute. For example, the `JSPPortlet`'s `init.jsp` file retrieves the `FooLocalService` object from the `"fooLocalService"` attribute.

```
...
<%@
page import="com.liferay.blade.samples.servicebuilder.service.FooLocalService" %>
...

<liferay-theme:defineObjects />

<portlet:defineObjects />

<%
...

//get service bean
FooLocalService fooLocalService = (FooLocalService)request.getAttribute("fooLocalService");
%>
```

All JSPs that include the above `init.jsp` can use the `fooLocalService` variable to invoke the local service component's methods.

Step 2: Call the Component's Methods

Now that you have the service component object, you can invoke its methods as you would any Java object's methods.

The `basic-web` sample module's `view.jsp` and `edit_foo.jsp` files include the `init.jsp` shown in the previous section. Therefore, they can access the `fooLocalService` variable which references the service component object. The `view.jsp` file uses the component's `getFoosCount` method and `getFoos` method in a Liferay Search Container that lists `Foo` instances.

```
<liferay-ui:search-container
    total="<%= fooLocalService.getFoosCount() %>"
>
    <liferay-ui:search-container-results
        results="<%= fooLocalService.getFoos(searchContainer.getStart(), searchContainer.getEnd()) %>"
    />
    ...
</liferay-ui:search-container>
```

The `edit_foo.jsp` file calls `getFoo(long id)` to retrieve a `Foo` entity based on the entity instance's ID.

```
long fooId = ParamUtil.getLong(request, "fooId");
Foo foo = null;
if (fooId > 0) {
    foo = fooLocalService.getFoo(fooId);
}
```

Important: When invoking service entity updates (e.g., `fooService.update(object)`) for services that have MVCC enabled, make sure to do so in transactions. Propagate rejected transactions to the UI for the user to handle. For details, see Multiversion concurrency control (MVCC).

Using the `@Reference` annotation, you can inject your application's OSGi DS components (such as a portlet DS component) with instances of your application's Service Builder-generated local service components. Also you can provide your JSPs access to the component instances via `RenderRequest` attributes.

Related Topics

Creating Local Services
    Creating Remote Services
    Invoking Remote Services
    Service Security Layers
    Invoking Services from Service Builder Code
    OSGi Services and Dependency Injection with Declarative Services

63.8 Invoking Services from Service Builder Code

All the services created within a Service Builder application are wired using an internal Spring Application Context. This uses AOP proxies to adapt the services for transactions, indexing, and security. In a module's `module-spring.xml` Spring Application Context file, Service Builder defines each entity's `*LocalServiceImpl`, `*ServiceImpl`, and `*PersistenceImpl` classes as Spring Beans. For example, Service Builder defines Spring Beans for the `Foo` entity in the Liferay Blade Service Builder `basic-service` sample module's `src/main/resources/META-INF/spring/module-spring.xml` file:

```
<?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.xsd"
>
    <bean class="com.liferay.blade.samples.servicebuilder.service.impl.FooLocalServiceImpl" id="com.liferay.blade.samples.servicebuilder.service.Fool
    <bean class="com.liferay.blade.samples.servicebuilder.service.impl.FooServiceImpl" id="com.liferay.blade.samples.servicebuilder.service.FooServic
    <bean class="com.liferay.blade.samples.servicebuilder.service.persistence.impl.FooPersistenceImpl" id="com.liferay.blade.samples.servicebuilder.s
</beans>
```

Here's a summary of the beans the example context defines:

Interface ID	Implementation Class
`com.liferay.blade.samples.servicebuilder.service.FooLocalService`	`com.liferay.blade.samples.servicebuilder.service.impl.FooLocalServic`
`com.liferay.blade.samples.servicebuilder.service.FooService`	`com.liferay.blade.samples.servicebuilder.service.impl.FooServiceImpl`
`com.liferay.blade.samples.servicebuilder.service.persistence.FooPersistence`	`com.liferay.blade.samples.servicebuilder.service.persistence.impl.Foo`

Since these classes are Spring Beans and NOT OSGi Declarative Services components, they use annotations other than the `@Reference` Declarative Services annotation to inject Spring Beans and OSGi services. Here are the recommended Liferay annotations a Service Builder Spring Bean can use.

- Use `@BeanReference` to reference a Spring Bean that is in the Application Context.
- Use `@ServiceReference` to reference an OSGi service.

Important: When invoking service entity updates (e.g., `fooService.update(object)`) for services that have MVCC enabled, make sure to do so in transactions. Propagate rejected transactions to the UI for the user to handle. For details, see Multiversion concurrency control (MVCC).

The `@BeanReference` annotation is explained first.

Referencing a Spring Bean that is in the Application Context

A Service Builder Spring Bean class, such as a `*LocalServiceImpl` class, should use Liferay's `@BeanReference` annotation to access other Spring Beans the module's Spring Application Context defines.

For example, if your service module's `service.xml` file defines local services for entities named `Foo` and `Bar`, Service Builder generates a `module-spring.xml` file that defines local service Spring Beans for both entities. To inject the `BarLocalService` Spring Bean into the `FooLocalServiceImpl` class, for example, the `FooLocalServiceImpl` class would declare a `BarLocalService` field and apply an `@BeanReference` annotation to it.

```
@BeanReference
private BarLocalService _barLocalService;
```

The `@BeanReference` lets Liferay's AOP treat the bean reference for use in transactions, search indexing, or security, if needed. The referencing class can invoke the Spring Bean class's methods.

Besides the services Service Builder makes available for your application, Service Builder Spring Bean classes can also access any service published in 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 OSGi Declarative Services
- Services registered using the OSGi low level API

These are all OSGi services. The next section demonstrates a Service Builder Spring Bean referencing OSGi services.

Referencing OSGi Services

In many cases, your Service Builder code (Spring Beans) must use external services. Liferay's `@ServiceReference` annotation lets Liferay Spring Beans reference OSGi services.

Suppose you're building an application with a simple entity your service module defines in its `service.xml` file. The application must send an SMS every time a new entity is created, and the `SMSService` is provided by a module installed in the system.

Your `*LocalServiceImpl` (Spring Bean) could use an `@ServiceReference` annotation to reference the *external* service.

```
@ServiceReference
private SMSService _smsService;
```

This annotation retrieves a reference to the OSGi service and provides some nice benefits. None of the Spring context is created until the `SMSService` service is available. Likewise, if the `SMSService` suddenly disappears, the whole Spring Application Context is destroyed. This makes Liferay Spring apps much more robust and versatile.

Fortunately, Service Builder generates this kind of code for every entity your `service.xml` file references. For example, the Liferay Blade Service Builder sample project `basic-service` module's `service.xml` file defines a `Foo` entity that references an `AssetEntry` entity:

```
<reference entity="AssetEntry" package-path="com.liferay.portlet.asset" />
```

Service Builder generated the `FooLocalServiceBaseImpl` class (the base class is part of the `FooLocalServiceImpl` class's hierarchy), which references the `AssetEntry` entity's local service `AssetEntryLocalService` using a field annotated with `@ServiceReference`:

```
@ServiceReference(type = com.liferay.asset.kernel.service.AssetEntryLocalService.class)
protected com.liferay.asset.kernel.service.AssetEntryLocalService assetEntryLocalService;
```

Great! You now know how to add a reference to any OSGi service to a Service Builder Spring Bean. You also know how to add a reference to any other Spring Bean in the Application Context of your Service Builder Spring Bean.

Related Topics

Invoking Local Services
    Invoking Remote Services
    JSON Web Services Invoker
    Service Trackers

CHAPTER 64

APPLICATION SECURITY

Liferay's development framework provides an application security platform that's got years of experience behind it. You don't need to roll your own security for your applications. Instead, you can specify security for your applications using Liferay's framework.

Beyond security for applications, there are many ways to extend the default security model by customizing the authentication process. This group of tutorials teaches you about them:

- Resources, Roles, and Permissions
- Custom SSO Providers
- Authentication Pipelines
- Service Access Policies
- Authentication Verifiers

64.1 Defining Application Permissions

When you're writing an application, there are almost always parts of the application or its data that should be protected by permissions. Some users should access all the functions or data, but most users shouldn't.

On many platforms, developers have to create the security model themselves. On Liferay DXP, an application security model has been provided for you; you only need to make use of it.

Fortunately, no matter what your application does, access to it and to its content can be controlled with permissions. Read on to learn about Liferay's permissions system and how add permissions to your application.

The permissions system has three parts: *Resources*, *Actions*, and *Permissions*.

Action: An operation that can be performed by a user. For example, users can perform these actions on the Bookmarks application: `ADD_TO_PAGE`, `CONFIGURATION`, and `VIEW`. Users can perform these actions on Bookmarks entry entities: `ADD_ENTRY`, `DELETE`, `PERMISSIONS`, `UPDATE`, and `VIEW`.

Resource: A generic representation of any application or entity on which an action can be performed. Resources are used for permission checking. For example, resources 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: A flag that determines whether an action can be performed on a resource. In the 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.

To do permissions, therefore, you define *Users* (Roles) who have *Permission* to perform *Actions* on *Resources*. User definition is done by administrators once your application is deployed; developers define resources, actions, and default permissions.

You can implement permissions in your applications in four steps that spell the acronym *DRAC*:

1. Define all resources and their permissions.

2. Register all defined resources in the permissions system.

3. Associate the necessary permissions with resources.

4. Check permission before returning resources.

The next four tutorials show these steps in detail.

64.2 Defining Resources and Permissions

Your first step in implementing permissions is to define the resources and the permissions that protect them. There are two different kinds of resources: *portlet resources* and *model resources*.

Portlet resources represent portlets. The names of portlet resources are the portlet IDs from the portlets' `@Component` properties or if you're using a WAR file, `portlet.xml` files. Model resources refer to model objects, usually persisted as entities to a database. The names of model resources are their fully qualified class names. In the XML displayed below, permission implementations are first defined for the *portlet* resource and then for the *model* resources.

Model resources represent models, such as blog entries. Resources are named using the fully qualified class names of the entities they represent.

Note: For each resource, there are four scopes to which the permissions can be applied: company, group, group-template, or individual. Because these are called *portlet resources* here and in the code, this can be confusing. The other scopes are mostly used internally for various Liferay constructs (such as Sites or Categories).

You define resources and their permissions using an XML file. By convention, this file is called `default.xml` and exists in a module's `src/main/resources/resource-actions` folder.

Because of the two different types of resources, you'll have two of these files: one in your portlet module to define the portlet resources and one in your service module to define the model resources.

Defining Portlet Resource Permissions

Define the portlet resources first; here's an example using Liferay's Blogs application.

1. Start with the DTD declaration:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action Mapping 7.1.0//EN" "http://www.liferay.com/dtd/liferay-resource-action-mapping_7_0_0.dtd">
```

2. The root tag contains all the resources to be declared:

```
<resource-action-mapping>

</resource-action-mapping>
```

3. Inside these tags, define your resources. The Blogs application defines two portlet resources:

```
<portlet-resource>
    <portlet-name>com_liferay_blogs_web_portlet_BlogsAdminPortlet</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_blogs_web_portlet_BlogsPortlet</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>
        <site-member-defaults>
            <action-key>VIEW</action-key>
        </site-member-defaults>
        <guest-defaults>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
            <action-key>ADD_PORTLET_DISPLAY_TEMPLATE</action-key>
            <action-key>CONFIGURATION</action-key>
        </guest-unsupported>
    </permissions>
</portlet-resource>
```

The Blogs application comprises two portlets: the Blogs portlet itself and the Blogs Admin portlet that appears in the Site menu for administrators. Define your portlets by their names, and then list the permissions for the portlet. The Blogs portlet, for example, supports four permissions: ADD_PORTLET_DISPLAY_TEMPLATE, ADD_TO_PAGE, CONFIGURATION, and VIEW. The Blogs Admin portlet has an additional permission: ACCESS_IN_CONTROL_PANEL, which defines who can see the entry in the Site menu.

Once you've defined permissions at the portlet level, you can set default permissions for different types of users. The DTD allows for site member and guest defaults. Since guests are users that aren't logged in, there's also a `guest-unsupported` tag for defining permissions guests can *never* have (in other words, the user must be logged in and identifiable).

That's all there is to it! Your next task is to define permissions for your model resources.

Defining Model Resource Permissions

Defining permissions for models is a similar process. Create a `default.xml` file in your service module's `src/main/resources/resource-actions` folder. In this file, you must define top-level function permissions and individual entity permissions using the same `<model-resource>` tag.

This can be confusing, so some explanation is in order. Model permissions for what Liferay calls the *root model* are defined separately from permissions on stored entities, which Liferay calls the *model*. This makes sense when you think about the functions users can perform:

- Creating something new
- Editing something that exists

Creating something new (like adding a new Blog entry) is different from accessing something that exists. A Blog owner should be able to create or edit a Blog entry, but a User or guest should have read permission for existing entries and no permission to create them.

Permission to create something new that doesn't yet exist is a *root model* permission, whether that functionality is exposed in a portlet or not. Permission on an existing resource is a *model* permission.

Now you're ready to define both your root model and model permissions.

1. First, create the skeleton for your file:

```
<?xml version="1.0"?>
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action Mapping 7.1.0//EN" "http://www.liferay.com/dtd/liferay-
resource-action-mapping_7_0_0.dtd">

<resource-action-mapping>

</resource-action-mapping>
```

2. Inside the `<resource-action-mapping>` tags, use a `<model-resource>` tag to define permissions for the root model:

```
<model-resource>
    <model-name>com.liferay.blogs</model-name>
    <portlet-ref>
        <portlet-name>com_liferay_blogs_web_portlet_BlogsAdminPortlet</portlet-name>
        <portlet-name>com_liferay_blogs_web_portlet_BlogsPortlet</portlet-name>
    </portlet-ref>
    <root>true</root>
    <weight>1</weight>
```

```
    <permissions>
        <supports>
            <action-key>ADD_ENTRY</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>SUBSCRIBE</action-key>
        </supports>
        <site-member-defaults>
            <action-key>SUBSCRIBE</action-key>
        </site-member-defaults>
        <guest-defaults />
        <guest-unsupported>
            <action-key>ADD_ENTRY</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>SUBSCRIBE</action-key>
        </guest-unsupported>
    </permissions>
</model-resource>
```

The model name (`com.liferay.blogs`) is just a package name. The `<root>true</root>` tag defines this as a root model. The `<portlet-ref>` element is the name of the portlet that uses the model. The `<weight>` tag defines the order of these permissions in the GUI. The permissions defined are ADD_ENTRY (add a Blog entry), PERMISSIONS (set permissions on Blog entries), and SUBSCRIBE (receive notifications when Blog entries are created). These are all root model permissions, because no primary key in the database can be assigned to any of these functions. The default permissions (for both model and portlet resources) are added when the portlet defined by the `<portlet-ref>` tag initializes.

3. Finally, define your model permissions:

```
<model-resource>
    <model-name>com.liferay.blogs.model.BlogsEntry</model-name>
    <portlet-ref>
        <portlet-name>com_liferay_blogs_web_portlet_BlogsAdminPortlet</portlet-name>
        <portlet-name>com_liferay_blogs_web_portlet_BlogsPortlet</portlet-name>
    </portlet-ref>
    <weight>2</weight>
    <permissions>
        <supports>
            <action-key>ADD_DISCUSSION</action-key>
            <action-key>DELETE</action-key>
            <action-key>DELETE_DISCUSSION</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>UPDATE</action-key>
            <action-key>UPDATE_DISCUSSION</action-key>
            <action-key>VIEW</action-key>
        </supports>
        <site-member-defaults>
            <action-key>ADD_DISCUSSION</action-key>
            <action-key>VIEW</action-key>
        </site-member-defaults>
        <guest-defaults>
            <action-key>ADD_DISCUSSION</action-key>
            <action-key>VIEW</action-key>
        </guest-defaults>
        <guest-unsupported>
            <action-key>DELETE</action-key>
            <action-key>DELETE_DISCUSSION</action-key>
            <action-key>PERMISSIONS</action-key>
            <action-key>UPDATE</action-key>
            <action-key>UPDATE_DISCUSSION</action-key>
        </guest-unsupported>
    </permissions>
</model-resource>
```

Note the lack of a `<root>` tag, the fully qualified class name for the model, and the permissions that operate on an entity with a primary key.

Enabling Your Permissions Configuration

Your last step is to enable your permission definitions. Each module that contains a `default.xml` permissions definition file must also have a `portlet.properties` file with a property that defines where to find the permissions definition file. For your service and your web modules, create a `portlet.properties` file in `src/main/resources` and make sure it has this property:

```
resource.actions.configs=resource-actions/default.xml
```

Once you've defined portlet permissions, root model permissions, and model permissions, you've completed step 1 (the *D* in DRAC). Congratulations! You're now ready to *register* the resources you've now defined in the permissions system.

64.3 Registering Permissions

Defining permissions was your first step; now you're ready to register the permissions you've defined. You must register your entities both in the database and in the permissions service running in the OSGi container.

Registering Permissions Resources in the Database

All this takes is a call to Liferay's resource service in your service layer. If you're using Service Builder, this is very easy to do.

1. Open your `-LocalServiceImpl` class.

2. In your method that adds an entity, add a call to add a resource with the entity. For example, Liferay's Blogs application adds resources this way:

   ```
   resourceLocalService.addResources(
       entry.getCompanyId(), entry.getGroupId(), entry.getUserId(),
       BlogsEntry.class.getName(), entry.getEntryId(), false,
       addGroupPermissions, addGuestPermissions);
   ```

   This method requires passing in the company ID, the group ID, the user ID, the entity's class name, the entity's primary key, and some boolean settings. In order, these settings define

   - Whether the permission is a portlet resource
   - Whether the default group permissions defined in `default.xml` should be added
   - Whether the default guest permissions defined in `default.xml` should be added

Note that the resource local service is injected automatically into your Service Builder-generated service.

If you're not using Service Builder, but you are using OSGi modules for your application, you should be able to inject the resource service with an `@Reference` annotation. If you're building a

WAR-style plugin, you need a service tracker to gain access to the service. Note that your model classes must also implement Liferay's `ClassedModel` interface.

Similarly, when you delete an entity, you should also delete its associated resource. Here's how the Blogs application does it in its `deleteEntry()` method:

```
resourceLocalService.deleteResource(
    entry.getCompanyId(), BlogsEntry.class.getName(),
    ResourceConstants.SCOPE_INDIVIDUAL, entry.getEntryId());
```

As with adding resources, the method needs to know the entity's company ID, class, and primary key. Most of the time, its scope is an individual entity of your own choosing. Other scopes available as constants are for company, group, or group template (site template). These are used internally for those objects, so you'd only use them if you were customizing functionality for creating and deleting them.

Now you're ready to register your entities with the permissions service.

Registering Entities to the Permissions Service

The permissions service that's running must know about your entities and how to check permissions for them. This requires creating a permissions registrar class.

1. In your service bundle, create a package that by convention ends in `internal.security.permission.resource`. For example, the Blogs application's package is named `com.liferay.blogs.internal.security.permission.resou`
2. Create a class in this package called `[Entity Name]ModelResourcePermissionRegistrar`. For example, the Blogs application's class is named `BlogsEntryModelResourcePermissionRegistrar`.
3. This class is a component class that requires overriding the `activate` method to register the permissions logic you want for your entities. For example, this is how the Blogs application registers its permissions:

```
@Component(immediate = true)
public class BlogsEntryModelResourcePermissionRegistrar {

    @Activate
    public void activate(BundleContext bundleContext) {
        Dictionary<String, Object> properties = new HashMapDictionary<>();

        properties.put("model.class.name", BlogsEntry.class.getName());

        _serviceRegistration = bundleContext.registerService(
            ModelResourcePermission.class,
            ModelResourcePermissionFactory.create(
                BlogsEntry.class, BlogsEntry::getEntryId,
                _blogsEntryLocalService::getEntry, _portletResourcePermission,
                (modelResourcePermission, consumer) -> {
                    consumer.accept(
                        new StagedModelPermissionLogic<>(
                            _stagingPermission, BlogsPortletKeys.BLOGS,
                            BlogsEntry::getEntryId));
                    consumer.accept(
                        new WorkflowedModelPermissionLogic<>(
                            _workflowPermission, modelResourcePermission,
                            BlogsEntry::getEntryId));
                }),
            properties);
    }
```

```
    @Deactivate
    public void deactivate() {
        _serviceRegistration.unregister();
    }

    @Reference
    private BlogsEntryLocalService _blogsEntryLocalService;

    @Reference(target = "(resource.name=" + BlogsConstants.RESOURCE_NAME + ")")
    private PortletResourcePermission _portletResourcePermission;

    private ServiceRegistration<ModelResourcePermission> _serviceRegistration;

    @Reference
    private StagingPermission _stagingPermission;

    @Reference
    private WorkflowPermission _workflowPermission;

}
```

We call these types of classes Registrars because the classes' job is to configure, register and unregister the `ModelResourcePermission`.

1. The `model.class.name` is set in the properties so that other modules' service trackers can find this model resource permission by its type when it's needed. Liferay has several service trackers checking for model resource permissions. The `service.ranking` property can also be set to a value greater than zero to override other module's model resource permissions.

2. This registrar uses two portal-kernel permission logic classes for Staging and Workflow. Custom logic classes can be reused or composed inline since `ModelResourcePermissionLogic` is a `@FunctionalInterface`. Permission logic classes are executed in order of when they are accepted in the `Consumer`.

3. `ModelResourcePermissionLogic` classes return `true` when users have permission for the action, `false` when they are denied permission for the action, and `null` when wanting to delegate responsibility to the next permission logic. If all permission logics return null then the `PermissionChecker.hasPermission` method is called to determine if the action is allowed for the user.

This class uses an `@Reference` with the target filter to inject the appropriate `PortletResourcePermission`. `BlogsConstants.RESOURCE_NAME` evaluates to `com.liferay.blogs`, which is defined in the `default.xml` you created earlier. If you were to reference this `ModelResourcePermission`, you'd use a target filter matching the `model.class.name` property set in the `activate` method.

Note that you specify your entity's class, primary key, and the entity itself for the factory so it can create permission objects specific to your entity.

Great! You've now completed step 2 in *DRAC* by registering your permissions. Now you're ready to provide users the interface to associate permissions with resources.

64.4 Associating Permissions with Resources

Now that you've defined and registered permissions, you must expose the permissions interface so users can set permissions.

To allow permissions to be configured for model resources, you must add the permissions interface to the UI. Add these two Liferay UI tags to your JSP:

1. `<liferay-security:permissionsURL>`: Returns a URL to the permission settings configuration page.
2. `<liferay-ui:icon>`: Shows an icon to the user. These are defined in the theme and one of them (see below) is used for permissions.

The Blogs application uses these tags like this:

```
<liferay-security:permissionsURL
            modelResource="<%= BlogsEntry.class.getName() %>"
            modelResourceDescription="<%= BlogsEntryUtil.getDisplayTitle(resourceBundle, entry) %>"
            resourceGroupId="<%= String.valueOf(entry.getGroupId()) %>"
            resourcePrimKey="<%= String.valueOf(entry.getEntryId()) %>"
            var="permissionsEntryURL"
            windowState="<%= LiferayWindowState.POP_UP.toString() %>"
        />

        <liferay-ui:icon
            label="<%= true %>"
            message="permissions"
            method="get"
            url="<%= permissionsEntryURL %>"
            useDialog="<%= true %>"
        />
```

For the `<liferay-security:permissionsURL />` tag, specify these attributes:

`modelResource`: The fully qualified class name of the entity class. This class name gets translated into a more readable name as specified in `Language.properties`.

`Language.properties`: The entity class in the example above is the Blogs entry class for which the fully qualified class name is `com.liferay.blogs.model.BlogsEntry`.

`modelResourceDescription`: You can enter anything that best describes this model instance. In the example above, the Blog title is used for the model resource description.

`resourcePrimKey`: Your entity's primary key.

`var`: The name of the variable to which the resulting URL string is assigned. The variable is then passed to the `<liferay-ui:icon>` tag so the permission icon has the proper URL link.

There's an optional attribute called `redirect` that's available if you want to override the default behavior of the upper right arrow link. That's it; now your users can configure the permission settings for model resources!

You've completed step 3 in *DRAC*. Your next step is to check for permissions in the appropriate areas of your application.

64.5 Checking Permissions

Now that you've defined your permissions, registered resources in the database and with the OSGi container, and enabled users to associate permissions with resources, you're ready to add permission checks in the appropriate places in your application. This takes three steps:

1. Add permission checks to your service calls.
2. Create permission helper classes in your web module.

3. Add permission checks to your web application.

These things are covered next.

Add Permission Checks to Your Service Calls

A best practice is to create methods in your `-ServiceImpl` classes that call the same methods in your `-LocalServiceImpl` classes, but wrap those calls in permission checks. If you expose your services as web services, then any client calling those services must have permission to call the service. In this way, you separate your business logic (contained in the `-LocalServiceImpl` class) from your permissions logic (contained in the `-ServiceImpl` class).

1. Open your entity's `-ServiceImpl` class.

2. Use the `ModelResourcePermissionFactory` and the `PortletResourcePermissionFactory` to reference permission checkers that can check permissions as you've defined them in `default.xml`. Here's how the Blogs portlet does this:

```
private static volatile ModelResourcePermission<BlogsEntry>
    _blogsEntryFolderModelResourcePermission =
        ModelResourcePermissionFactory.getInstance(
            BlogsEntryServiceImpl.class,
            "_blogsEntryFolderModelResourcePermission", BlogsEntry.class);
private static volatile PortletResourcePermission
    _portletResourcePermission =
        PortletResourcePermissionFactory.getInstance(
            BlogsEntryServiceImpl.class, "_portletResourcePermission",
            BlogsConstants.RESOURCE_NAME);
```

   You declare the class, the variable, and for the portlet resource, the resource name from `default.xml`. In the Blogs application, `BlogsConstants.RESOURCE_NAME` is a `String` with the value `com.liferay.blogs`.

   You must use `ModelResourcePermissionFactory.getInstance()` in the service because Service Builder is wired with Spring, so `@Reference` can't be used. Make sure to use the correct service class and the name of the field that's being set (in this case `"_blogsEntryFolderModelResourcePermission"`), because it's set with reflection when the service is registered. If you get the field wrong, it'll be set wrong. The field must be `static` and `volatile`, and should never be used outside of `-ServiceImpl` classes.

3. Check permissions in the appropriate places. For example, adding a blog entry requires the `ADD_ENTRY` permission, so the Blogs application does this:

```
@Override
public BlogsEntry addEntry(
        String title, String subtitle, String description, String content,
        int displayDateMonth, int displayDateDay, int displayDateYear,
        int displayDateHour, int displayDateMinute, boolean allowPingbacks,
        boolean allowTrackbacks, String[] trackbacks,
        String coverImageCaption, ImageSelector coverImageImageSelector,
        ImageSelector smallImageImageSelector,
        ServiceContext serviceContext)
    throws PortalException {

    _portletResourcePermission.check(
        getPermissionChecker(), serviceContext.getScopeGroupId(),
```

```
        ActionKeys.ADD_ENTRY);

    return blogsEntryLocalService.addEntry(
        getUserId(), title, subtitle, description, content,
        displayDateMonth, displayDateDay, displayDateYear, displayDateHour,
        displayDateMinute, allowPingbacks, allowTrackbacks, trackbacks,
        coverImageCaption, coverImageImageSelector, smallImageImageSelector,
        serviceContext);
}
```

The check throws an exception if it fails, preventing the local service call that adds the entry. A convention Liferay uses is to place the action keys from `default.xml` as constants in an `ActionKeys` class. If `ActionKeys` doesn't have an action key appropriate for your application, extend Liferay's class and add your own keys.

Add permission checks where necessary to protect your application's functions at the service level. Next, you'll learn how to create permission helper classes for your web module.

Create Permission Helper Classes in Your Web Module

A helper class can make it easier to check permissions in your portlet application. You can create helper classes for both portlet permissions and model permissions. Here's how to create a portlet permission helper:

1. Create a package with the suffix `web.internal.security.permission.resource`. For example, the Blogs application has the package `com.liferay.blogs.web.internal.security.permission.resource`.
2. Create a component class with at least one static method for checking permissions. For example, here's the `BlogsPermission` class:

```
@Component(immediate = true)
public class BlogsPermission {

    public static boolean contains(
        PermissionChecker permissionChecker, long groupId, String actionId) {

        return _portletResourcePermission.contains(
            permissionChecker, groupId, actionId);
    }

    @Reference(
        target = "(resource.name=" + BlogsConstants.RESOURCE_NAME + ")",
        unbind = "-"
    )
    protected void setPortletResourcePermission(
        PortletResourcePermission portletResourcePermission) {

        _portletResourcePermission = portletResourcePermission;
    }

    private static PortletResourcePermission _portletResourcePermission;

}
```

Note the `@Reference` annotation that tells the OSGi container to supply an object via the permission registrar you created previously. The `_portletResourcePermission` field is static, while the setter method is an instance method: this is how Liferay avoids having service references in JSPs.

The procedure for creating a model permission helper is similar:

1. In the same package, create a component class with at least one static method for checking permissions. For example, here's the `BlogsEntryPermission` class:

```
@Component(immediate = true)
public class BlogsEntryPermission {

    public static boolean contains(
            PermissionChecker permissionChecker, BlogsEntry entry,
            String actionId)
        throws PortalException {

        return _blogsEntryFolderModelResourcePermission.contains(
            permissionChecker, entry, actionId);
    }

    public static boolean contains(
            PermissionChecker permissionChecker, long entryId, String actionId)
        throws PortalException {

        return _blogsEntryFolderModelResourcePermission.contains(
            permissionChecker, entryId, actionId);
    }

    @Reference(
        target = "(model.class.name=com.liferay.blogs.model.BlogsEntry)",
        unbind = "-"
    )
    protected void setEntryModelPermission(
        ModelResourcePermission<BlogsEntry> modelResourcePermission) {

        _blogsEntryFolderModelResourcePermission = modelResourcePermission;
    }

    private static ModelResourcePermission<BlogsEntry>
        _blogsEntryFolderModelResourcePermission;

}
```

   As you can see, this class is almost the same as the portlet permission class. The real difference is in the `@Reference` annotation that specifies the fully qualified class name of the model, rather than the resource name from `default.xml`.

2. Save both files.

Now you're ready to use these helper classes to check permissions in your web module.

Add Permission Checks to Your Web Application

You can use the permission helper classes to check for permissions before displaying UI elements. If the element never appears, a user can't access it (though you should also protect your services as described above). Here's how to do that:

1. When you have a function you want to protect, wrap it in an `if` statement that uses the permission helper class. For example, the Blogs application has many functions protected by permissions, including `ADD_ENTRY` and `SUBSCRIBE`. Clearly, only blog owners should be able to add blog entries. The button for this, therefore, should only appear if a user has permission to add entries:

```
<c:if test="<%= BlogsPermission.contains(permissionChecker, scopeGroupId, ActionKeys.ADD_ENTRY) %>">
    <div class="button-holder">
        <portlet:renderURL var="editEntryURL" windowState="<%= WindowState.MAXIMIZED.toString() %>">
            <portlet:param name="mvcRenderCommandName" value="/blogs/edit_entry" />
            <portlet:param name="redirect" value="<%= currentURL %>" />
        </portlet:renderURL>

        <aui:button href="<%= editEntryURL %>" icon="icon-plus" value="add-blog-entry" />
    </div>
</c:if>
```

2. Do this for any function. For example, the Permissions function you added in step 3 should definitely be protected by permissions:

```
<c:if test="<%= BlogsEntryPermission.contains(permissionChecker, entry, ActionKeys.PERMISSIONS) %>">
    <liferay-security:permissionsURL
        modelResource="<%= BlogsEntry.class.getName() %>"
        modelResourceDescription="<%= BlogsEntryUtil.getDisplayTitle(resourceBundle, entry) %>"
        resourceGroupId="<%= String.valueOf(entry.getGroupId()) %>"
        resourcePrimKey="<%= String.valueOf(entry.getEntryId()) %>"
        var="permissionsEntryURL"
        windowState="<%= LiferayWindowState.POP_UP.toString() %>"
    />

    <liferay-ui:icon
        label="<%= true %>"
        message="permissions"
        method="get"
        url="<%= permissionsEntryURL %>"
        useDialog="<%= true %>"
    />
</c:if>
```

   This prevents anyone without the permission to set permissions from seeing the permissions button. Say that three times fast!

That's all there is to it! You've now learned all the steps in *DRAC*:

1. Define permissions
2. Register permissions
3. Associate permissions with resources
4. Check permissions

Follow these steps, and your applications can take advantage of Liferay's integrated and well-tested permissions system.

64.6 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://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>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 65

AUTHENTICATION PIPELINES

The authentication process is a pipeline through which users can be validated by one or several systems. As a developer, you can authenticate users to anything you wish, rather than be limited by what Liferay DXP 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 the Login Portlet to support whatever user interface any of these systems need. This gives you full flexibility over the entire authentication process.

This structure lets you 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.

These tutorials guide 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.

65.1 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.

Creating an Auto Login Component

Create a Declarative Services component. The component should implement the `com.liferay.portal.kernel.securit` 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.

For example, say that there's a request attribute that contains the encrypted value of a user key. 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

65.2 Password-Based Authentication Pipelines

By default, once a user submits credentials, 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 the database, you can write an `Authenticator` and insert it as a step in the 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 user name 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 database. In this step, you can skip credential validation against the database. Implemented by `Authenticator`.
2. Default (optional) authentication to the 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 can't perform its task: 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 *@example.com* or *@example.com*. You can implement this using one module that does everything. If you think other modules might use the functionality that validates the email addresses, you should 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:

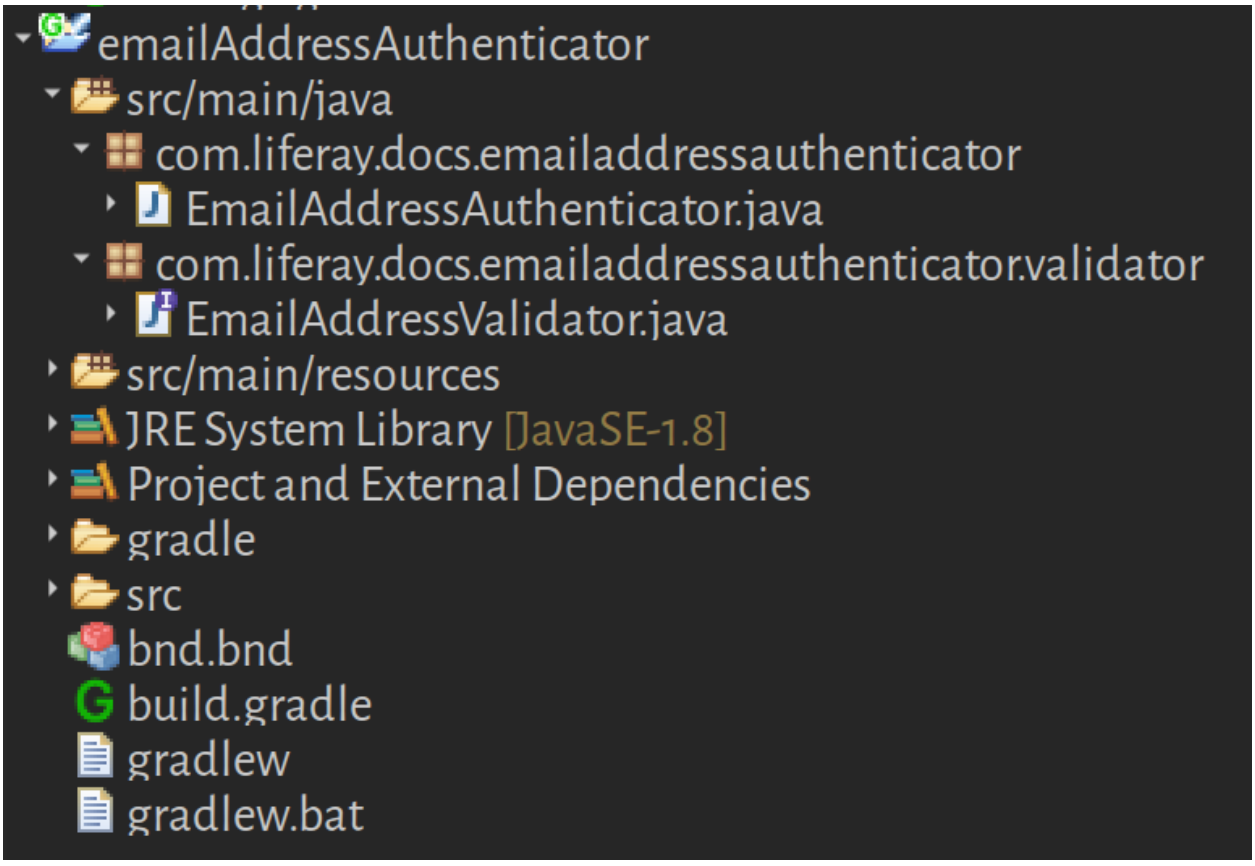


Figure 65.1: The Authenticator module contains the validator's interface and the authenticator.

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 `UserLocalService` to look up the user's email address before checking it. The OSGi container injects this service 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
```

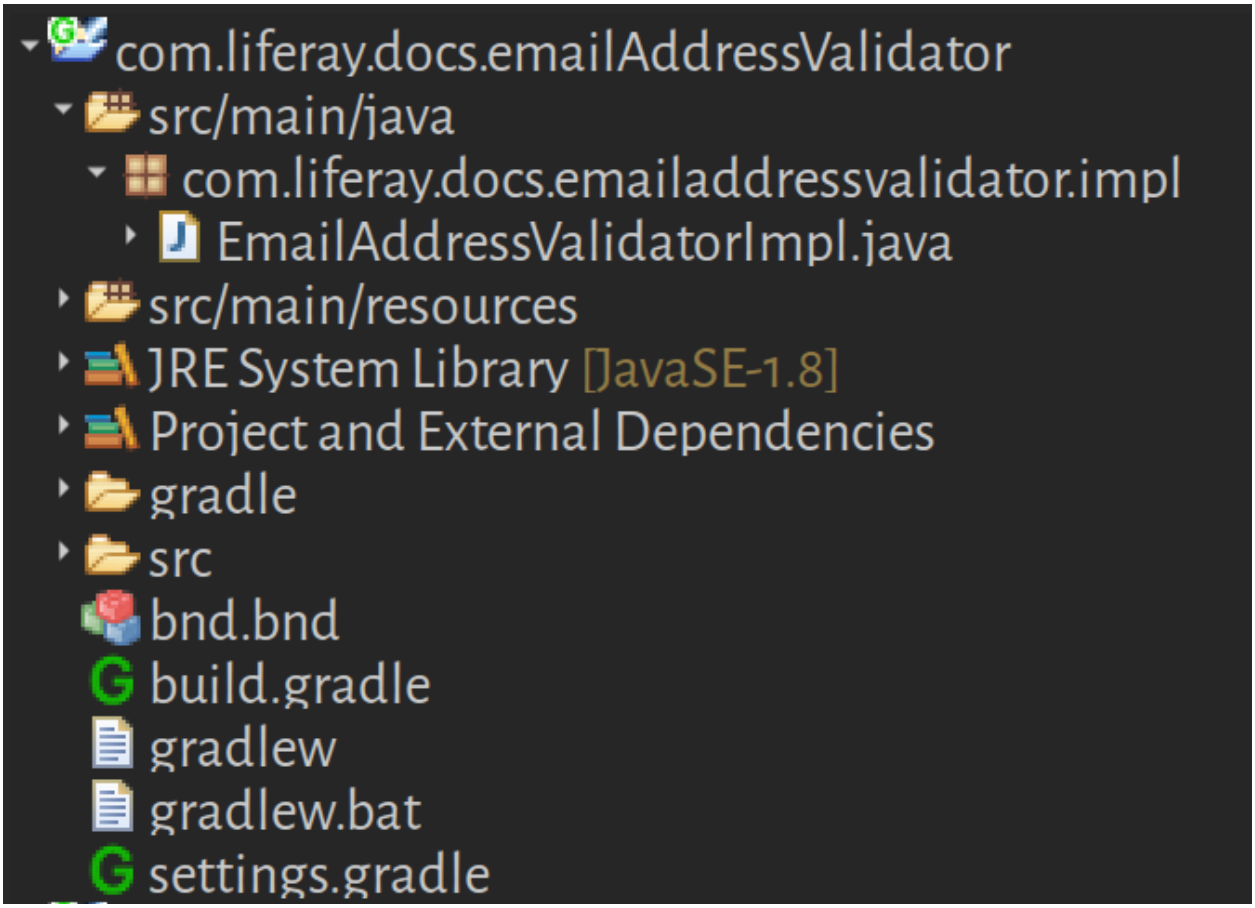


Figure 65.2: The validator project implements the Validator Interface and depends on the authenticator module.

```
    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[] {"@example.com", "@example2.com"}));
}
```

This code checks to make sure that the email address is from the *@example.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

65.3 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 the 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.

Note: When using the example project, set the session timeout portal property like this:

```
session.timeout.auto.extend.offset=45
```

This is needed because the default (as of LPS-68543) setting is `0`, causing the browser to execute an `extend_session` call. This may force users attempting to log in to make the attempt twice.

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 (the 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

65.4 Service Access Policies

Service access policies provide web service security beyond user authentication to remote services. Together with permissions, 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 web service, 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 further limit the remote client's access to the 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. Since the client doesn't perform other 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

A remote client's request to a web service contains the user's credentials or an authorization token. An authentication module 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.

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 must use the service access policies API:

- It uses custom remote API authentication (tokens) and require certain services to be available for clients using the tokens.
- It requires its services be made available to guest users, with no authentication necessary.
- It contains 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` 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.

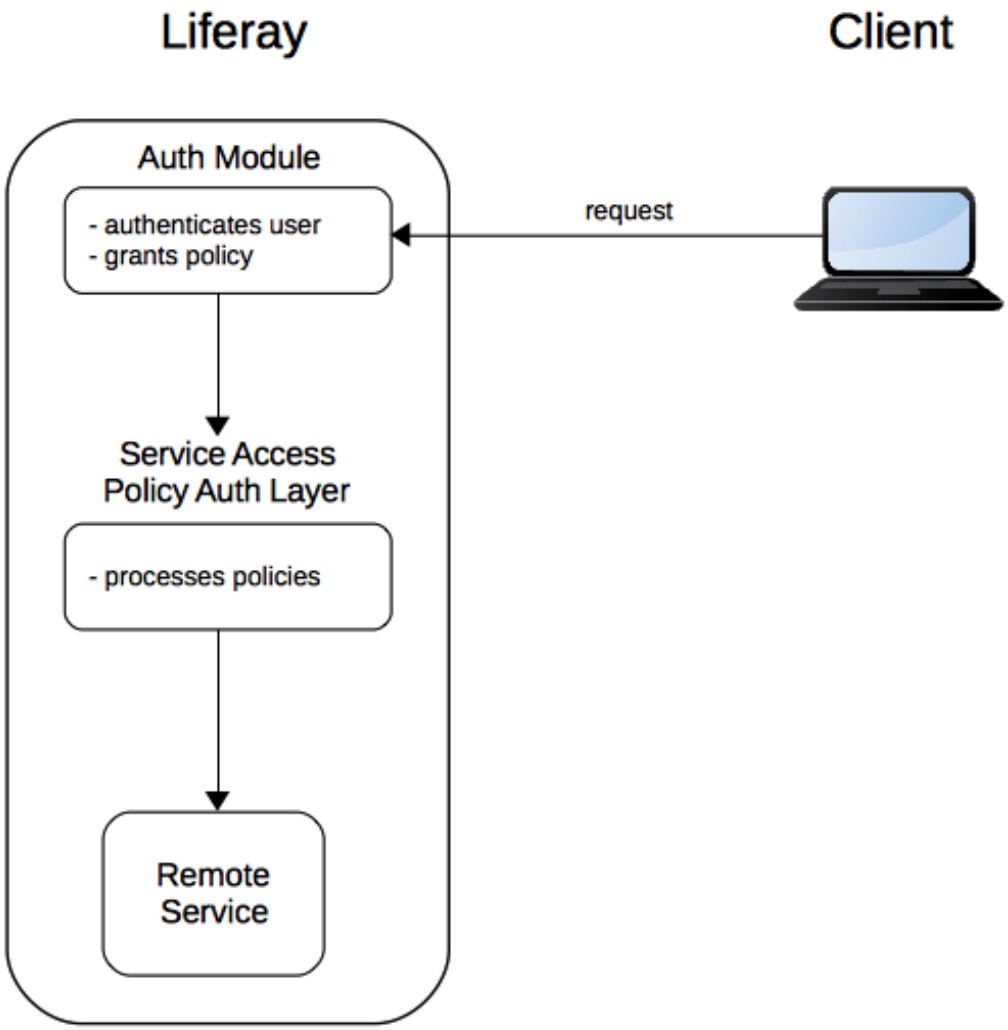


Figure 65.3: The authorization module maps the credentials or token to the proper Service Access Policy.

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 by default, and you can use them to manage Service Access Policies programmatically. You can find their source code here in GitHub. 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.servic` 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.

Service Access Policy Example

Liferay Sync's `sync-security` module is a service access policy module. It uses `com.liferay.portal.security.service.` 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());
            }
    }

    ...

}
```

This class creates the policies when the module starts. Note that this module is included and enabled 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.

CHAPTER 66

WEB SERVICES

It's important for apps on different machines to communicate. To enable this, an app can expose APIs so remote components (other apps or devices) can access the app's features. For example, one service could have a client app presenting information to users, a server app processing data in B2B setting, and an IoT device requesting data to do its work. Exposing web APIs lets external applications or devices communicate with yours.

Because Liferay DXP contains so many apps and features, it's prudent for Liferay to let developers access those apps and features from external apps and devices by exposing their APIs. Additionally, Liferay's development platform makes it easy to extend them and create new ones.

There are two different approaches for clients to connect to Liferay DXP's web APIs:

Headless REST APIs: You can consume RESTful web services independent of Liferay DXP's front end (hence *headless*). These APIs conform to the OpenAPI specification. This is the modern, preferred way to work with web services in Liferay DXP.

Plain Web/REST Services: This is the old way to build and consume web services in Liferay DXP, but is still supported. For example, you can use JAX-RS, JAX-WS, or Service Builder to implement plain REST or SOAP web services.

The tutorials that follow show you how to consume and create web services in Liferay DXP, beginning with headless REST APIs.

66.1 Headless REST APIs

Liferay DXP's headless REST APIs follow the OpenAPI specification and let your apps consume RESTful web services. What's more, you can consume these APIs without being tied to Liferay DXP's UI (hence the term *headless*). This gives you a great deal of freedom when designing and developing your apps.

The articles in this section show you how to navigate and consume Liferay DXP's headless REST APIs. But first, you'll learn the design approach for these APIs.

API Vocabulary

When defining an API, the developer must decide how to expose the representation of its resources. This determines its ease of use and how it can evolve. Traditionally, there are two approaches:

Contract Last: The code is written first and features are exposed as web or REST services. This approach is typically easier for developers, as they must only implement and expose the business logic. Service Builder is an example of this.

Contract First: The structure for client-server messages is written before the code that implements the services. Such messages are defined independent of the code. This avoids tight coupling and is less likely to break clients as APIs evolve.

Liferay DXP's headless web APIs use a mixture of both approaches. An OpenAPI profile uses a contract first approach by defining the paths and schemas before writing any code. It then generates an API automatically based on that profile, using the contract-last characteristic of code generation (like Service Builder). This allows fast development for developers.

This mixed approach delivers the best of both worlds, allowing a step of conscious API design and then simplifying the developer experience by exposing only the business logic to implement.

When writing the OpenAPI profile, the main focus should be on defining how client-server messages represent the APIs' resources. In other words, the APIs' schemas are defined first and the attributes, resources, and operations are named to clearly define what they represent and how they should be used.

66.2 Get Started: Discover the API

To begin consuming web services, you must first know where they are (e.g., a service catalog), what operations you can invoke, and how to invoke them. Because Liferay DXP's headless REST APIs leverage OpenAPI (originally known as Swagger), you don't need a service catalog. You only need to know the OpenAPI profile from which to discover the rest of the API.

Liferay DXP's headless APIs are available in SwaggerHub at `https://app.swaggerhub.com/organizations/liferayin`. Each API has its own URL in SwaggerHub. For example, you can access the delivery API definition at `https://app.swaggerhub.com/apis/liferayinc/headless-delivery/v1.0`.

Each OpenAPI profile is also deployed dynamically in your portal instance under this schema:

```
http://[host]:[port]/o/[insert-headless-api]/[version]/openapi.yaml
```

For example, if you're running Liferay DXP locally on port `8080`, the home URL for discovering the headless delivery API is:

```
http://localhost:8080/o/headless-delivery/v1.0/openapi.yaml
```

You must be logged in to access this URL, or use basic authentication and a browser or other tool like Postman, Advanced REST Client, or even the `curl` command in your system console.

For simplicity, the examples in this documentation use the `curl` command and send requests to a Liferay DXP instance running locally on port `8080`.

Run this `curl` command to access the home URL:

```
curl http://localhost:8080/o/headless-delivery/v1.0/openapi.yaml -u test@example.com:test
```

You should get a response like this:

```
openapi: 3.0.1
info:
  title: Headless Delivery
  version: v1.0
paths:
```

```
  /v1.0/blog-posting-images/{blogPostingImageId}:
    get:
      tags:
      - BlogPostingImage
      operationId: getBlogPostingImage
      parameters:
      - name: blogPostingImageId
        in: path
        required: true
        schema:
          type: integer
          format: int64
      responses:
        default:
          description: default response
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/BlogPostingImage'
(...)
```

This response follows the OpenAPI version 3.0 syntax to specify the endpoints (URLs) of the API and schemas returned. You can also open the OpenAPI profile in an OpenAPI editor like the Swagger Editor. You can use this editor to inspect the documentation and parameters and make requests to the API.

There are also many other tools that support OpenAPI, such as client generators, validators, parsers, and more. See OpenAPI.Tools for a comprehensive list. Leveraging OpenAPI provides standards support, extensive documentation, and industry-wide conventions.

Related Topics

Get Started: Invoke a Service

66.3 Get Started: Invoke a Service

Once you know which API you want to call via the OpenAPI profile, you can send a request to that resource's URL. For example, suppose you want to retrieve all the blog entries from a Site. If you consult the OpenAPI profile for Liferay DXP's delivery API, you can find this endpoint:

```
"/sites/{siteId}/blog-postings":
       get:
           operationId: getSiteBlogPostingsPage
           parameters:
               - in: path
                 name: siteId
                 required: true
                 schema:
                     format: int64
                     type: integer
               - in: query
                 name: filter
                 schema:
                     type: string
               - in: query
                 name: page
                 schema:
                     type: integer
               - in: query
                 name: pageSize
```

```
            schema:
                type: integer
          - in: query
            name: search
            schema:
                type: string
          - in: query
            name: sort
            schema:
                type: string
      responses:
          200:
              content:
                  application/json:
                      schema:
                          items:
                              $ref: "#/components/schemas/BlogPosting"
                          type: array
              description: ""
      tags: ["BlogPosting"]
```

The only required parameter is `siteId`, the ID of the blog postings' Site. Internally, the `siteId` is a `groupId` that you can retrieve from the database, a URL, or Liferay DXP's UI via the Site Administration menu. The following GET request gets the site's blog postings by providing the site ID (`20124`) in the URL:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/sites/20124/blog-postings/" -u 'test@example.com:test'
```

If you send such a request to a site that contains some blog entries, the response should look like this:

```
{
  "items": [
    {
      "alternativeHeadline": "The power of OpenAPI & Liferay",
      "articleBody": "<p>We are happy to announce...</p>",
      "creator": {
        "familyName": "Test",
        "givenName": "Test",
        "id": 20130,
        "name": "Test Test",
        "profileURL": "/web/test"
      },
      "dateCreated": "2019-04-22T07:04:47Z",
      "dateModified": "2019-04-22T07:04:51Z",
      "datePublished": "2019-04-22T07:02:00Z",
      "encodingFormat": "text/html",
      "friendlyUrlPath": "new-headless-apis",
      "headline": "New Headless APIs",
      "id": 59301,
      "numberOfComments": 0,
      "siteId": 20124
    }
  ],
  "lastPage": 1,
  "page": 1,
  "pageSize": 20,
  "totalCount": 1
}
```

This response is a JSON object with information about the collection of blogs. The response's attributes contain information about the resource (blogs, in this case). Also note that the results

are paginated. The *page* attributes refer to pages of results. Here's a description of some common attributes:

`id`: Each item has an ID. You can use the ID to retrieve more information about that item. For example, there are two `id` attributes in the above response: one for the blog posting (`59301`) and one for the blog post's creator (`20130`).

`lastPage`: The page number of the final page of results. The above response only contains a single page, so its last page is `1`.

`page`: The page number of the current page. The page in the above response is `1`.

`pageSize`: The possible number of this resource's items to be included in a single page. In the above response this is `20`.

`totalCount`: The total number of this resource's existing items (independent of pagination). The above response lists the total number of blog postings (`1`) in a Site.

To get information on a specific blog posting, send a GET request to the `blogPostingId` resource's URL with the blog posting's ID (`/blog-postings/{blogPostingId}`). For example, the URL for such a request to the blog posting in the above response is `/blog-postings/59301`. Here's an example response:

```
{
  "alternativeHeadline": "The power of OpenAPI & Liferay",
  "articleBody": "<p>We are happy to announce...</p>",
  "creator": {
    "familyName": "Test",
    "givenName": "Test",
    "id": 20130,
    "name": "Test Test",
    "profileURL": "/web/test"
  },
  "dateCreated": "2019-04-22T07:04:47Z",
  "dateModified": "2019-04-22T07:04:51Z",
  "datePublished": "2019-04-22T07:02:00Z",
  "encodingFormat": "text/html",
  "friendlyUrlPath": "new-headless-apis",
  "headline": "New Headless APIs",
  "id": 59301,
  "numberOfComments": 0,
  "siteId": 20124
}
```

Although this response is JSON, the API's consumer can select other formats to use (like XML). For more information, see API Formats and Content Negotiation.

Related Topics

Get Started: Discover the API

API Formats and Content Negotiation

66.4 Making Authenticated Requests

To make an authenticated request, you must authenticate as a specific user.

There are two authentication mechanisms available when invoking web APIs:

Basic Authentication: Sends the user credentials as an encoded user name and password pair. This is the simplest authentication protocol (available since HTTP/1.0).

OAuth 2.0: In 7.0, you can use OAuth 2.0 for authentication. See the OAuth 2.0 documentation for more information.

First, you'll learn how send requests with basic authentication.

Basic Authentication

Basic authentication requires that you send an HTTP `Authorization` header containing the encoded user name and password. You must first get that encoded value. To do so, you can use `openssl` or a `Base64` encoder. Either way, you must encode the `user:password` string. Here's an example of the `openssl` command for encoding the `user:password` string for a user `test@example.com` with the password `Liferay`:

```
openssl base64 <<< test@example.com:Liferay
```

This returns the encoded value:

```
dGVzdEBleGFtcGxlLmNvbTpMaWZlcmF5Cg==
```

If you don't have `openssl` installed, try the `base64` command:

```
base64 <<< test@example.com:Liferay
```

Warning: Encoding a string as shown here does not encrypt the resulting string. Such an encoded string can easily be decoded by executing `base64 <<< the-encoded-string`, which returns the original string.

Anyone listening to your request could therefore decode the `Authorization` header and reveal your user name and password. To prevent this, ensure that all communication is made through HTTPS, which encrypts the entire message (including headers).

Use the encoded value for the HTTP Authorization header when sending the request:

```
curl -H "Authorization: Basic dGVzdEBleGFtcGxlLmNvbTpMaWZlcmF5Cg==" http://localhost:8080/o/headless-delivery/v1.0/sites/{siteId}/blog-postings/
```

The response contains data instead of the 403 error that an unauthenticated request receives. For more information on the response's structure, see Working with Collections of Data.


```
{
  "items": [
    {
      "alternativeHeadline": "The power of OpenAPI & Liferay",
      "articleBody": "<p>We are happy to announce...</p>",
      "creator": {
        "familyName": "Test",
        "givenName": "Test",
        "id": 20130,
        "name": "Test Test",
        "profileURL": "/web/test"
      },
      "dateCreated": "2019-04-22T07:04:47Z",
      "dateModified": "2019-04-22T07:04:51Z",
      "datePublished": "2019-04-22T07:02:00Z",
      "encodingFormat": "text/html",
      "friendlyUrlPath": "new-headless-apis",
      "headline": "New Headless APIs",
      "id": 59301,
```

```
      "numberOfComments": 0,
      "siteId": 20124
    },
    {
      "alternativeHeadline": "How to work with OAuth",
      "articleBody": "<p>To configure OAuth...</p>",
      "creator": {
        "familyName": "Test",
        "givenName": "Test",
        "id": 20130,
        "name": "Test Test",
        "profileURL": "/web/test"
      },
      "dateCreated": "2019-04-22T09:35:09Z",
      "dateModified": "2019-04-22T09:35:09Z",
      "datePublished": "2019-04-22T09:34:00Z",
      "encodingFormat": "text/html",
      "friendlyUrlPath": "authenticated-requests",
      "headline": "Authenticated requests",
      "id": 59309,
      "numberOfComments": 0,
      "siteId": 20124
    }
  ],
  "lastPage": 1,
  "page": 1,
  "pageSize": 20,
  "totalCount": 2
}
```


OAuth 2.0 Authentication

7.0 supports authorization via OAuth 2.0, which is a token-based authentication mechanism. For more details, see Liferay DXP's OAuth 2.0 documentation. The following sections show you how to use OAuth 2.0 to authenticate web API requests.

Obtaining the OAuth 2.0 Token

Before using OAuth 2.0 to invoke a web API, you must register your application (your web API's consumer) as an authorized OAuth client. To do this, follow the instructions in the Creating an Application section of the OAuth 2.0 documentation. When creating the application, fill in the form as follows:

Application Name: Your application's name.

Client Profile: Headless Server.

Allowed Authorization Types: Check *Client Credentials*.

After clicking *Save* to finish creating the application, write down the Client ID and Client Secret values that appear at the top of the form.

Next, you must get an OAuth 2.0 access token. To do this, see the tutorial Authorizing Account Access with OAuth 2.

Invoking the Service with an OAuth 2.0 Token

Once you have a valid OAuth 2.0 token, include it in the request's `Authorization` header, specifying that the authentication type is a bearer token. For example:

```
curl -H "Authorization: Bearer d5571ff781dc555415c478872f0755c773fa159" http://localhost:8080/o/headless-delivery/v1.0/sites/{siteId}/blog-postings/
```

The response contains the resources that the authenticated user has permission to access, just like the response from Basic authentication.

Making Unauthenticated Requests

Unauthenticated requests are disabled by default in Liferay DXP's headless REST APIs. You can, however, enable them manually by following these steps:

1. Create the config file `com.liferay.headless.delivery.internal.jaxrs.application.HeadlessDeliveryApplicatio default.config` and add this code to it:

```
oauth2.scopechecker.type="none"
auth.verifier.auth.verifier.BasicAuthHeaderAuthVerifier.urls.includes="*"
auth.verifier.auth.verifier.OAuth2RestAuthVerifier.urls.includes="*"
auth.verifier.guest.allowed="true"

Note that the last property (`auth.verifier.guest.allowed`) lets guests
access public content via the APIs. To turn this off, set the property to
`false`.
```

2. Deploy the config file to `[Liferay Home]/osgi/configs`. Note that Liferay Home is typically the application server's parent folder.

3. Test the APIs by making a request to an OpenAPI profile URL:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/openapi.yaml"
```

You should get the OpenAPI profile for the API you sent the request to.

Related Topics

Get Started: Invoke a Service
Working with Collections of Data

66.5 Working with Collections of Data

Collection resources are common in Liferay DXP web APIs. If you followed along with the previous examples that sent requests to the portal's `blog-postings` resource URL, you've already seen collections in action: the `BlogPosting` resource is a collection.

Here, you'll learn more detailed information about working with collection resources. But first you should learn about how collections are returned in pages.

Pagination

A small collection can be transmitted in a single response without difficulty. Transmitting a large collection all at once, however, can consume too much bandwidth, time, and memory. It can also overwhelm the user with too much data. It's therefore best to get and display the elements of a large collection in discrete chunks, or pages.

Liferay DXP's headless REST APIs return paginated collections by default. The following attributes in the responses also contain the information needed to navigate between those pages:

`totalCount`: The total number of this resource's items.

`pageSize`: The number of this resource's items to be included in this response.

`page`: The current page's number.

`lastPage`: The last page's number.

`items`: The collection elements present in this page. Each element also contains the data of the object it represents, so there's no need for additional requests for individual elements.

`id`: Each item's identifier. You can use this, if necessary, to get more information on a specific item.

For examples of working with collection pages, see Pagination.

66.6 Getting Collections

Requests for collection resources are the same as those for non-collection resources. For example, an authenticated request to the `UserAccount` endpoint returns a collection containing the portal's users. When sending this request, use the credentials of an administrative user who has permission to view other portal users:

```
curl "http://localhost:8080/o/headless-admin-user/v1.0/user-accounts"  -u 'test@example.com:test'
```

The response (below) has two main parts:

- The list of collection elements, inside the `items` attribute. This example contains data on two users: an administrator (Test), and a user named Javier Gamarra.
- A set of metadata about the collection. This is the rest of the data in the response. This lets clients know how to use the collection.

This response is in JSON, which is the default response format for web APIs in Liferay DXP. For information on specifying other response formats, see API Formats and Content Negotiation.


```
{
  "items": [
    {
      "alternateName": "test",
      "birthDate": "1970-01-01T00:00:00Z",
      "contactInformation": {},
      "dashboardURL": "/user/test",
      "dateCreated": "2019-04-17T20:37:19Z",
      "dateModified": "2019-04-22T09:56:35Z",
      "emailAddress": "test@example.com",
      "familyName": "Test",
      "givenName": "Test",
      "id": 20130,
      "name": "Test Test",
      "profileURL": "/web/test",
      ...
    },
    {
      "alternateName": "nhpatt",
      "birthDate": "1970-01-01T00:00:00Z",
      "contactInformation": {},
      "dateCreated": "2019-04-22T10:38:36Z",
      "dateModified": "2019-04-22T10:38:37Z",
      "emailAddress": "nhpatt@gmail.com",
      "familyName": "Gamarra",
      "givenName": "Javier",
      "id": 59347,
```

```
      "name": "Javier Gamarra",
      ...
    }
  ],
  "lastPage": 1,
  "page": 1,
  "pageSize": 20,
  "totalCount": 2
}
```

Related Topics

Pagination
    Making Authenticated Requests
    API Formats and Content Negotiation

66.7 Pagination

Collection resources are returned in pages of information. Working with Collections of Data explains this in more detail. Here, you'll learn how to work with collection pages.

For example, suppose that there are 123 users your portal and you want to get information on them. To do this, send an authenticated request to the UserAccount URL:

```
curl "http://localhost:8080/o/headless-admin-user/v1.0/user-accounts"  -u 'test@example.com:test'
```

The response contains the first 30 users and IDs for navigating the rest of the collection. Note that most of the contents of the `items` attribute, which contains the users, are omitted here so you can focus on the metadata for navigating the collection:

```
{
     "items": [
       {
         "id": 20130,
         ...
       },
       {
         "id": 59347,
         ...
       }
     ],
     "lastPage": 5,
     "page": 1,
     "pageSize": 30,
     "totalCount": 123
   }
```

The attributes `page` and `pageSize` allow client applications to navigate through the results. For example, such a client could send a request for a specific page. This example gets the second page (`?page=2`) of documents that exist on the site with the ID `20124`:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/sites/20124/documents?page=2"  -u 'test@example.com:test'
```

Similarly, you can customize the number of elements per page via the optional parameter `pageSize` (e.g., `?pageSize=20`).

Related Topics

Working with Collections of Data
    Making Authenticated Requests

66.8 Navigating from a Collection to its Elements

When you get a collection, you can use the response to get an element of that collection. Follow these steps to do so:

1. Get a collection. This example gets a list of users by sending an authenticated request to the `user-accounts` collection:

   ```
   curl "http://localhost:8080/o/headless-admin-user/v1.0/user-accounts"  -u 'test@example.com:test'
   ```

   Recall from Getting Collections that the response's `items` attribute contains the collection elements. In this case, the collection contains two users: Test Test and Javier Gamarra:

   ```
   json {     "totalItems": 2,     "numberOfItems": 2,     "view": {         {           "items":
   [             {               "alternateName": "test",               "birthDate": "1970-
   01-01T00:00:00Z",               "contactInformation": {},               "dashboardURL":
   "/user/test",               "dateCreated": "2019-04-17T20:37:19Z",               "dateModified":
   "2019-04-22T09:56:35Z",               "emailAddress": "test@example.com",               "familyName":
   "Test",               "givenName": "Test",               "id": 20130,               "name":
   "Test Test",               "profileURL": "/web/test",               "roleBriefs": [
   {                   "id": 20108,                   "name": "Administrator"                },
   {                   "id": 20111,                   "name": "Power User"                },
   {                   "id": 20112,                   "name": "User"                }
   ],               "siteBriefs": [                {                   "id": 20128,
   "name": "Global"                },                {                   "id": 20124,
   "name": "Guest"                }              ]            },            {               "alternate
   "nhpatt",               "birthDate": "1970-01-01T00:00:00Z",               "contactInformation":
   {},               "dateCreated": "2019-04-22T10:38:36Z",               "dateModified":
   "2019-04-22T10:38:37Z",               "emailAddress": "nhpatt@gmail.com",               "familyName":
   "Gamarra",               "givenName": "Javier",               "id": 59347,               "name":
   "Javier Gamarra",               "roleBriefs": [                {                   "id":
   20112,                   "name": "User"                }              ],              "siteBriefs":
   [                {                   "id": 20128,                   "name": "Global"
   },                {                   "id": 20124,                   "name": "Guest"
   }              ]            }          ],          "lastPage": 1,          "page":
   1,          "pageSize": 20,          "totalCount": 2         }     }
   ```

2. In the response, locate the ID of the element you want and look in the OpenAPI profile for the appropriate GET item endpoint. For example, the `user-accounts` GET item endpoint is `/user-accounts/{userAccountId}`.

3. Send a GET request to that endpoint. For example, this request gets information for the user with the ID `59347` (Javier Gamarra):

   ```
   curl "http://localhost:8080/o/headless-admin-user/v1.0/user-accounts/59347"  -u 'test@example.com:test'
   ```

Related Topics

Getting Collections
    Pagination
    Making Authenticated Requests

66.9 API Formats and Content Negotiation

The responses in the preceding examples use a standard JSON format, which is the default response format for Liferay DXP's headless REST APIs. You can also use other formats like XML. Formats typically differ in the resource metadata's structure or semantics. There's no best format; use the one that best fits your use case.

You use *content negotiation* to specify different formats for use. Content negotiation is how the client and server establish the format they use to exchange messages. The client tells the server its preferred format via the HTTP headers `Accept` and `Content-Type`. Each format has a string identifier (its MIME type) that you can use in the HTTP headers to specify the format. The following table lists the MIME type for each supported format.

API Format	MIME Type
application/json	application/json
application/xml	application/xml

When you send a request without specifying the API format, the server responds with the default JSON. For example, here's such a request for a list of folders from the Site with the ID `20124`:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/sites/20124/document-folders" -u 'test@example.com:test'
```


```
{
  "items": [
    {
      "creator": {
        "familyName": "Test",
        "givenName": "Test",
        "id": 20130,
        "name": "Test Test",
        "profileURL": "/web/test"
      },
      "dateCreated": "2019-04-22T10:21:20Z",
      "dateModified": "2019-04-22T10:21:20Z",
      "id": 59319,
      "name": "REST APIs Documentation",
      "numberOfDocumentFolders": 0,
      "numberOfDocuments": 0,
      "siteId": 20124
    }
  ],
  "lastPage": 1,
  "page": 1,
  "pageSize": 20,
  "totalCount": 1
}
```

If you request the headers, the `Content-Type` response attribute lists the content type's format (JSON, in this case):

```
curl "http://localhost:8080/o/headless-delivery/v1.0/sites/20124/document-folders" -u 'test@example.com:test' --head

HTTP/1.1 200
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1
Set-Cookie: JSESSIONID=9F61AEB8721DD9149BD577ECBC31AE3F; Path=/; HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Cache-Control: private, no-cache, no-store, must-revalidate
Pragma: no-cache
Set-Cookie: COOKIE_SUPPORT=true; Max-Age=31536000; Expires=Tue, 21-Apr-2020 10:23:57 GMT; Path=/; HttpOnly
Set-Cookie: GUEST_LANGUAGE_ID=en_US; Max-Age=31536000; Expires=Tue, 21-Apr-2020 10:23:57 GMT; Path=/; HttpOnly
Date: Mon, 22 Apr 2019 10:23:57 GMT
Content-Type: application/json
Transfer-Encoding: chunked
```

To get the response in XML instead, specify `application/xml` in the request's `Accept` header. Note that the XML response includes the same information as JSON, but is structured differently:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/documents/59203"  -H 'Accept: application/xml'  -u 'test@example.com:test'

    <Page>
        <items>
            <items>
                <creator>
                    <familyName>Test</familyName>
                    <givenName>Test</givenName>
                    <id>20130</id>
                    <name>Test Test</name>
                    <profileURL>/web/test</profileURL>
                    </creator>
                <dateCreated>2019-04-22T10:21:20Z</dateCreated>
                <dateModified>2019-04-22T10:21:20Z</dateModified>
                <id>59319</id>
                <name>REST APIs Documentation</name>
                <numberOfDocumentFolders>0</numberOfDocumentFolders>
                <numberOfDocuments>0</numberOfDocuments>
                <siteId>20124</siteId>
            </items>
        </items>
        <lastPage>1</lastPage>
        <page>1</page>
        <pageSize>20</pageSize>
        <totalCount>1</totalCount>
    </Page>
```

Requesting the headers, you can see that the response is in XML (`application/xml`):

```
curl "http://localhost:8080/o/headless-delivery/v1.0/documents/59203"  -H 'Accept: application/xml'  -u 'test@example.com:test' --
head

HTTP/1.1 200
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Cache-Control: private, no-cache, no-store, must-revalidate
Pragma: no-cache
Date: Mon, 22 Apr 2019 10:26:21 GMT
Content-Type: application/xml
Transfer-Encoding: chunked
```

Language Negotiation

The same mechanism used for requesting another response format (content negotiation) is used for requesting content in another language.

APIs that are available in different languages return the options in a block called `availableLanguages`. For example, this block in the following response lists U.S. English (`en-US`) and Spain/Castilian Spanish (`es-ES`):

```
{
  "availableLanguages": [
    "en-US",
    "es-ES"
  ],
  "contentFields": [
    {
      "dataType": "html",
      "name": "content",
      "repeatable": false,
      "value": {
        "data": "<p>The main reason is because Headless APIs have been designed with real use cases in mind...</p>"
      }
    }
  ],
  "contentStructureId": 36801,
  "creator": {
    "familyName": "Test",
    "givenName": "Test",
    "id": 20130,
    "name": "Test Test",
    "profileURL": "/web/test"
  },
  "dateCreated": "2019-04-22T10:29:40Z",
  "dateModified": "2019-04-22T10:30:31Z",
  "datePublished": "2019-04-22T10:28:00Z",
  "friendlyUrlPath": "why-headless-apis-are-better-than-json-ws-services-",
  "id": 59325,
  "key": "59323",
  "numberOfComments": 0,
  "renderedContents": [
    {
      "renderedContentURL": "http://localhost:8080/o/headless-delivery/v1.0/structured-contents/59325/rendered-content/36804",
      "templateName": "Basic Web Content"
    }
  ],
  "siteId": 20124,
  "title": "Why Headless APIs are better than JSON-WS services?",
  "uuid": "e1c4c152-e47c-313f-2d16-2ee4eba5cd26"
}
```

To request the content in another language, specify your desired locale in the request's `Accept-Language` header:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/structured-contents/59325"  -H 'Accept-Language: es-ES'  -u 'test@example.com:test'
```

```
    {
      "availableLanguages": [
        "en-US",
        "es-ES"
      ],
      "contentFields": [
        {
          "dataType": "html",
          "name": "content",
          "repeatable": false,
```

```
      "value": {
        "data": "<p>La principal razón es porque las APIs Headless se han diseñado pensando en casos de uso reales...</p>"
      }
    }
  ],
  "contentStructureId": 36801,
  "creator": {
    "familyName": "Test",
    "givenName": "Test",
    "id": 20130,
    "name": "Test Test",
    "profileURL": "/web/test"
  },
  "dateCreated": "2019-04-22T10:29:40Z",
  "dateModified": "2019-04-22T10:30:31Z",
  "datePublished": "2019-04-22T10:28:00Z",
  "friendlyUrlPath": "%C2%BFpor-qu%C3%A9-las-apis-headless-son-mejores-que-json-ws-",
  "id": 59325,
  "key": "59323",
  "numberOfComments": 0,
  "renderedContents": [
    {
      "renderedContentURL": "http://localhost:8080/o/headless-delivery/v1.0/structured-contents/59325/rendered-content/36804",
      "templateName": "Contenido web básico"
    }
  ],
  "siteId": 20124,
  "title": "¿Por qué las APIs Headless son mejores que JSON-WS?",
  "uuid": "e1c4c152-e47c-313f-2d16-2ee4eba5cd26"
}
```


Creating Content with Different Languages

By default, when sending a POST/PUT request, the `Accept-Language` header is used as the content's language. However, there is one exception. Some entities require the first POST to be in the Site's default language. In such cases, a POST request for a different language results in an error.

After creating a new resource, PUT requests in a different language add that translation. PATCH requests return an error (you are expected to update, not create, in a PATCH request).

Related Topics

Get Started: Discover the API

Get Started: Invoke a Service

66.10 OpenAPI Profiles

All the APIs exposed by Liferay DXP are available under the liferayinc SwaggerHub organization.

Liferay DXP's headless APIs are categorized in two different use cases:

- Delivering content (delivery APIs)
- Managing and administering content (admin APIs)

The available APIs demonstrate this categorization.

Headless Delivery

The following table lists the APIs that Headless Delivery contains. Note that the second column shows which internal model in Liferay DXP that the API maps to.

API	Internal Model
BlogPosting	BlogsEntry
BlogPostingImage	DLFileEntry (associated with a BlogsEntry)
Comment	DiscussionComment
ContentDocument	DLFileEntry (associated with a JournalArticle)
ContentSet	AssetListEntry
ContentStructure	DDMStructure
Document	DLFileEntry
DocumentFolder	Folder
KnowledgeBaseArticle	KBArticle
KnowledgeBaseAttachment	FileEntry (associated with a KBArticle)
KnowledgeBaseFolder	KBFolder
MessageBoardAttachment	FileEntry (associated with a MBMessage)
MessageBoardMessage	MBMessage
MessageBoardSection	MBCategory
MessageBoardThread	MBThread
Rating	RatingsEntry
StructuredContent	JournalArticle
StructuredContentFolder	JournalFolder

Headless Administration

There are several headless admin APIs, each containing its own set of APIs. The following tables list these, as well as any internal models in Liferay DXP that each API maps to.

Headless Admin User contains the following APIs for retrieving and managing information about users and organizations.

API	Internal Model
EmailAddress	N/A
Organization	N/A
Phone	N/A
PostalAddress	Address
Role	N/A
Segment	SegmentEntry
SegmentUser	N/A
SiteBrief	N/A
UserAccount	User
WebUrl	WebSite

Headless Admin Taxonomy contains the following APIs for managing asset categories, asset vocabularies, and asset tags.

API	Internal Model
`Keyword`	`AssetTag`
`TaxonomyCategory`	`AssetCategory`
`TaxonomyVocabulary`	`AssetVocabulary`

Headless Admin Workflow contains APIs for transitioning workflows.

Related Topics

API Formats and Content Negotiation

66.11 Filter, Sort, and Search

You can use Liferay DXP's headless REST APIs to search for content you're interested in. You can also sort and filter content. Here, you'll learn how.

Filter

It's often useful to filter large collections for the exact data that you need. Not all collections, however, allow filtering. The ones that support it contain the optional parameter `filter` in their OpenAPI profile. To filter a collection based on the value of one or more fields, use the `filter` parameter following a subset of the oData standard.

Filtering mainly applies to fields indexed as keywords in Liferay DXP's search. To find content by terms contained in fields indexed as text, you should instead use search.

Comparison Operators

Operator | Description | Example | `eq` | Equal | `addressLocality eq 'Redmond'` | | Equal null | `addressLocality eq null` | `ne` | Not equal | `addressLocality ne 'London'` | | Not null | `addressLocality ne null` | `gt` | Greater than | `price gt 20` | `ge` | Greater than or equal | `price ge 10` | `lt` | Less than | `dateCreated lt 2018-02-13T12:33:12Z` | `le` | Less than or equal | `dateCreated le 2012-05-29T09:13:28Z` | `startsWith` | Starts with | `addressLocality startsWith 'Lond'` |

Logical Operators

Operator	Description	Example
`and`	Logical and	`price le 200 and price gt 3.5`

Operator	Description	Example
`or`	Logical or	`price le 3.5 or price gt 200`
`not`	Logical not	`not (price le 3.5)`

Note that the `not` operator needs a space character after it.

Grouping Operators

Operator | Description | Example | `( )` | Precedence grouping | `(price eq 5) or (addressLocality eq 'London')` |

String Functions

Function	Description	Example
`contains`	Contains	`contains(title,'edmon')`

Lambda Operators

Lambda operators evaluate a boolean expression on a collection. They must be prepended with a navigation path that identifies a collection.

Lambda Operator	Description	Example
`any`	Any	`keywords/any(k:contains(k,'substring1'))`

The `any` operator applies a boolean expression to each collection element and evaluates to `true` if the expression is true for any element.

Escaping in Queries

You can escape a single quote in a value by adding another single quote. For example, to filter for a blog posting whose headline is `New Headless APIs`, append this filter string to the request URL:

```
?filter=headline eq 'New Headless APIs'
```

Here's an example of the full request:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/sites/20124/blog-postings/?filter=headline%20eq%20%27New%20Headless%20APIs%27"  -
u 'test@example.com:test'
```

```
{
  "items": [
    {
      "alternativeHeadline": "The power of OpenAPI & Liferay",
      "articleBody": "<p>We are happy to announce...</p>",
      "creator": {
        "familyName": "Test",
        "givenName": "Test",
        "id": 20130,
        "name": "Test Test",
        "profileURL": "/web/test"
      },
      "dateCreated": "2019-04-22T07:04:47Z",
      "dateModified": "2019-04-22T07:04:51Z",
      "datePublished": "2019-04-22T07:02:00Z",
      "encodingFormat": "text/html",
      "friendlyUrlPath": "new-headless-apis",
      "headline": "New Headless APIs",
      "id": 59301,
      "numberOfComments": 0,
      "siteId": 20124
    }
  ],
  "lastPage": 1,
  "page": 1,
  "pageSize": 20,
  "totalCount": 1
}
```


Filtering in Structured Content Fields (ContentField)

To filter for a `ContentField` value (dynamic values created by the end user), you must use the endpoints that are scoped to an individual `ContentStructure`. To do so, find the ID of the `ContentStructure` and use it in place of `{contentStructureId}` in this URL:

```
"/content-structures/{contentStructureId}/structured-contents"
```

Search

It's often useful to search large collections with keywords. Use search when you want results from any field, rather than specific ones. To perform a search, use the optional parameter `search` followed by the search terms. For example, this request searches for all the `BlogEntry` fields containing OAuth:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/sites/20124/blog-postings/?search=OAuth"  -u 'test@example.com:test'
```


```
{
  "items": [
    {
      "alternativeHeadline": "How to work with OAuth",
      "articleBody": "<p>To configure OAuth...</p>",
      "creator": {
        "familyName": "Test",
        "givenName": "Test",
        "id": 20130,
        "name": "Test Test",
        "profileURL": "/web/test"
      },
      "dateCreated": "2019-04-22T09:35:09Z",
      "dateModified": "2019-04-22T09:35:09Z",
      "datePublished": "2019-04-22T09:34:00Z",
      "encodingFormat": "text/html",
      "friendlyUrlPath": "authenticated-requests",
      "headline": "Authenticated requests",
```

```
      "id": 59309,
      "numberOfComments": 0,
      "siteId": 20124
    }
  ],
  "lastPage": 1,
  "page": 1,
  "pageSize": 20,
  "totalCount": 1
}
```

Sorting

Sorting collection results is another common task. Note, however, that not all collections allow sorting. The ones that support it contain the optional parameter {lb}?sort{rb} in their OpenAPI profile.

To get sorted collection results, append ?sort=<param-name> to the request URL. For example, appending ?sort=title to the request URL sorts the results by title.

The default sort order is ascending (0-1, A-Z). To perform a descending sort, append :desc to the parameter name. For example, to perform a descending sort by title, append ?sort=title:desc to the request URL.

To sort by more than one parameter, separate the parameter names by commas and put them in order of priority. For example, to sort first by title and then by creation date, append ?sort=title,dateCreated to the request URL.

To specify a descending sort for only one parameter, you must explicitly specify ascending sort order (:asc) for the other parameters. For example:

```
?sort=headline:desc,dateCreated:asc
```

Flatten

Some collections (as defined in their OpenAPI profile) allow the query parameter flatten, which returns all resources and disregards folders or other hierarchical classifications. This parameter's default value is false, so a document query to the root folder returns only the documents in that folder. With flatten set to true, the same query also returns documents in any subfolders, regardless of how deeply those folders are nested. In other words, setting flatten set to true and querying for documents in a Site's root folder returns all the documents in the Site.

Related Topics

Making Authenticated Requests
    API Formats and Content Negotiation
    Working with Collections of Data

66.12 Restrict Properties

Retrieving large entities or collections increases the response's size and uses more bandwidth. You can alleviate this by telling the server via the request which fields it should include in the response. This is known as *sparse fieldsets*. To make a request with sparse fieldsets, include the fields parameter in the URL with the name of each field's attribute.

For example, this request doesn't use sparse fieldsets and therefore returns all the fields of a blog posting:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/blog-postings/59301"  -u 'test@example.com:test'
```

```
{
  "alternativeHeadline": "The power of OpenAPI & Liferay",
  "articleBody": "<p>We are happy to announce...</p>",
  "creator": {
    "familyName": "Test",
    "givenName": "Test",
    "id": 20130,
    "name": "Test Test",
    "profileURL": "/web/test"
  },
  "dateCreated": "2019-04-22T07:04:47Z",
  "dateModified": "2019-04-22T07:04:51Z",
  "datePublished": "2019-04-22T07:02:00Z",
  "encodingFormat": "text/html",
  "friendlyUrlPath": "new-headless-apis",
  "headline": "New Headless APIs",
  "id": 59301,
  "numberOfComments": 0,
  "siteId": 20124
}
```

To get only the headline, creation date, and creator, append the `fields` parameter to the URL with the fields `headline`, `dateCreated`, and `creator`:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/blog-postings/59301?fields=headline,dateCreated,creator"  -u 'test@example.com:test'
```

```
{
  "creator": {
    "familyName": "Test",
    "givenName": "Test",
    "id": 20130,
    "name": "Test Test",
    "profileURL": "/web/test"
  },
  "dateCreated": "2019-04-22T07:04:47Z",
  "headline": "New Headless APIs"
}
```

In the response, the `creator` attribute is a nested JSON object. To return only the creator's name, specify that nested field via dot notation (`creator.name`):

```
curl "http://localhost:8080/o/headless-delivery/v1.0/blog-postings/59301?fields=headline,dateCreated,creator.name"  -u 'test@example.com:test'
```

```
{
  "creator": {
    "name": "Test Test"
  },
  "dateCreated": "2019-04-22T07:04:47Z",
  "headline": "New Headless APIs"
}
```

The `fields` parameter also works with collection resources to return the specified attributes for every collection item. For example, this request gets the headlines for all the blog postings in the Site with the ID `20124`:

```
curl "http://localhost:8080/o/headless-delivery/v1.0/sites/20124/blog-postings/?fields=headline"  -u 'test@example.com:test'
```

```
{
  "items": [
    {
      "headline": "New Headless APIs"
    },
    {
      "headline": "Authenticated requests"
    }
  ],
  "lastPage": 1,
  "page": 1,
  "pageSize": 20,
  "totalCount": 2
}
```

Related Topics

Making Authenticated Requests
API Formats and Content Negotiation
Working with Collections of Data

66.13 Multipart Requests

Several operations accept a binary file via a multipart request. For example, the definition for posting a file to a `DocumentFolder` specifies a multipart request:

```
post:
    operationId: postDocumentFolderDocument
    parameters:
        - in: path
          name: documentFolderId
          required: true
          schema:
              format: int64
              type: integer
    requestBody:
        content:
            multipart/form-data:
                schema:
                    properties:
                        document:
                            $ref: "#/components/schemas/Document"
                        file:
                            format: binary
                            type: string
                    type: object
    responses:
        200:
            content:
                application/json:
                    schema:
                        $ref: "#/components/schemas/Document"
                application/xml:
                    schema:
                        $ref: "#/components/schemas/Document"
            description: ""
    tags: ["Document"]
```

This operation returns a `Document` (in JSON or XML). To create this `Document`, you must supply the operation's multipart request with 2 components:

- A binary file (bytes) via the `file` property
- A JSON string with the binary file's metadata, via the `document` property

To send this request, the `Content-Type` must be `multipart/form-data`, and you must also specify a boundary name (the boundary name can be arbitrary).

Here's an example request (without the file's bytes) that creates a document in the folder with the ID `38549`:

```
curl -X "POST" "http://localhost:8080/o/headless-delivery/v1.0/document-folders/38549/documents" \
     -H 'Accept: application/json' \
     -H 'Content-Type: multipart/form-data; boundary=PART' \
     -u 'test@example.com:test' \
     -F "file=" \
     -F "document={\"title\": \"podcast\"}"
```

And here's the response:

```
{
  "contentUrl": "/documents/20123/38549/podcast.mp3/e978e316-620c-df9f-e0bd-7cc0447cca49?version=1.0&t=1556100111417",
  "creator": {
    "familyName": "Test",
    "givenName": "Test",
    "id": 20129,
    "name": "Test Test",
    "profileURL": "/web/test"
  },
  "dateCreated": "2019-04-24T10:01:51Z",
  "dateModified": "2019-04-24T10:01:51Z",
  "documentFolderId": 38549,
  "encodingFormat": "audio/mpeg",
  "fileExtension": "mp3",
  "id": 38553,
  "numberOfComments": 0,
  "sizeInBytes": 28482097,
  "title": "podcast"
}
```

Related Topics

Making Authenticated Requests

API Formats and Content Negotiation

Working with Collections of Data

CHAPTER 67

SERVICE BUILDER WEB SERVICES

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 Service Builder can also generate remote web services automatically. Here, you'll learn how to both generate and invoke remote JSON and SOAP web services. Because Liferay's web services are created this way, knowing how to invoke these services opens many development possibilities. 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.
- 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.

67.1 Creating Remote Services

Many default Liferay DXP services are published as JSON and SOAP web services. If you run the portal locally on port 8080, visit this URL to browse the default JSON web services:

```
http://localhost:8080/api/jsonws/
```

Visit this URL to browse the default SOAP web services:

```
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' 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 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.

For example, consider Web Content articles. Web Content 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 the best practice of checking permissions and calling the corresponding local service method. For example, each `addArticle` method in `JournalArticleServiceImpl` checks permissions and then calls the local service's matching `addArticle` method:

```
@Override
public JournalArticle addArticle(...)
    throws PortalException {

    ModelResourcePermissionHelper.check(
        _journalFolderModelResourcePermission, getPermissionChecker(),
        groupId, folderId, ActionKeys.ADD_ARTICLE);

    return journalArticleLocalService.addArticle(...);
}
```

Note the use of `ModelResourcePermissionHelper.check(...)`. This handy helper class was introduced in Liferay 7.1. For model resource permission checks, you can use this helper class instead of using a custom permissions helper class. 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. Your Service Builder-generated classes do the same thing.

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 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, modify 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 Dev Studio DXP, 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 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 Dev Studio DXP 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, modify 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 Dev Studio DXP:** 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 not satisfying 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 dependencies are often required for portlet development:

```
compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
compileOnly group: "com.liferay", name: "com.liferay.registry.api", version: "2.0.0"
```

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 where 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 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.

Building the WSDD for Built-in 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 Wiki 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/bookmarks/bookmarks-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`. 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
    What is Service Builder?

67.2 Invoking Remote Services

You can invoke the remote services of any installed Liferay application the same way 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 is 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 Web Content app. Notice that the `addArticle` methods invoke `journalArticleLocalService.addArticle` after a permission check.

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 through 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 appear 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 appears 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 use your browser to view the SOAP services of Liferay and Liferay apps 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:

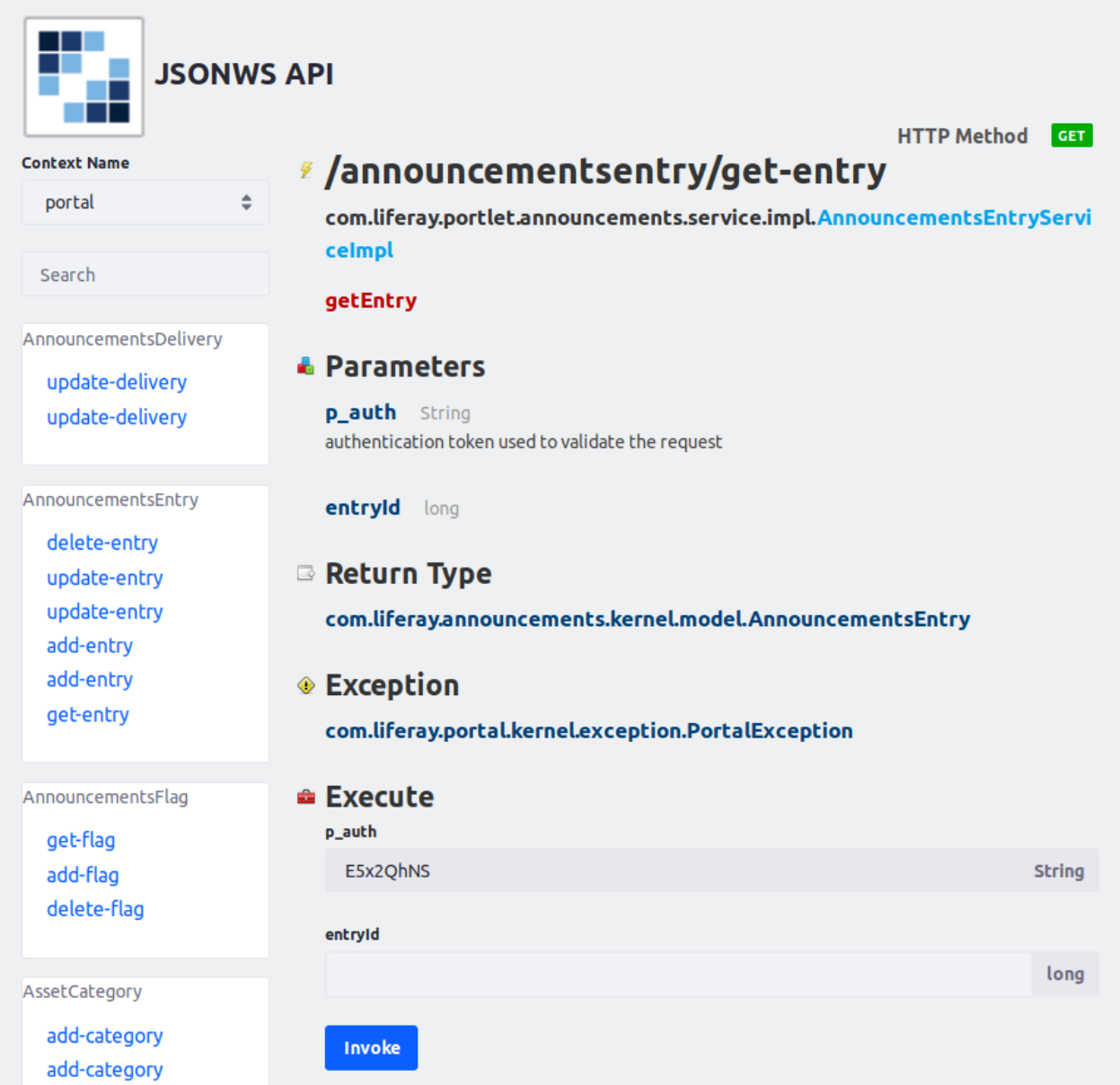


Figure 67.1: The JSON web services page for an entity's remote service method also lets you invoke that service.

```
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 [https://docs.liferay.com/dxp/portal/7.0-latest/javadocs/](### Related Topics

Invoking JSON Web Services

JSON Web Services Invoker

SOAP Web Services

Creating Remote Services

67.3 Service Security Layers

Liferay's remote services are secured by default, because they only allow local connections. Enabling remote access requires peeling away several layers of security, first by IP address, then by user authentication and verification. Users invoking web services 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 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 where the batch job runs. 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 can connect to Liferay's web services, pass in the appropriate user credentials, and upload the documents.

Note: The `portal-ext.properties` file resides on the Liferay server and is controlled by its administrator. 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 describes these properties.

Next, if you invoke the remote service via web services, 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.

The user permission layer is the last security layer triggered for remote services. It's used for every object, regardless of whether a local or remote service is involved. The user ID associated with a web service invocation must have permission to operate on the objects it's trying to access. A remote exception is thrown if the user ID doesn't have permission. 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 with 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 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 web service via a browser. Since you should never pass credentials over the network unencrypted, use HTTPS whenever accessing Liferay services. Most HTTP clients (like 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 supplied automatically 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. Authentication by email address is the default. A user can find his or her user 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 you've defined authentication 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 you've now defined authentication 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 you've defined dictates the authentication type you'll use to access your web service. The authentication type can be set 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 password policies, since they're enforced on your administrative user. If a password policy requires them to change their passwords on a periodic basis, the password for an administrative user accessing web services in a batch job expires too.

To prevent a password from expiring, add a new password policy that doesn't enforce password expiration, and then add your administrative user to the policy. Then your batch job can run as many times as you need it to, without the 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 want 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
SOAP Web Services

67.4 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 the public methods of that interface become registered and available as JSON web services.

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.

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 tutorials.

Next, you'll see how to 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

For example, say your app named `SupraSurf` has some services you want exposed 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 your Liferay instance, 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 this 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 here.

This same mechanism registers Liferay's own services. They're 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 predictably 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`). 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 this 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 a read-only method and is mapped as a *GET HTTP* method by default. All other methods are mapped as *POST HTTP* methods.

In the list of JSON web services at http://localhost:8080/api/jsonws, when you select a method, the part of its HTTP method URL that follows `http://[server]:[port]/api/jsonws` appears at the top of the screen.

Conveniently, remote service requests can use authentication credentials associated with the user's current 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 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 a class-level annotation with *manual mode*. Then it's up to you to annotate only those methods you want to expose:

```
@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 other methods are excluded from the API.

Related Topics

Invoking JSON Web Services

JSON Web Services Invoker

67.5 Invoking JSON Web Services

If you know the URL and are connected to the Internet, you can 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 techniques for working with JSON web services and includes details about invoking them via URL.

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

Figure 67.2: 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.

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.

Using Hints When Invoking a Service via URL

Numeric hints specify how many method arguments a service has. 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 are 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` is automatically 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

Specify parameters in name-value pairs after the service URL. Parameter names must be formed from method argument names by converting them from camel case to all lower case, dash-separated names. 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; you need not follow the order of the arguments in the method signatures.

When a method name is overloaded, the *best match* is 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. You can also 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`) happens directly. 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.

For 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 `Strings` `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. 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 must first 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. 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.

You can extend the JSON-RPC object parameter example above by populating its inner parameters:

```
"+foo" : "com.liferay.impl.FooBean",
"foo.field1" : "test",
"foo.field2" : "true",
"foo.field3" : 123
```

Here's the same with JavaScript (assuming we have a remote service under the `foo` context which accepts one argument with type `com.liferay.impl.FooBean` and it has the specified fields):

```
Liferay.Service(
    '/foo/update-foo',
{
        "+foo": "com.liferay.impl.FooBean",
    "foo.field1" : "test",
    "foo.field2" : "true",
    "foo.field3": 123
    },
    function(obj) {
        console.log(obj);
    }
);
```

Tip: Use inner parameters with object parameters to set inner contents of created object parameter instances!

Next, you'll examine returned values 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.

Here's an example. 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 like this:

```
{
  "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. Also, note that Liferay provides a `JSONFactory` service that allows developers to create JSON objects and arrays, serialize and deserialize JSON strings, and perform other JSON-related operations.

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 with the parameter name in your URL path. The parameter value must follow the parameter name. This is wrong:

  ```
  /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 error means that 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 there 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 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
    Service Security Layers
    Invoking Remote Services

67.6 JSON Web Services Invoker

With JSON web services, you send a request to a service method with 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.

Say you're working with two related objects: a `User` and its corresponding `Contact`. Normally 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. This sends two HTTP requests to get two separate JSON objects. 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.

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 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 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
```

If you're running Liferay locally on port `8080`, here's how you invoke a JSON web service:

1. Collect your credentials. Here's an example:

    - Email: `test@example.com`
    - User ID: `20127`
    - Authorization Token: `htXjvt5d`

2. Invoke the service:

    ```
    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
    }
}
```

In the URL, the double quotes are URL-encoded. To find your user ID, check the User Menu under *My Account → Account Settings*. To find your `p_auth` authentication token, 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.

Use JSON syntax to supply values for objects and arrays as parameters. To supply a value for an object, use curly brackets: { and }. To supply a value for an array, use square brackets: [and].

If you want to pass an array as a parameter using the same credential token as above, here's an example how, using two vocabularies with vocabulary IDs of `20783` and `20784`:

```
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,%22passw
```

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, 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, you invoke multiple services with a single HTTP request. This is helpful for gathering related information from the service call results, but it you can also use a single request to invoke multiple unrelated service calls by batching service calls together to improve performance. Do this by passing in a JSON array of commands:

```
[
    {/* 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

67.7 Configuring JSON Web Services

JSON web services are enabled in Liferay by default. If you must disable them, specify this portal property setting in a `portal-ext.properties` file:

```
json.web.service.enabled=false
```

This tutorial presents other 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, set your Liferay instance's JSON web services to work in read-only mode by disabling HTTP methods other than `GET`:

```
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 call 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*,ha` Liferay makes all read-only JSON methods publicly accessible. All other JSON methods are disabled. 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

67.8 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 this 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
```

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

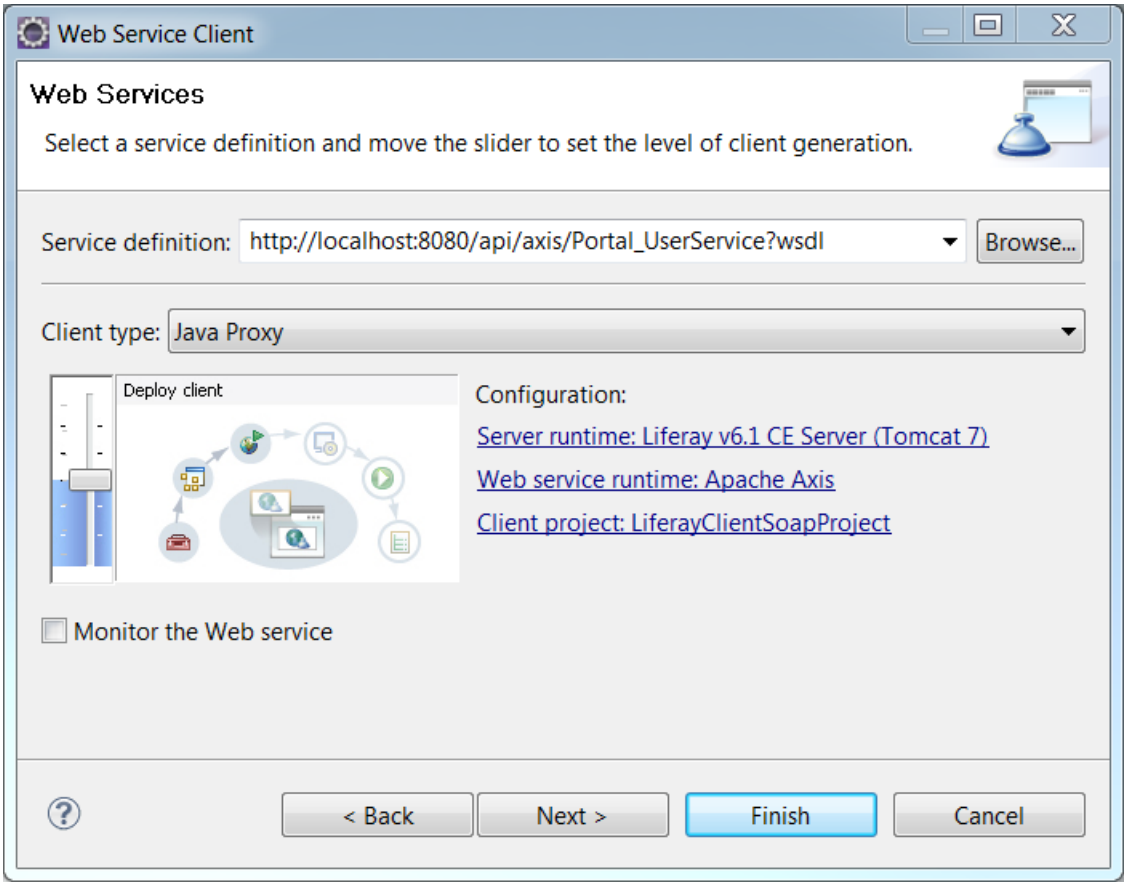


Figure 67.3: Service Definition

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?

CHAPTER 68

JAX-RS AND JAX-WS

You can deploy JAX-RS and JAX-WS web services and consume them much as you would outside of Liferay DXP. There are, however, a few things you should know, particularly with regard to deploying them into Liferay DXP's OSGi container. The tutorials in this section discuss these things and show you how to use JAX-RS and JAX-WS in Liferay DXP.

68.1 JAX-RS

JAX-RS web services work in Liferay modules the same way they work outside of Liferay. The only difference is that you must register the class in the OSGi framework. Liferay's development tools make this easy by providing a template.

In Liferay Developer Studio, create a new module using the *rest* template:

1. Click *File → New → Liferay Module Project*.
2. Give the project a name and select the *rest* template.
3. Select *Next* and enter a class name and a package name for your service.
4. Click *Finish*.

Alternatively, use Blade CLI to create the project.

Note: The initial release of 7.0's development tools created a project with an invalid configuration. If your class contains an `@ApplicationPath` annotation, you must update the following files:

Your Java Class:

1. Add the following import:

```
import org.osgi.service.jaxrs.whiteboard.JaxrsWhiteboardConstants;
```

2. Remove the `@ApplicationPath` annotation.

3. Modify the `@Component` annotation so it looks like this:

```
@Component(
  property = {
  JaxrsWhiteboardConstants.JAX_RS_APPLICATION_BASE + "=/greetings",
  JaxrsWhiteboardConstants.JAX_RS_NAME + "=Greetings.Rest"
  },
```

service = Application.class)

build.gradle:
Add the following dependency:

```
compileOnly group: "org.osgi", name: "org.osgi.service.jaxrs", version: "1.0.0"
```

Files to delete:
Delete everything under `src/main/resources/configuration`.

The class that's generated contains a working JAX-RS web service. You can deploy it and use it immediately.

While it's beyond the scope of this article to cover JAX-RS Whiteboard in its entirety, essentially it's JAX-RS unchanged except for configuration properties in the `@Component` annotation. These properties declare two things:

1. The endpoint for the service
2. The service name as it appears in the OAuth 2.0 configuration

The generated class contains this configuration:

```
@Component(
        property = {
            JaxrsWhiteboardConstants.JAX_RS_APPLICATION_BASE + "=/greetings",
            JaxrsWhiteboardConstants.JAX_RS_NAME + "=Greetings.Rest"
        },
        service = Application.class)
```

This configuration registers the service at this endpoint:

```
https://[server-name]:[port]/o/greetings
```

If you're testing this locally on Tomcat, the URL is

```
https://localhost:8080/o/greetings
```

As you might guess, you don't have access to the service by just calling the URL above. You must authenticate first, which you'll learn how to do next.

Using OAuth 2.0 to Invoke a JAX-RS Web Service

Your JAX-RS web service requires authorization by default. To enable this, you must create an OAuth 2.0 application to provide a way to grant access to your service:

1. Go to the *Control Panel → Configuration → OAuth2 Administration* and click the  button to add an application.
2. Give your application a descriptive name.
3. Choose the Client Profile appropriate for this service. These are templates that auto-select the appropriate authorization types or "flows" from the OAuth 2 standard. For this example choose the *Headless Server* profile, which auto-selects the *Client Credentials* authorization type.
4. Click *Save*.

The form now reappears with two additional generated fields: Client ID and Client Secret. You'll use these to authenticate to your web service.

To make your service accessible,

1. Click the *Scopes* tab.
2. You'll see an entry for your deployed `Greetings.Rest` service. Expand it by clicking the arrow.
3. Check the box labeled *read data on your behalf*.
4. Click *Save*.

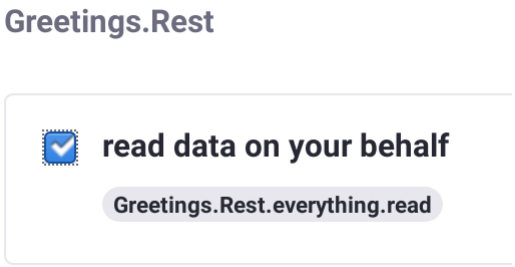


Figure 68.1: Enable the scope to grant access to the service.

For simplicity, the examples below use Curl to authenticate. You need the two pieces of information generated for your application: the Client ID and the Client Secret. For example, say those fields contain these values:

Client ID: `id-12e14a84-e558-35a7-cf9a-c64aafc7f`
Client Secret: `secret-93f14320-dc39-d67f-9dec-97717b814f`

First, you must request an OAuth token. If you're testing locally, you'd make a request like this:

```
curl http://localhost:8080/o/oauth2/token -d 'grant_type=client_credentials&client_id=id-12e14a84-e558-35a7-cf9a-c64aafc7f&client_secret=secret-93f14320-dc39-d67f-9dec-97717b814f'
```

The response is JSON:

```
{"access_token":"a7f12bef7f2e578cf64bce4085db8f17b6a3c2963f865a65b374e89784bbca5","token_type":"Bearer","expires_in":600,"scope":"GET POST PUT"}
```

It contains a token, generated for this client. It expires in 600 seconds, and it grants GET, POST, and PUT for this web service.

When you want to call the service, you must supply the token in the HTTP header, like this:

```
curl --header "Authorization: Bearer a7f12bef7f2e578cf64bce4085db8f17b6a3c2963f865a65b374e89784bbca5" http://localhost:8080/o/greetings/morning
```

With authorization, your web service can be called and responds to the request:

```
Good morning!
```

Of course, this is only one of the authorization flows for OAuth 2.0. If you're creating a web-based client whose back-end is a JAX-RS web service hosted on Liferay DXP, you'd want one of the other flows. See the OAuth 2.0 documentation for further information. Additionally, OAuth 2.0 assumes the use of HTTPS for its security: the above URLs are only for local testing purposes. You certainly would not want to pass OAuth tokens between clients and servers in the clear. Make sure that in production your server uses HTTPS.

Great! Now you know how to create, deploy, and invoke JAX-RS web services on Liferay DXP's platform!

Related Topics

Service Builder Web Services

68.2 JAX-WS

Liferay supports JAX-WS via the Apache CXF implementation. Apps can publish JAX-WS web services to the CXF endpoints defined in your Liferay instance. CXF endpoints are effectively context paths the JAX-WS web services are deployed to and accessible from. To publish any kind of JAX-WS web service, one or more CXF endpoints must be defined. To access JAX-WS 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.

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.

SOAP 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 endpoint `/soap`, you could later create 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 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 web services. The following topics are covered:

- Configuring Endpoints and Extenders with the Control Panel
- Configuring Endpoints and Extenders Programmatically
- Publishing JAX-WS Web Services

Configuring Endpoints and Extenders with the Control Panel

Liferay's Control Panel lets administrators configure endpoints and extenders for JAX-WS 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.

To configure a CXF endpoint with the Control Panel, first go to *Control Panel → Configuration → System Settings → Web API*. Then select *CXF Endpoints* from the list. If there are any existing CXF endpoints, they're shown here. To add a new one, click the *Add* button. The form that appears lets you configure a new CXF endpoint by filling out these fields:

Context Path: The path the JAX-WS 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.

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 → Web API*. Then select *SOAP Extenders* from the list. If there are any existing SOAP extenders, they're shown here. To add a new one, click on the *Add* button. 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 are 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`.

Next, you'll learn how to create 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
```

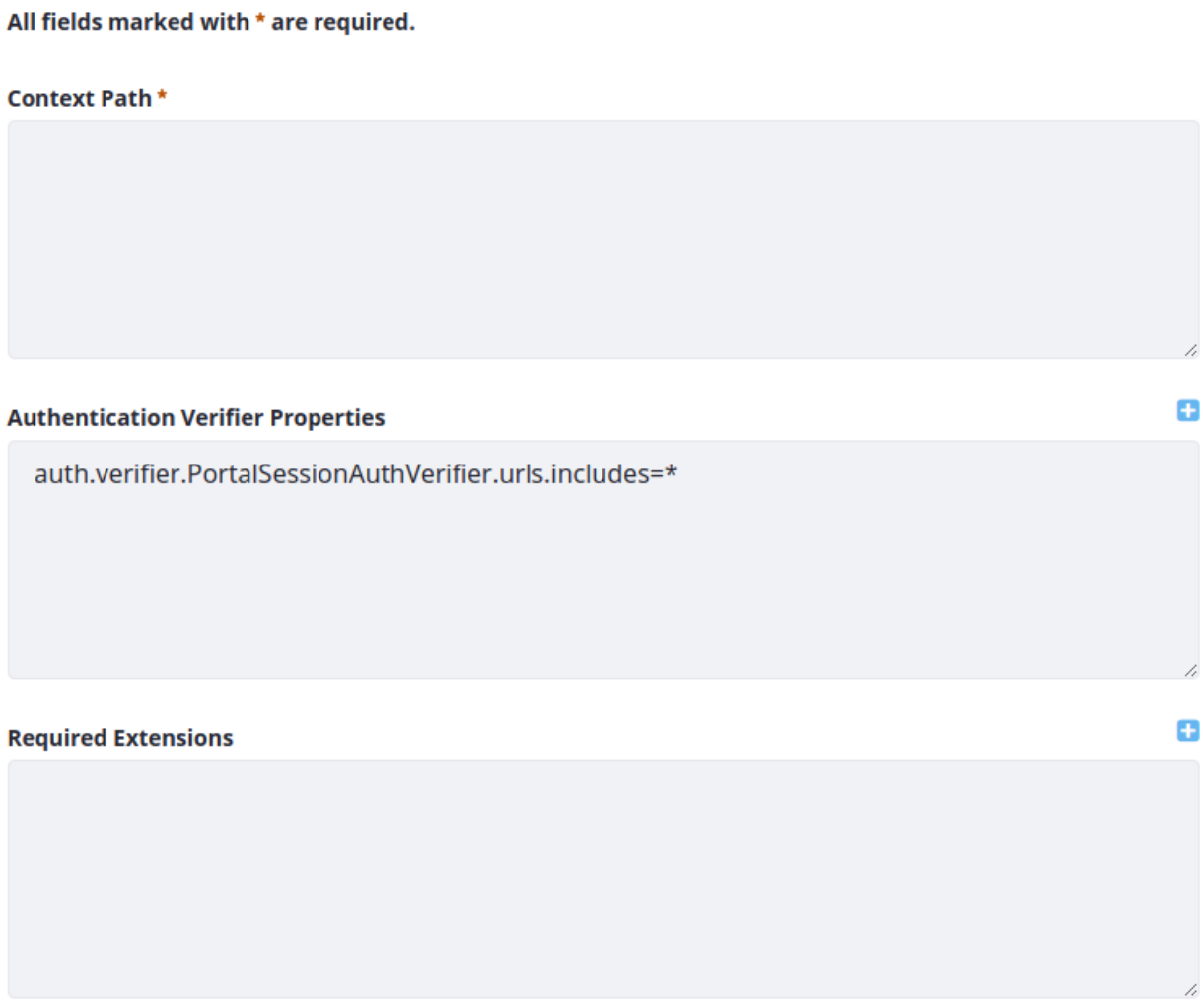


Figure 68.2: Fill out this form to create a CXF endpoint.

```
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 that Liferay-specific Bnd instructions are prefixed with `Liferay` to avoid conflicts.

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
```

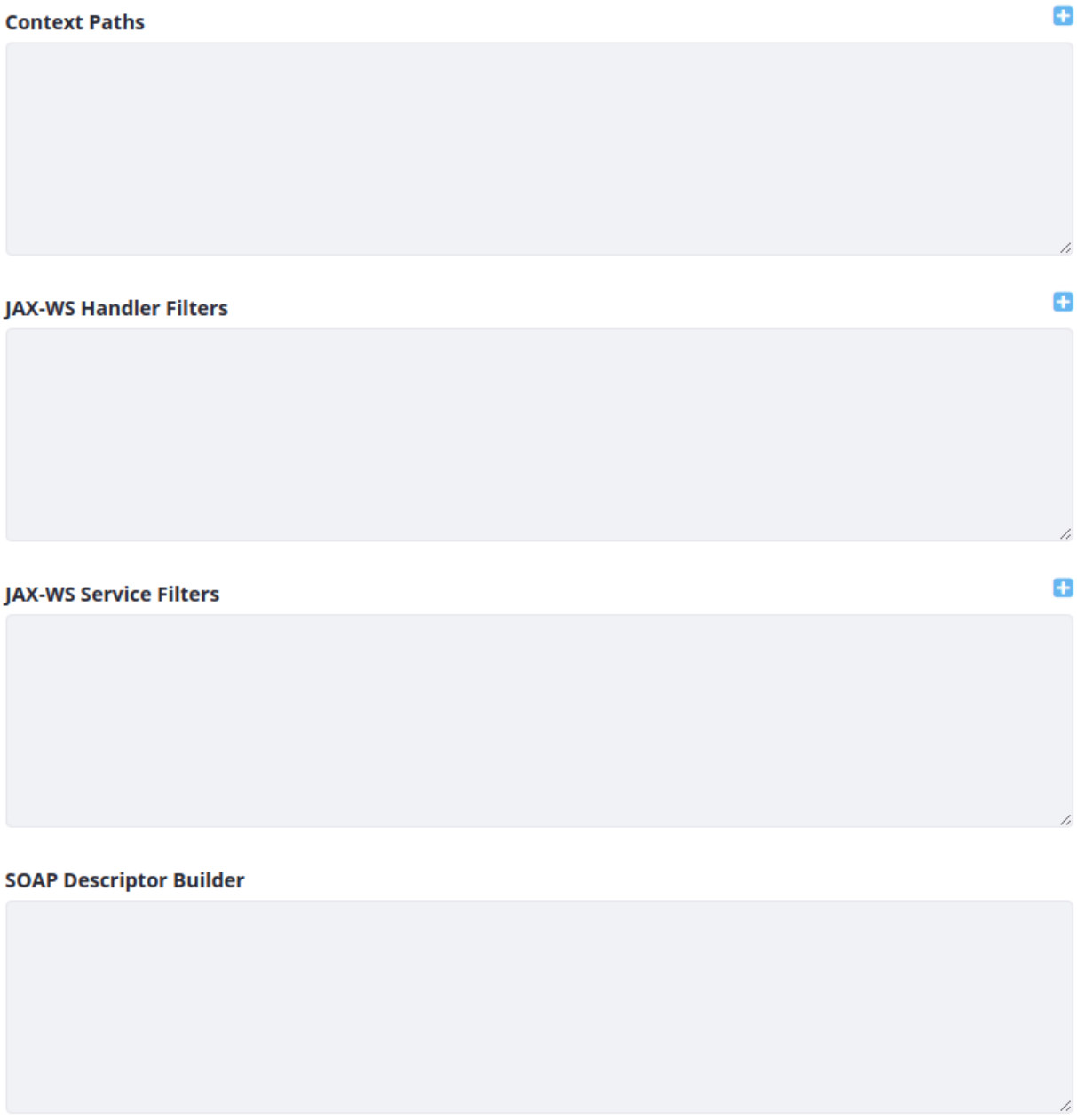


Figure 68.3: Fill out this form to create a SOAP extender.

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();

}
```

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 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.

CHAPTER 69

SEARCH

Liferay stores its information in a database, so why not search the database directly? Why add the complexity of a search engine? First, because database table merges are expensive! Documents in a search index often contain searchable fields from multiple tables in the database. Traversing the date in this way takes too long.

In addition to the performance problem, search engines provide access to additional features, like relevance and scoring. By contrast, databases do not support features like fuzzy searching or relevancy. Moreover, search engines can apply algorithms such as "More Like This" to return similar content. Search engines also support geolocation, faceting of search results, and multi-lingual searching.

This section contains information on extending Liferay's search functionality, enabling your custom entities to be indexed and searched in Liferay DXP, and configuring the developer-friendly embedded Elasticsearch server to suit your needs. First, some basic search concepts.

69.1 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).

69.2 Mapping Definitions

Search engines are semi-intelligent, automatically deciphering how to process documents passed to them. However, there are instances where it's desirable to configure explicitly how to process a field.

Mappings control how a search engine processes a given field. For instance, if a field name ends 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, mappings are defined in `liferay-type-mappings.json` and `schema.xml`, respectively.

The Elasticsearch mapping JSON file can be seen in the Liferay DXP source code, in the `portal-search-elasticsearch6` module:

```
portal-search-elasticsearch6-impl/src/main/resources/META-INF/mappings/liferay-type-mappings.json
```

The Solr `schema.xml` can be seen in the `portal-search-solr7` module's source code:

```
portal-search-solr7-impl/src/main/resources/META-INF/resources/schema.xml
```

Access the Solr 7 module's source code from the `liferay-portal` repository here

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.

69.3 Liferay Search Infrastructure

Search engines already provide native APIs. Why does Liferay provide search infrastructure to wrap the search engine? Liferay's search infrastructure does several things:

1. Index documents with the fields Liferay needs (`entryClassName`, `entryClassPK`, `assetTagNames`, `assetCategories`, `companyId`, `groupId`, staging status, etc.).
2. Apply the right filters to search queries (e.g., for scoping results).
3. Apply permission checking on the results.
4. Summarizing returned results.

That's just a taste of Liferay's Search Infrastructure. Continue reading to learn more.

69.4 Elasticsearch Logging

When you first start Liferay DXP, an embedded Elasticsearch server starts so that search works out of the box. The embedded search engine is not suitable for production, but is useful for testing and development.

In Fix Pack 3 and CE GA2, the Liferay log was slimmed down by removing INFO level Elasticsearch logs for the embedded Elasticsearch. WARN and ERROR logs are still displayed. If you miss

hearing from Elasticsearch during startup, you can enable the INFO log level for the embedded Elasticsearch server.

Note: These instructions show you how to adjust the embedded Elasticsearch server's logs. Logging for Liferay DXP's search functionality is configurable via the Log Levels screen at Control Panel → Configuration → Server Administration → Log Levels tool. Narrow down the list to include only the search classes and packages by searching for *com.liferay.portal.search*.

To adjust logging for a remote Elasticsearch server, see Elastic's documentation.

Here's an example log message that *is* displayed by default:

```
2018-09-13 16:49:24.442 WARN  [Elasticsearch initialization thread][EmbeddedElasticsearchConnection:315]
```

Manage the log levels for the `EmbeddedElasticsearchConnection` in Server Administration.

Here's an example Elasticsearch log message that *isn't* displayed by default:

```
[2018-09-05T17:25:30,107][WARN ][o.e.d.i.m.MapperService  ] [unmapped_type:string] should be replaced with [unmapped_type:keyword]
```

To adjust logging for the `o.e.d.i.m.MapperService` and other Elasticsearch classes,

1. Create a `config` folder in `Liferay Home/data/elasticsearch6/`.
2. Create a `log4j2.properties` file in the new folder.
3. To enable INFO level logging, populate the `log4j2.properties` file with these contents:

   ```
   appender.console.layout.pattern=[%d{ISO8601}][%-5p][%-25c{1.}] %marker%m%n
   appender.console.layout.type=PatternLayout
   appender.console.name=console
   appender.console.type=Console
   logger.action.level=info
   logger.action.name=org.elasticsearch.action
   rootLogger.appenderRef.console.ref=console
   rootLogger.level=info
   status=error
   ```

   Read the Elasticsearch logging documentation for more information.
4. Restart the Liferay DXP server or just the embedded Elasticsearch.
5. To restart just the embedded Elasticsearch server, create a file called

   ```
   com.liferay.portal.bundle.blacklist.internal.BundleBlacklistConfiguration.config
   ```

   in `Liferay Home/osgi/configs` directory and populate it with

   ```
   blacklistBundleSymbolicNames="com.liferay.portal.search.elasticsearch6.impl"
   ```
6. Save the `.config` file and the bundle is stopped. Once you see the `[STOPPED]` message in the logs, restart the bundle by commenting the line out and re-saving the file:

   ```
   #blacklistBundleSymbolicNames="com.liferay.portal.search.elasticsearch6.impl"
   ```

Once this is accomplished, the embedded Elasticsearch server displays Elasticsearch logs at the INFO level and above.

Two common Elasticsearch logs can be configured further: the Slow Log and the JVM's Garbage Collection log.

Configuring Slow Log

Read about Elasticsearch's Slow Log here.

Configure the Slow Log for the embedded Elasticsearch server in the Elasticsearch 6 entry in System Settings. Add these settings to the Additional Index Configurations property:

```
index.indexing.slowlog.threshold.index.debug: 2s
index.indexing.slowlog.threshold.index.info: 5s
index.indexing.slowlog.threshold.index.trace: 500ms
index.indexing.slowlog.threshold.index.warn: 10s

index.search.slowlog.threshold.fetch.debug: 500ms
index.search.slowlog.threshold.fetch.info: 800ms
index.search.slowlog.threshold.fetch.trace: 200ms
index.search.slowlog.threshold.fetch.warn: 1s

index.search.slowlog.threshold.query.debug: 2s
index.search.slowlog.threshold.query.info: 5s
index.search.slowlog.threshold.query.trace: 500ms
index.search.slowlog.threshold.query.warn: 10s
```

These are example values. Adjust as needed.

Configuring JVM Garbage Collection Logging

As with the Slow Log configuration, Elasticsearch's JVM Garbage Collection logging is adjustable in the Elasticsearch 6 entry in System Settings. Add these settings to the Additional Configurations property:

```
monitor.jvm.gc.enabled: true
monitor.jvm.gc.overhead.debug: 40
monitor.jvm.gc.overhead.info: 70
monitor.jvm.gc.overhead.warn: 90
```

These are example values. Adjust as needed.

69.5 Indexing Framework

Unless you're searching for model entities using database queries (not recommended in most cases), each asset in Liferay DXP must be indexed in the search engine. The indexing code is specific to each asset, as the asset's developers know what fields to index and what filters to apply to the search query.

In past versions of Liferay DXP, when your asset required indexing, you would implement a new Indexer by extending `com.liferay.portal.kernel.search.BaseIndexer<T>`. 7.0 introduces a new pattern that relies on composition instead of inheritance. If you want to use the old approach, feel free to extend `BaseIndexer`. It's still supported.

Search and Indexing Overview

Liferay's original Search API was built around the Lucene search and indexing library. To this day, familiarity with Lucene will jump-start your understanding of Liferay's Search API. However, starting with the 7.0 version of Liferay DXP, the Search API is being reworked, so that the parts closely tied to Lucene are becoming more generic. Elasticsearch support was added (in addition to Solr), and most of the newer searching and indexing APIs aim to leverage/map Elasticsearch

APIs. This means that in many cases (for example the `Query` types) there is a one-to-one mapping between the Liferay and Elasticsearch APIs.

In addition to the Elasticsearch centered APIs, Liferay's Search Infrastructure includes additional APIs serving these purposes:

- Indexed documents include the fields needed by Liferay DXP (e.g., `entryClassName`, `entryClassPK`, `assetTagNames`, `assetCategories`, `companyId`, `groupId`, staging status).
- It ensures the scope of returned search results is appropriate by applying the right filters to search requests.
- It provides permission checking and hit summaries to display in the search portlet.

To understand how search and indexing code makes your custom models seamlessly searchable, you must know how to influence each portion of the search and indexing cycle.

Indexing

Model entities store data fields in the database. For example, Guestbooks store a *name* field. During the cycle's Indexing step, you prepare the model entity to be searchable by defining the model's fields that are sent to the search engine, later used during a search.

To control how model entity fields are indexed in search engine documents,

`ModelDocumentContributor` classes specify which database fields are indexed in the model entity's search engine document. This class's `contribute` method is called each time the `add` and `update` methods in the entity's service layer are called.

`ModelIndexerWriterContributor` classes configure the re-indexing and batch re-indexing behavior for the model entity. This class's method is called when a re-index is triggered from the Search administrative application found in Control Panel → Configuration → Search, or when a CRUD operation is made on the entity, *if* the `modelIndexed` method is implemented in your contributor.

`DocumentContributor` classes (without any `indexer.class.name` component property or type parameter) contribute certain fields to every index document, regardless of what base entity is being indexed. For example, the `GroupedModelDocumentContributor` contributes the `GROUP_ID` and `SCOPE_GROUP_ID` fields for all documents with a backing entity that's also a `GroupedModel`.

Searching

Searches start with a user entering keywords into a search bar. The entered keywords are processed by the back-end search infrastructure, transformed into a *query* the search engine can understand, and used to match against each search document's fields.

To control the way model entity documents are searched,

`KeywordQueryContributor` classes contribute clauses to the ongoing search query. This is called at search time and ensures that all the fields you indexed are also the ones searched. For example, if you index fields with the Site locale appended to them (`title_en_us`, for example), you want the search query to include the same locale when the document is searched. If the search query contains another locale (like `title_es_ES`) or searches the plain field (`title`), inaccurate results are returned.

To contribute query clauses to every search, regardless of what base entity is being searched, implement a `KeywordQueryContributor` class registered without an `indexer.class.name` component property. For example, see `AlwaysPresentFieldsKeywordQueryContributor`.

`ModelPreFilterContributors` control how search results are filtered before they're returned from the search engine. For example, adding the workflow status to the query ensures that an entity in the trash isn't returned in the search results.

To contribute Filter clauses to every search, regardless of what base entity is being searched, implement a `QueryPreFilterContributor`. `QueryPreFilterContributor` is constructed only once under the root filter during a search. For example, see `AssetCategoryTitlesKeywordQueryContributor`.

What's the difference between `QueryPreFilterContributor` and `ModelPreFilterContributor`? `QueryPreFilterContributor` is constructed only once under the root filter during a search, while `ModelPreFilterContributor` is constructed once per model entity, and added under each specific entity's subfilter.

Returning Results

When a model entity's indexed search document is obtained during a search request, it's converted into a summary of the model entity.

To control the result summaries for model entity documents,

`ModelSummaryContributor` classes get the `Summary` object created for each search document, so you can manipulate it by adding specific fields or setting the length of the displayed content.

`ModelVisibilityContributor` classes control the visibility of model entities that can be attached to other asset types (for example, File Entries can be attached to Wiki Pages), in the search context.

One important step must occur to make sure the above classes are discovered by the search framework.

Search Service Registration

To register model entities with Liferay's search framework,

`SearchRegistrars` use the search framework's registry to define certain things about your model entity's `ModelSearchDefinition`: which fields are used by default to retrieve documents from the search engine, and which optional search services are registered for your entity. Registration occurs as soon as the Component is activated (during portal startup or deployment of the bundle).

Mapping the Composite Search and Indexing Framework to `Indexer/BaseIndexer` Code

If you're used to the old way of indexing custom entities (extending `BaseIndexer`, the abstract implementation of `Indexer`), the table below provides a quick overview about how the methods of the `Indexer` interface were decomposed into several new classes and methods.

Indexer/BaseIndexer method	Composite Indexer Equivalent	Example
Class Constructor	`SearchRegistrar`	`GuestbookSearchRegistrar`
`setDefaultSelectedFieldNames`	`SearchRegistrar.activate`	`GuestbookSearchRegistrar`
`setDefaultSelectedLocalizedFieldNames`	`SearchRegistrar.activate`	`GuestbookSearchRegistrar`
`setPermissionAware`	`ModelResourcePermissionRegistrar`	`GuestbookModelResourcePermissionReg`
`setFilterSearch`	`ModelResourcePermissionRegistrar`	`GuestbookModelResourcePermissionReg`
`getDocument/doGetDocument`	`ModelDocumentContributor`	`GuestbookModelDocumentContributor`
`reindex/doReindex`	`ModelIndexerWriterContributor`	`GuestbookModelIndexerWriterContribu`
`addRelatedEntryFields`	`RelatedEntryIndexer`	`DLFileEntryRelatedEntryIndexer`
`postProcessContextBooleanFilter/PostProcessContextQuery`	`ModelPreFilterContributor`	`DLFileEntryModelPreFilterContributo`
`postProcessSearchQuery`	`KeywordQueryContributor`	`GuestbookKeywordQueryContributor`

Indexer/BaseIndexer method	Composite Indexer Equivalent	Example
`getFullQuery`	`SearchContextContributor`	`DLFileEntryModelSearchContextContri`
`isVisible/isVisibleRelatedEntry`	`ModelVisibilityContributor`	`DLFileEntryModelVisibilityContribut`
`getSummary/createSummary/doGetSummary`	`ModelSummaryContributor`	`GuestbookModelSummaryContributor`
`Indexer.search/searchCount`	No change	Guestbook `view_search.jsp`
`Indexer.delete/doDelete`	No change	`MBMessageLocalServiceImpl.deleteDis`

In addition, you can index `ExpandoBridge` attributes. This was previously accomplished in `BaseIndexer`'s `getBaseModelDocument`. Now you implement an `ExpandoBridgeRetriever`. See `DLFileEntryExpandoBridgeRetriever` for an example implementation.

Permissions Aware Searching and Indexing

In previous versions of Liferay DXP, search was only *permissions aware* (indexed with the entity's permissions and searched with those permissions intact) if the application developer specified this line in the `Indexer` class's constructor:

```
setPermissionAware(true);
```

Now, search is permissions aware by default *if the new permissions approach,* as described in these tutorials, is implemented for an application.

Annotating Service Methods to Trigger Indexing

Having entities translated into database entities *and* search engine documents means that there's a possibility for a state mismatch between the database and search engine. For example, when a Blogs entry is added, updated or removed from the database, corresponding changes must be made in the search engine. To do this, intervention must be made into the service layer. For Service Builder entities, this occurs in the `LocalServiceImpl` classes. There's an annotation that simplifies this: `@Indexable`. It takes a `type` property that can have two values: `REINDEX` or `DELETE`. Commonly, a `deleteEntity` method in the service layer is annotated like this:

```
@Indexable(type = IndexableType.DELETE)
@Override
@SystemEvent(type = SystemEventConstants.TYPE_DELETE)
public BlogsEntry deleteEntry(BlogsEntry entry) throws PortalException {
    ...
}
```

The `@Indexable` annotation is executed by Liferay's Spring infrastructure, so if you have a method with that annotation, you must call it using a service instance variable with the spring wrapped logic. The reference is available by default in Service Builder generated `*LocalServiceImpl` classes because of this declaration in the parent `*LocalServiceBaseImpl`:

```
@BeanReference(type = BlogsEntryLocalService.class)
    protected BlogsEntryLocalService blogsEntryLocalService;
```

This means that in the `*LocalServiceImpl`, you must not call

```
this.deleteEntry(...)
```

The annotation won't be executed and you'll be left with a state mismatch between the search engine document and the database column. Instead follow the pattern in Liferay DXP's code, using the service instance variable to call service methods:

```
blogsEntryLocalService.deleteEntry(entry);
```

For step-by-step instructions on indexing model entities, visit the Search and Indexing section of the Developing a Web Application tutorials.

CHAPTER 70

ASSET FRAMEWORK

The asset framework powers the core Liferay features so you can add them to your application. For example, if you build 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.

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.

The tutorials that follow in this section explore how to leverage 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.

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!

70.1 Related Topics

What is Service Builder

Service Builder Persistence
Configurable Applications

70.2 Adding, Updating, and Deleting Assets

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 leverage assets, you must also implement indexers for your portlet's entities. Liferay's asset framework uses indexers to manage assets.

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<WikiPage> 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. That's all there is to it.

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 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, 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(), assetEntry.getEntityId());

Indexer<ENTITY> indexer = IndexerRegistryUtil.nullSafeGetIndexer(ENTITY.class);
indexer.delete(assetEntry);
```

In your -LocalServiceImpl class, you can write similar code. Replace the *ENTITY* class name and variable with your entity's name.

Important: For Liferay's Asset Publisher application to show your entity, the entity must have an Asset Renderer.

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?

70.3 Implementing Asset Categorization and Tagging

In this tutorial, you'll enable tags and categories entities in the UI through a set of JSP tags. Before beginning, your entities should be asset-enabled and you should have asset renderers enabled for them.

Now it's time to get started!

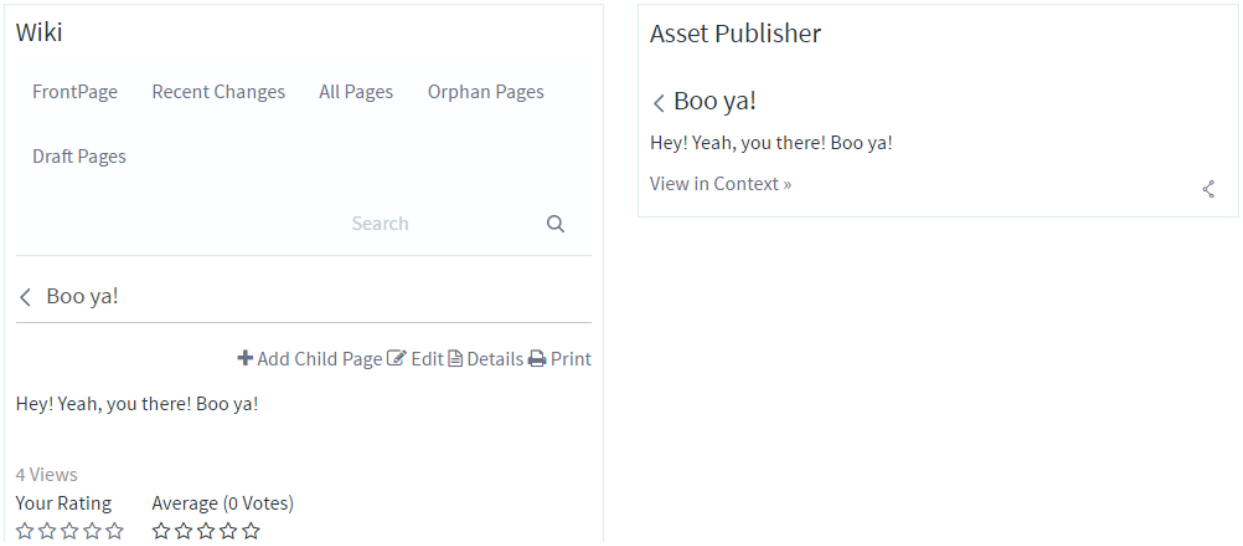


Figure 70.1: It can be useful to show custom entities, like this wiki page entity, in a JSP or in an Asset Publisher.

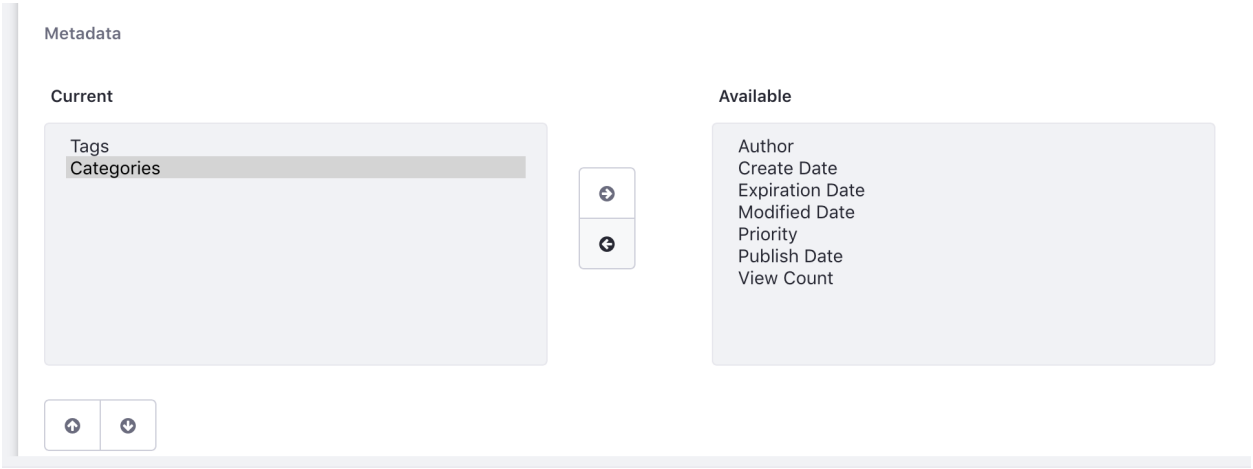


Figure 70.2: Adding category and tag input options lets authors aggregate and label custom entities.

Adding Tags and Categories

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:

```
<liferay-ui:asset-categories-error />
<liferay-ui:asset-tags-error />
...
<aui:fieldset-group markupView="lexicon">
    ...
    <aui:fieldset collapsed="<%= true %>" collapsible="<%= true %>" label="categorization">
        <liferay-asset:asset-categories-selector name="categories" type="assetCategories" />

        <liferay-asset:asset-tags-selector name="tags" type="assetTags" />
    </aui:fieldset>
    ...
</aui:fieldset-group>
```

The `liferay-asset:asset-categories-selector` and `liferay-asset:asset-tags-selector` tags generate form controls that let users browse/select 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` markup view.

Displaying Tags and Categories

Tags and categories should be displayed 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>
```

The `portletURL` parameter is used for both tags and categories. Each tag that uses this parameter becomes a link containing the `portletURL` *and* `tag` or `categoryId` parameter value. To implement this, you must 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
What is Service Builder?

70.4 Relating Assets

Relating assets connects individual pieces of content across your site or portal. This helps 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.

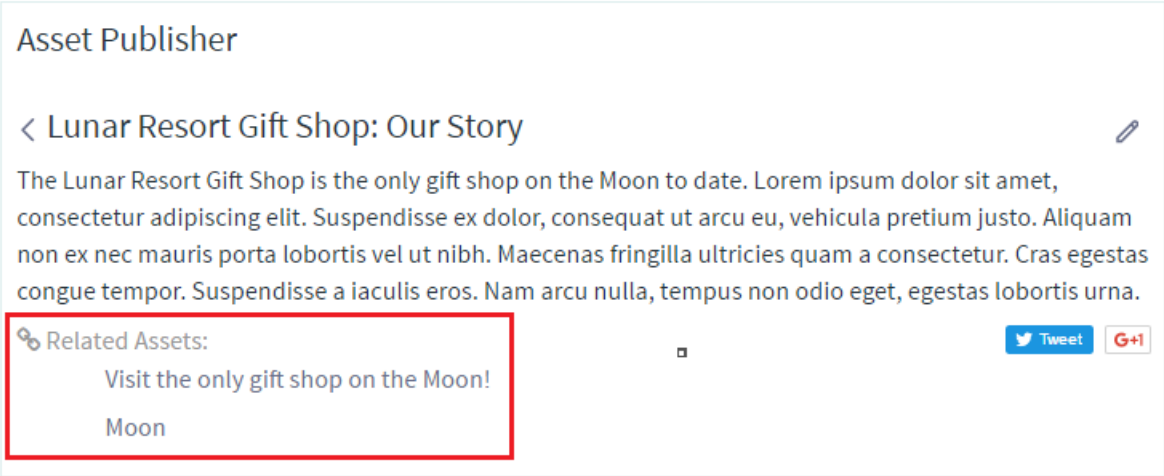


Figure 70.3: You and your users can find it helpful to relate assets to entities, such as this blogs entry.

Now you'll learn how to provide a way for authors to relate content. This tutorial assumes that you've Adding, Updating, and Deleting Assets your application. 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.

1. In your portlet's `service.xml`, put the following line of code below any finder method elements and then run Service Builder:

```
<reference package-path="com.liferay.portlet.asset" entity="AssetLink" />
```

2. 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 must pass in the current user's ID, the asset entry's ID, the asset link entries' IDs, and the link type. Invoke this method after creating the asset entry. If you assign to an `AssetEntry` variable (e.g., one called `assetEntry`) the value returned from invoking `assetEntryLocalService.updateEntry`, 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`.

3. 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

The UI for linking assets should be in the JSP where users 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 a collapsible panel. This code is placed inside the `aui:fieldset` tags of the JSP.

1. Add the `liferay-ui:input-asset-links` tag to your form. Here's how it's added in the Blogs application:

   ```
   <aui:fieldset collapsed="<%= true %>" collapsible="<%= true %>" label="related-assets">
       <liferay-ui:input-asset-links
           className="<%= BlogsEntry.class.getName() %>"
           classPK="<%= entryId %>"
       />
   ```

   The following screenshot shows the Related Assets menu for an application. Note that it is contained in a collapsible panel titled Related Assets.

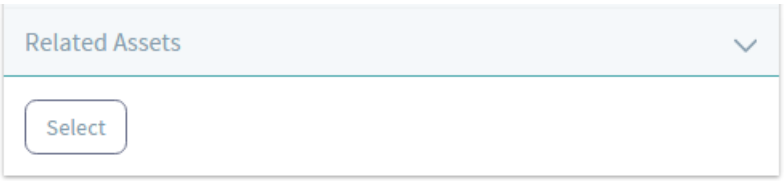


Figure 70.4: Your portlet's entity is now available in the Related Assets *Select* menu.

2. Unfortunately, the Related Assets menu shows your entity's fully qualified class name. To replace it with a simplified name for your entity, add a language key with the fully qualified class name for the key and the name you want for the 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.

1. You must get the `AssetEntry` object associated with your entity:

```
<%
long insultId = ParamUtil.getLong(renderRequest, "insultId");
Insult ins = InsultLocalServiceUtil.getInsult(insultId);
AssetEntry assetEntry = AssetEntryLocalServiceUtil.getEntry(Insult.class.getName(), ins.getInsultId());
%>
```

2. Use the `liferay-ui:asset-links` tag to show the entity's related assets. 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()%>" />
```

Great! Now you have the JSP that lets your users view related assets. 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.

Related Topics

Adding, Updating, and Deleting Assets
    What is Service Builder?
    Defining Content Relationships

70.5 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.

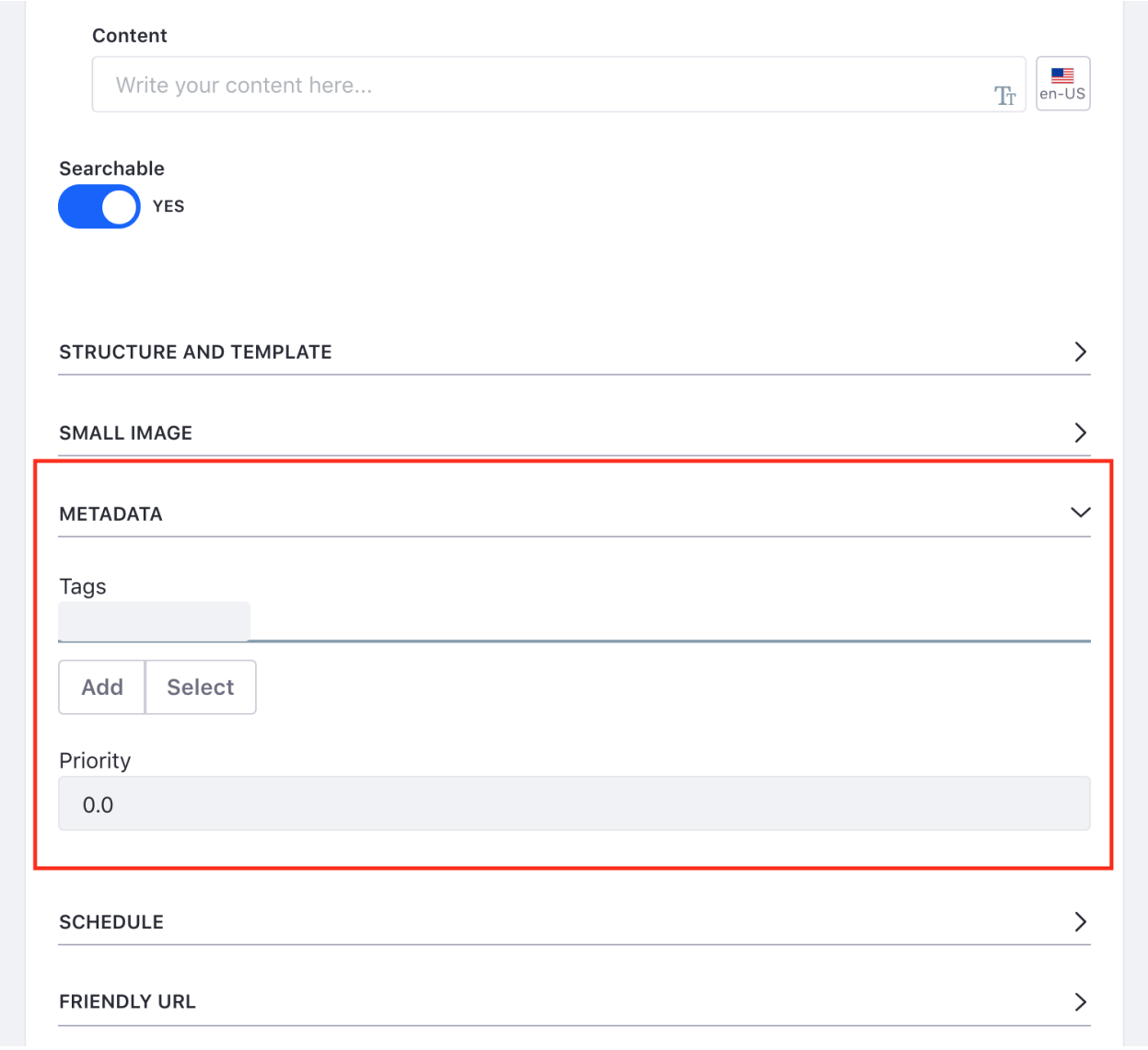


Figure 70.5: The Priority field lets users set an asset's priority.

This field isn't enabled when you create an asset. 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 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
- Implementing Asset Categorization and Tagging
- Relating Assets
- Rendering an Asset
- Publishing Assets

CHAPTER 71

RENDERING AN ASSET

Before you create a way to render your asset, make sure it's added to the asset framework by following the [Adding, Updating, and Deleting Assets tutorial. Once you add your asset to the framework, you can render the asset using the Asset Publisher application. The default render, however, only displays the asset's title and description text. Anything else requires additional coding. For instance, you might want these additional things:

- An edit feature for modifying an asset.
- View an asset in its original context (e.g., a blog in the Blogs application; a post in the Message Boards application).
- Embed images, videos, and audio.
- Restrict access to users who do not have permissions to interact with the asset.
- Allow users to comment on the asset.

You can dictate your asset's rendering capabilities by providing the *Asset Renderer* framework. There are two things you must do to get your asset renderer functioning properly for your asset:

1. Create an asset renderer for your custom asset.
2. Create an asset renderer factory to create an instance of the asset renderer for each asset entity.

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, and more. 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!

71.1 Prerequisites for Asset Enabling and Application

To asset-enable your application, you need two things:

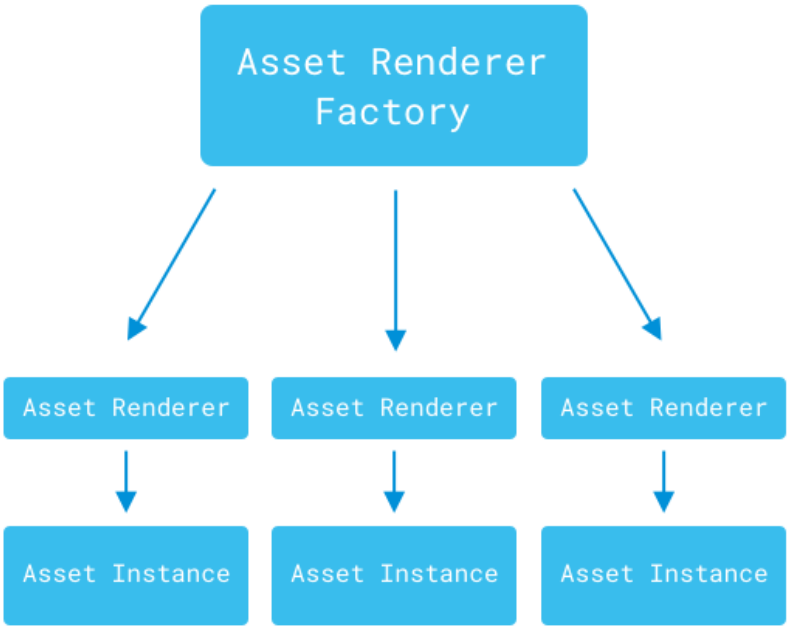


Figure 71.1: The asset renderer factory creates an asset renderer for each asset instance.

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 both enhances the usability of the application, and meets the requirement for adding a non-instanceable portlet to the application. See our tutorial on Adding Custom Panel Apps to learn how to add one.

Now you're ready to create an Asset Renderer.

71.2 Creating an Asset Renderer

In this tutorial, you'll learn how to create an `Asset Renderer` and associate your JSP templates with it, 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 applications, so they can 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();
}
```

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. 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, so the asset publisher can display it. The title and summary can be anything.

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;
}
```

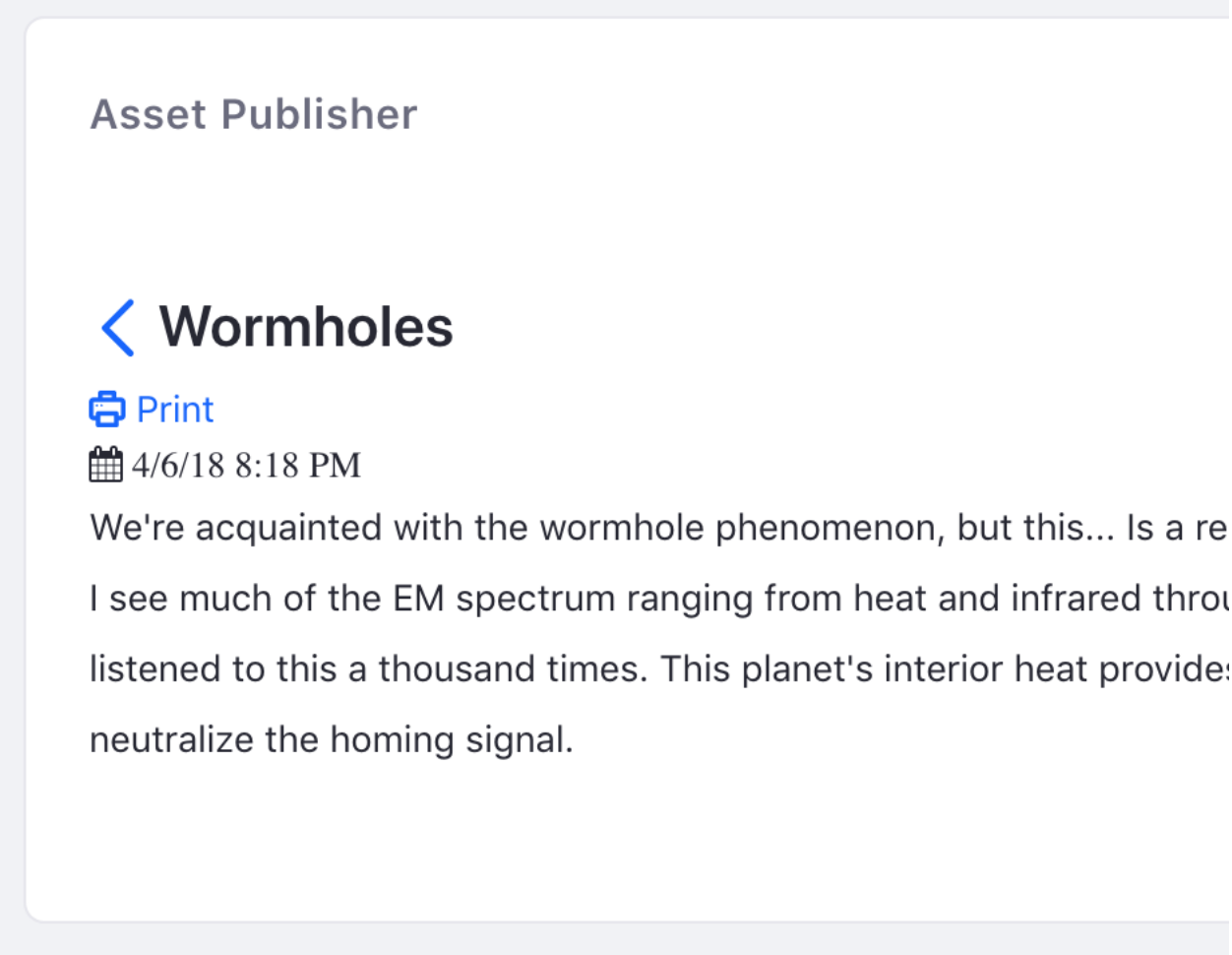


Figure 71.2: Enable printing in the Asset Publisher to display the Print icon for your asset.

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.

11. If your asset is protected by permissions, you can set permissions for the asset via the asset renderer. 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 use the `getUserId()` and `getUserName()` to retrieve the entry's user ID and username, respectively. Then there are three boolean permission methods that 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
- check permissions for the asset

Now you need to create the templates to render the HTML. 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.

71.3 Configuring JSP Templates for an Asset Renderer

An asset can be displayed in several different ways in the Asset Publisher. 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.

1. Blogs assets provide an Edit button that lets you edit the asset. Provide this by adding the following method to the `BlogsEntryAssetRenderer` class:

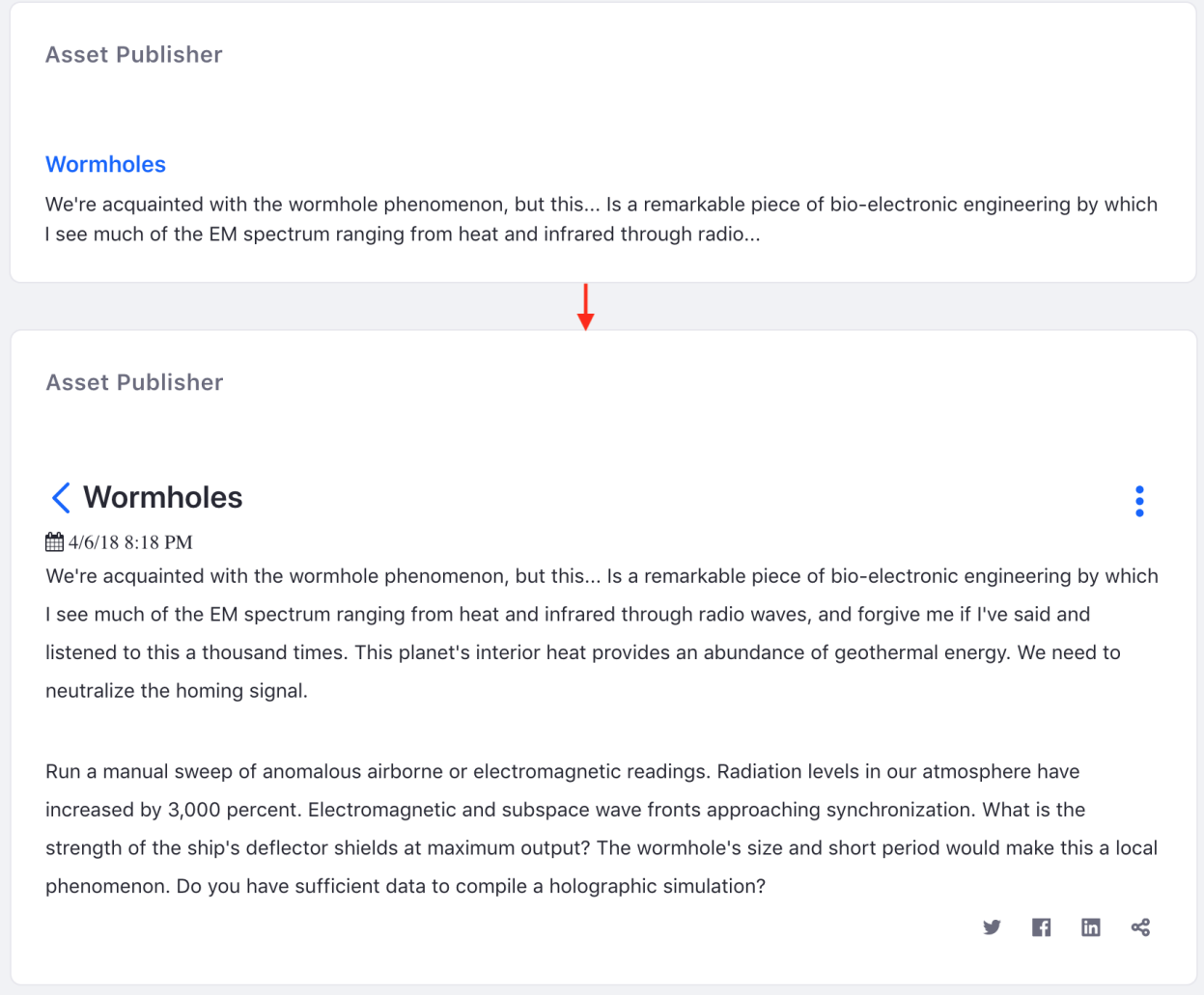


Figure 71.3: The abstract and full content views are rendererd differently for blogs.

```
@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 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.

71.4 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.

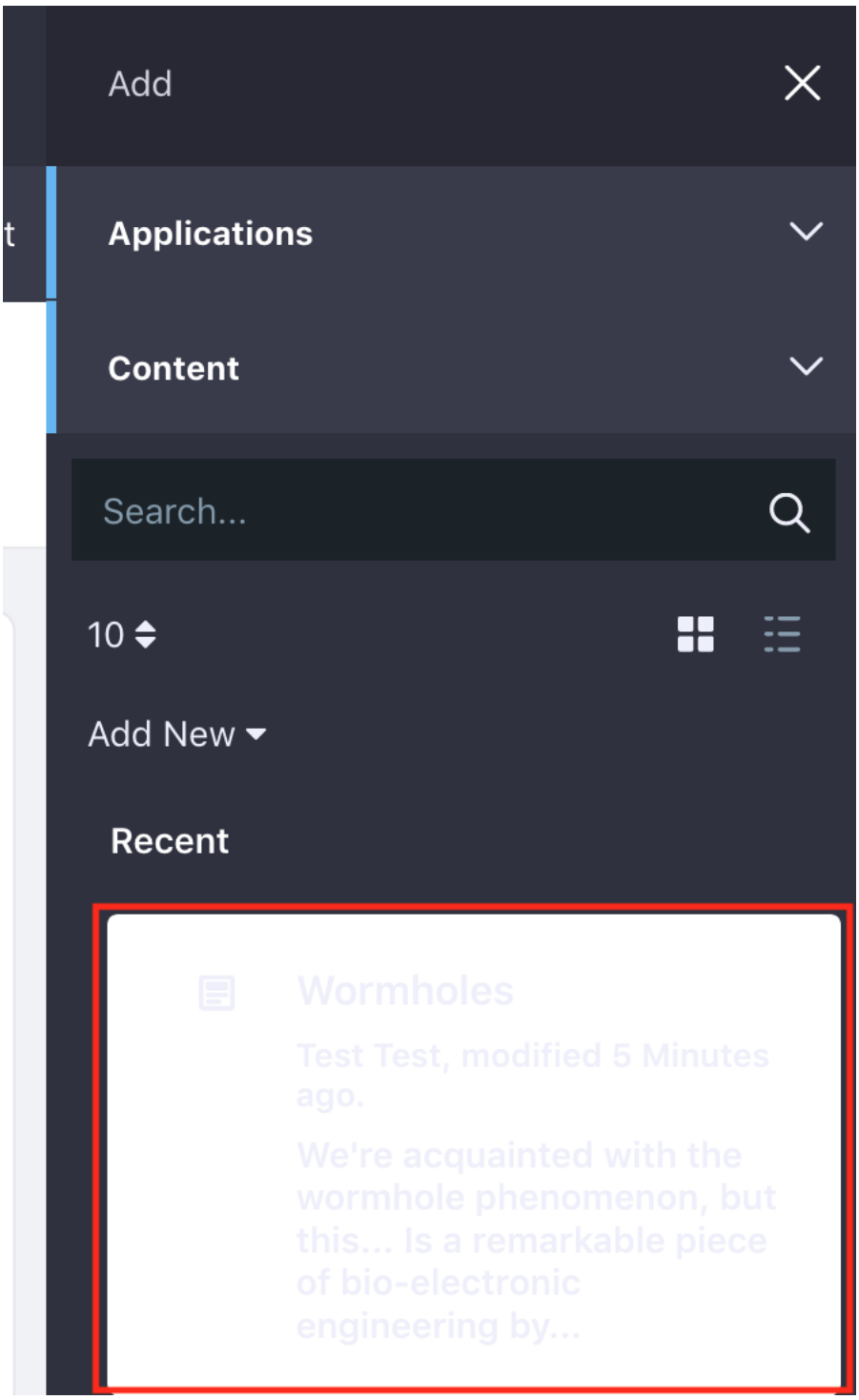


Figure 71.4: The `preview` template displays a preview of the asset in the Content section of the Add menu.

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 Local 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 72

THEMES AND LAYOUT TEMPLATES

A Theme provides the overall look and feel for a site. Understanding the page layout is crucial to targeting the correct markup for styling, organizing your content, and creating your site. Once you understand how the page is organized, you can develop your theme.

If you want to design a website, you must have three key components: CSS, JavaScript, and HTML. Liferay DXP provides CSS extensions and patterns out-of-the-box and supports SASS, as well as multiple JavaScript frameworks. The HTML, however, is rendered via FreeMarker theme templates.

Liferay DXP provides several default FreeMarker templates that each handle a key piece of functionality for the page. To help make the development process easier, Liferay DXP also provides several theme template utilities that you can use in your theme templates to include portlets, use taglibs, access theme objects, and more.

There are several mechanisms for customizing and extending themes, from color schemes to reusable pieces of code. Likewise, there are several mechanisms for customizing and extending portlets.

In this section of tutorials, you'll learn how to develop themes and layout templates, customize portlets, and more.

CHAPTER 73

THEMES

Themes let you customize the default look and feel of your site. Liferay DXP provides several mechanisms for customizing, developing, and extending themes. See the Theme Components and Workflow reference guide for a top-level overview of how themes work in Liferay DXP.

This section of tutorials shows how to create and develop themes for Liferay DXP.

CHAPTER 74

CREATING THEMES

The Liferay Theme Generator lets you create themes, themelets, layout templates and more. It is just one of Liferay JS Theme Toolkit's tools. There are a few dependencies required to run the generator. If you have NodeJS installed, you're already one step ahead.

Follow these steps to install the Liferay Theme Generator and generate a theme:

1. Install Node.js. We recommend installing v8.10.0, which is the version Liferay Portal 7.1 supports. Note that Node Package Manager (npm) is installed with this as well. You'll use npm to install the remaining dependencies and generator. Make sure to set up your npm environment before moving to the next step. Failing to do this can lead to permissions issues later on.

2. Use npm to install Yeoman and gulp:

```
npm install -g yo
```

```
**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.
```

3. Install the Liferay Theme Generator. A few versions of the Liferay Theme Generator are available. The version you must install depends on the version of Liferay DXP you're developing on. The required versions are listed in the table below:

```
Liferay Version | Liferay Theme Generator Version | Command |
--- | --- | --- |
6.2 | 7.x.x | `npm install -g generator-liferay-theme@^7.x.x` |
7.0 | 7.x.x or 8.x.x | Same as above or below |
7.1 | 8.x.x | `npm install -g generator-liferay-theme@^8.x.x` |
```

```
If you're on Windows, follow the instructions in step 4 to install Sass,
otherwise you can skip to step 5.
```

4. The generator uses node-sass. If you are on Windows, you must also install node-gyp and Python.
5. Run the generator and follow the prompts to create your theme:

    ```
    yo liferay-theme
    ```

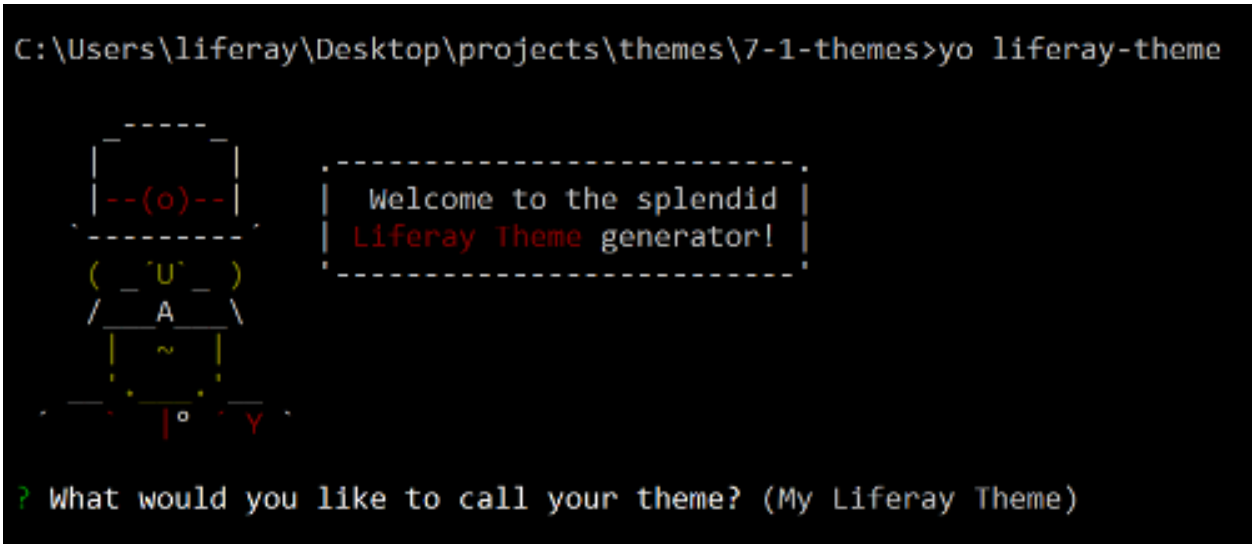


Figure 74.1: You can generate a theme by answering just a few configuration questions.

6. Navigate to your theme folder and run `gulp deploy` to deploy your new theme to the server.

Now you have a powerful theme development tool at your disposal. The sky is the limit!

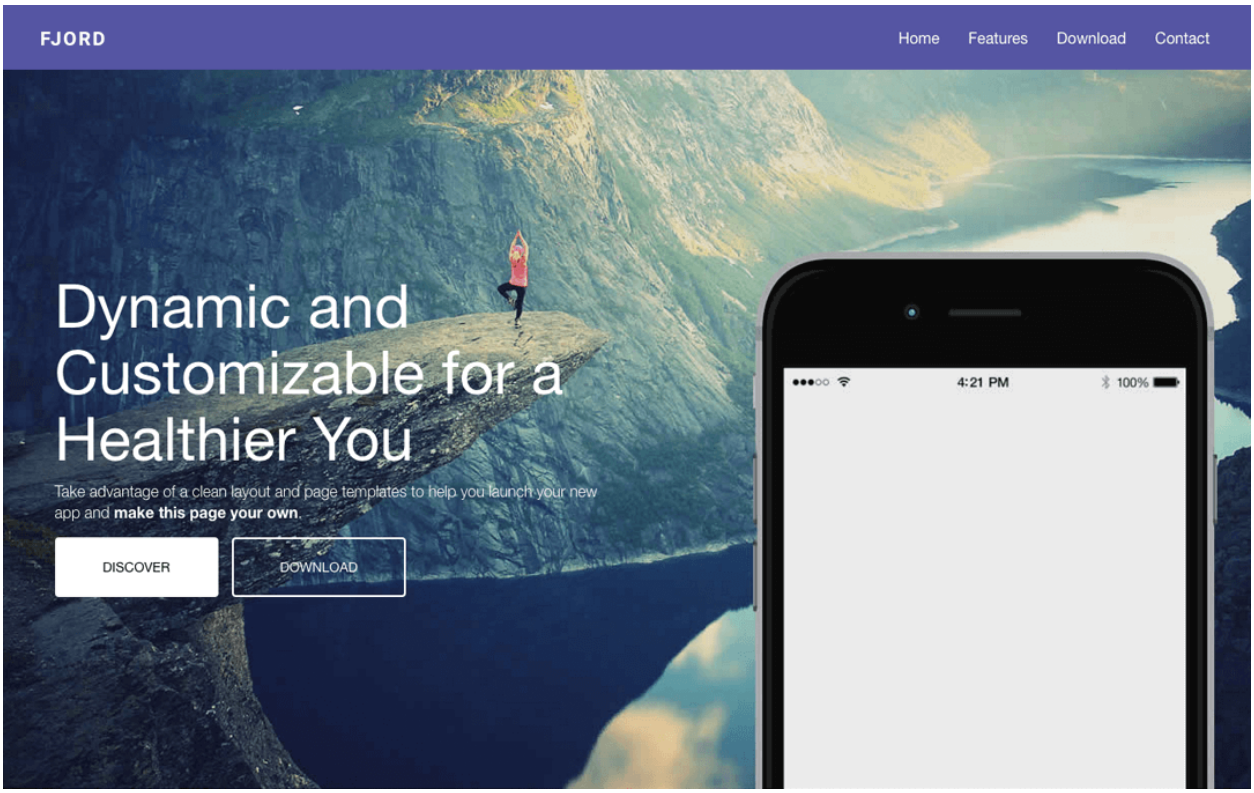


Figure 74.2: The tools are in your hands to create any theme you can imagine.

CHAPTER 75

DEVELOPING THEMES

Theme projects created using the Liferay Theme Generator have access to several gulp tasks you can execute to manage and develop your theme. This section of tutorials covers the available actions that these tasks provide, as well as other information you may find useful while developing your theme.

75.1 Using Developer Mode with Themes

Do you want to develop themes without having to redeploy to see your modifications? Use 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`).

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 Dev Studio DXP.

Setting Developer Mode for Your Server in Dev Studio

To enable Developer Mode for your server in Dev Studio DXP, 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.

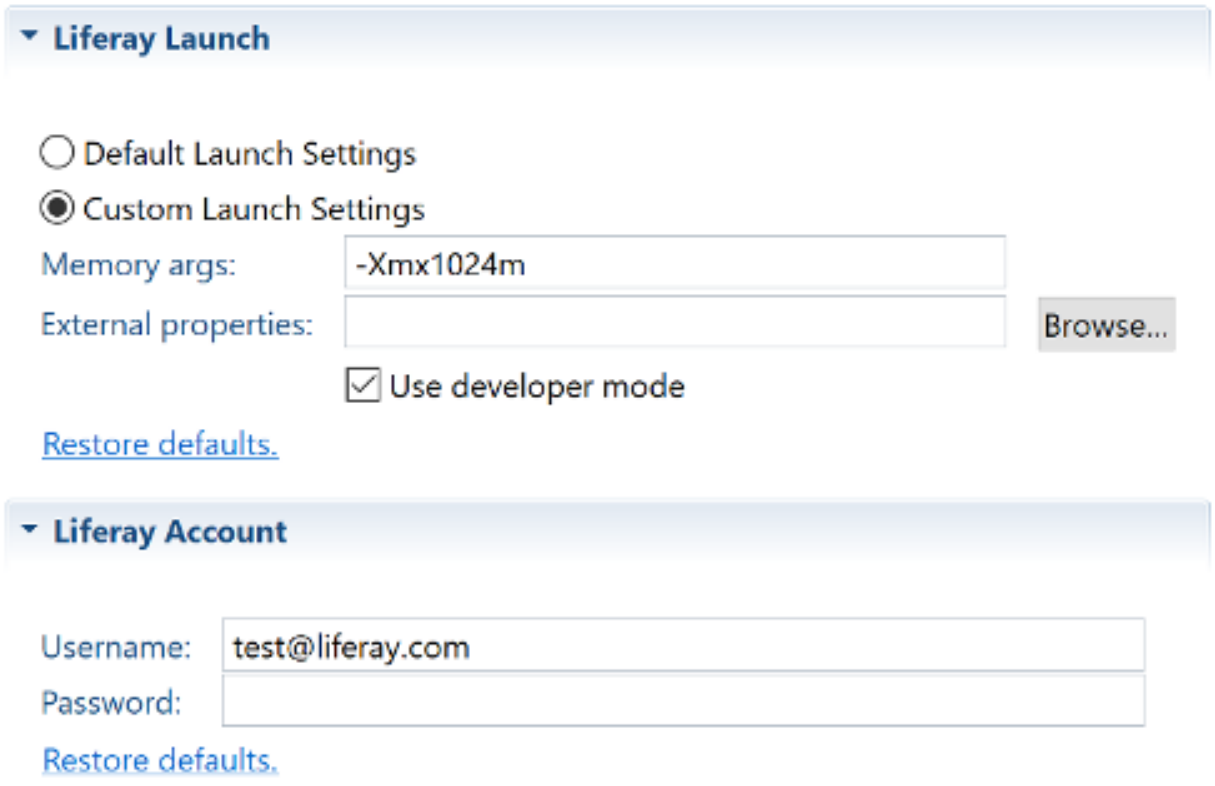


Figure 75.1: The *Use developer mode* option lets you enable Developer Mode for your server in Dev Studio DXP.

Warning: Only change the Server settings from the runtime environment's Liferay Launch section.

When starting your 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 Dev Studio DXP, 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.

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. Select *Template Engines* under the *PLATFORM* heading.
3. By default, the *Resource modification check* (the time in milliseconds that the template is cached) is set to `60000`. 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`). Preference is given 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 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 Dev Studio DXP

75.2 Building Your Theme's Files

The `gulp build` task generates the base theme files, compiles Sass into CSS, and zips all theme files into a WAR file that you can deploy to your server.

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.

Follow these steps to build your theme's files:

1. Navigate to your theme's root folder and run `gulp build`.

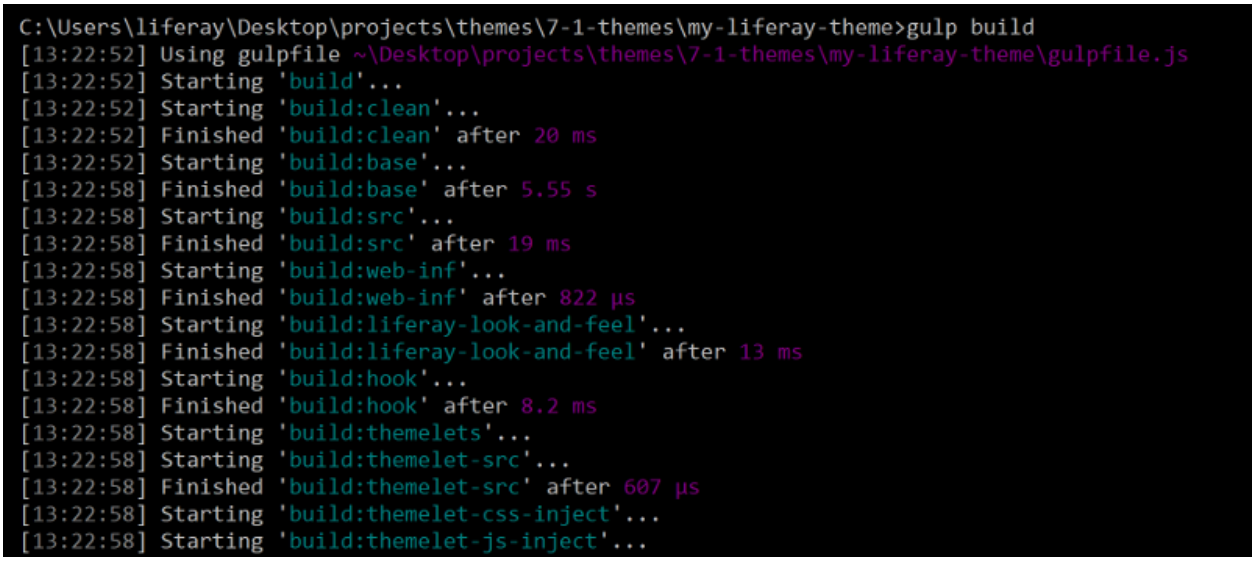


Figure 75.2: Run the `gulp build` task to build your theme's files.

2. A new `build` folder is created with all your theme's files. You can copy these files and folders to your theme's `src` folder to modify the theme.

3. Your theme's files are zipped into a `war` file in a new `dist` folder. Deploy the `war` file to your app server to make it available.

Related Topics

Automatically Deploying Theme Changes
Copying an Existing Theme's Files
Deploying Themes

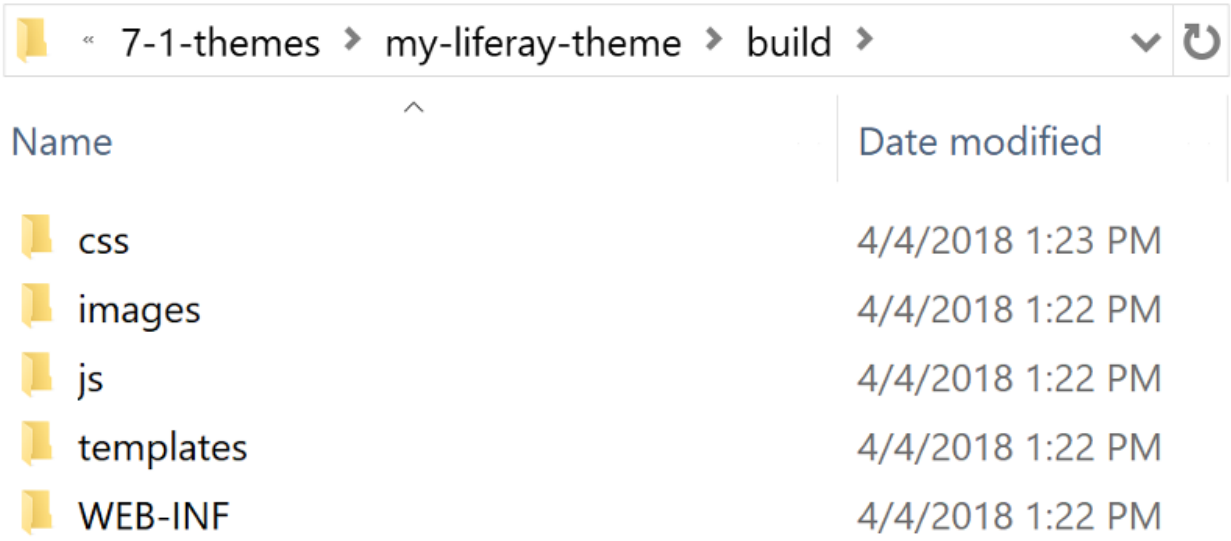


Figure 75.3: The build folder contains all your theme's files.

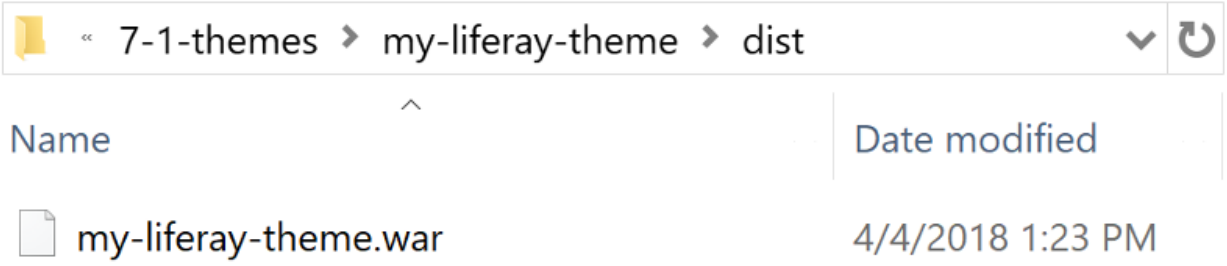


Figure 75.4: The dist folder contains your theme's WAR file.

75.3 Deploying Your Theme

To deploy your theme to your app server, run the `gulp deploy` task. The `gulp deploy` task builds your theme's files, and deploys the generated WAR file to the app server you configured when you created the theme.

Note: If you're running the Felix Gogo shell, you can also deploy your theme using the `gulp deploy:gogo` command.

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.

Follow these steps to deploy your theme:

1. Navigate to your theme's root folder and run `gulp deploy`.

```
C:\Users\liferay\Desktop\projects\themes\7-1-themes\my-liferay-theme>gulp deploy
[14:37:41] Using gulpfile ~\Desktop\projects\themes\7-1-themes\my-liferay-theme\gulpfile.js
[14:37:41] Starting 'deploy'...
[14:37:41] Starting 'build'...
[14:37:41] Starting 'build:clean'...
[14:37:42] Finished 'build:clean' after 1.54 s
[14:37:42] Starting 'build:base'...
[14:37:48] Finished 'build:base' after 5.92 s
[14:37:48] Starting 'build:src'...
[14:37:48] Finished 'build:src' after 12 ms
[14:37:48] Starting 'build:web-inf'...
[14:37:48] Finished 'build:web-inf' after 816 µs
[14:37:48] Starting 'build:liferay-look-and-feel'...
[14:37:48] Finished 'build:liferay-look-and-feel' after 10 ms
[14:37:48] Starting 'build:hook'...
[14:37:48] Finished 'build:hook' after 5.76 ms
[14:37:48] Starting 'build:themelets'...
[14:37:48] Starting 'build:themelet-src'...
[14:37:48] Finished 'build:themelet-src' after 543 µs
[14:37:48] Starting 'build:themelet-css-inject'...
[14:37:48] Starting 'build:themelet-js-inject'...
```

Figure 75.5: Run the `gulp deploy` task to build your theme's files and deploy it to your app server.

2. Your server's log displays that the OSGi bundle is started.
3. You can apply your theme through the *Navigation → Site Pages* menu in the Control Menu. Select the *Configure* option for your site pages, and click the *Change Current Theme* button to apply your theme.

Related Topics

Automatically Deploying Theme Changes
Copying an Existing Theme's Files
Creating Reusable Pieces of Code for Your Themes

```
2018-04-04 18:38:30.162 INFO  [Refresh Thread: Equinox Container: 5061071b-1438-0018-18fd-d13
3ebfa3ba3][ThemeHotDeployListener:93] Registering themes for my-liferay-theme
2018-04-04 18:38:33.214 INFO  [Refresh Thread: Equinox Container: 5061071b-1438-0018-18fd-d13
3ebfa3ba3][ThemeHotDeployListener:108] 1 theme for my-liferay-theme is available for use
2018-04-04 18:38:33.408 INFO  [Refresh Thread: Equinox Container: 5061071b-1438-0018-18fd-d13
3ebfa3ba3][BundleStartStopLogger:35] STARTED my-liferay-theme_1.0.0 [759]
```

Figure 75.6: Your server's log notifies you when the theme's bundle has started.

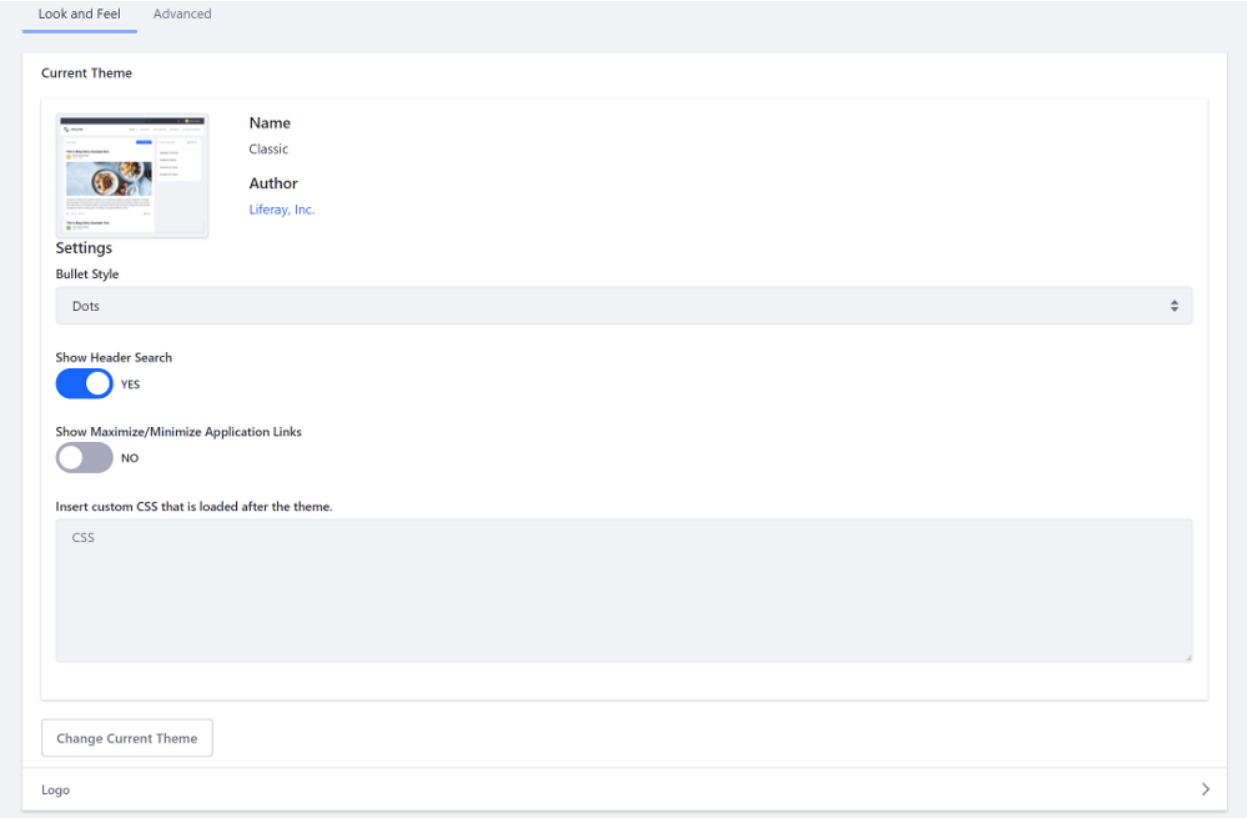


Figure 75.7: Run the `gulp deploy` task to build your theme's files and deploy it to your app server.

75.4 Changing Your Base Theme

Once your theme is built, you can use the `gulp extend` task to change your theme's base 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.

Follow these steps to change your base theme:

1. Navigate to your theme's root folder and run `gulp extend` and choose option 1 to change the base theme your theme extends.

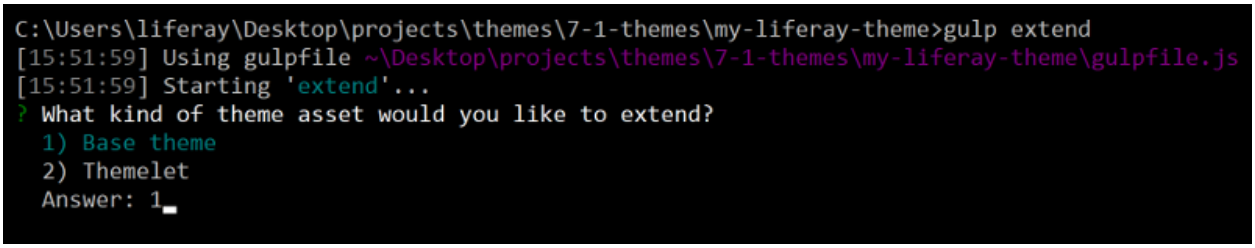


Figure 75.8: Run the `gulp extend` task to change your base theme or install a themelet.

2. Choose which base theme you want to extend. By default, themes created with the Liferay Theme Generator are based off of the styled theme. You can extend the styled or unstyled base theme, a globally installed theme, a theme published on the npm registry, or you can specify a package URL. Enter the number for the option you wish to select.

```
**Note:** You can retrieve the URL for a package by running
`npm show package-name dist.tarball`.
```

```
![ You can extend the styled or unstyled base theme, a globally installed theme, or a theme published to the npm registry.](./images/theme-dev-changing-base-themes-gulp-extend-base-theme-choice.png)
```

```
**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.
```

---

3. Your theme's `package.json` contains the updated base theme configuration :

```
{
  ...
    "liferayTheme": {
        "baseTheme": "styled",
        "screenshot": "",
        "rubySass": false,
        "templateLanguage": "ftl",
        "version": "7.1"
    },
  ...
}
```

When you build your theme's files or deploy it, your theme will inherit the updated base theme's files.

Related Topics

Configuring Your Theme's App Server
Deploying Themes
Listing Your Theme's Extensions

75.5 Copying an Existing Theme's Files

If you want to jump start developing your theme, you can copy an existing theme's files and build on top of them. The `gulp kickstart` task automates this process for you. It copies another theme's css, images, js, and templates into the `src` directory of your own. While this is similar to extending your theme with a base theme or a themelet, 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.

Note: The gulp kickstart task copies an existing theme's files into your own, which can potentially overwrite files with the same name. Proceed with caution.

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.

To kickstart your theme, follow these steps:

1. Navigate to your theme's root folder and run `gulp kickstart`.
2. Select where the theme is located. You can copy files from globally installed themes or themes published on the npm registry.

```
**Note:** To globally install a theme, run the `npm link` command from the
theme's root folder.
```

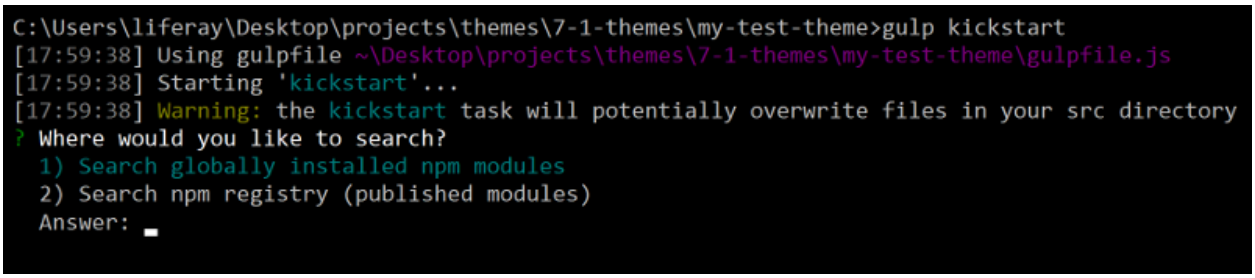


Figure 75.9: Run the `gulp kickstart` task to copy a theme's files into your own theme.

```
![ You can copy files from  globally installed themes.](./images/theme-dev-kickstarting-themes-global-theme.png)
```

3. The theme's files are copied into your own theme, jump starting development.

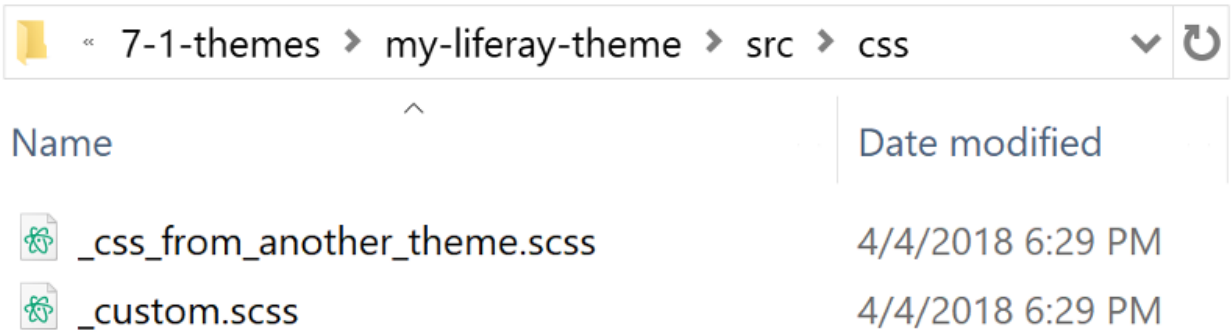


Figure 75.10: The kickstart tasks copies another theme's files into your own, potentially overwriting files.

Related Topics

Building Your Theme's files
    Creating Reusable Pieces of Code for Your Themes
    Deploying Themes

75.6 Configuring Your Theme's App Server

When your theme was first created with the Liferay Theme Generator, you had to specify the app server's location. This was done with the `gulp init` task. Your theme uses this information to deploy to the proper server. If your app server or site changes during development, you can update their configuration information by manually running the `gulp init` task.

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.

Follow these steps:

1. Navigate to your theme's root folder and run `gulp init`.

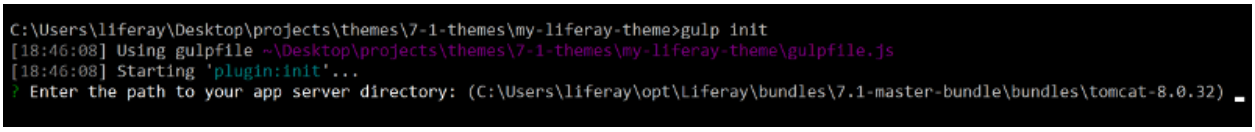


Figure 75.11: Run the `gulp init` task to update your app server configuration.

2. Enter the path to your app server and site.

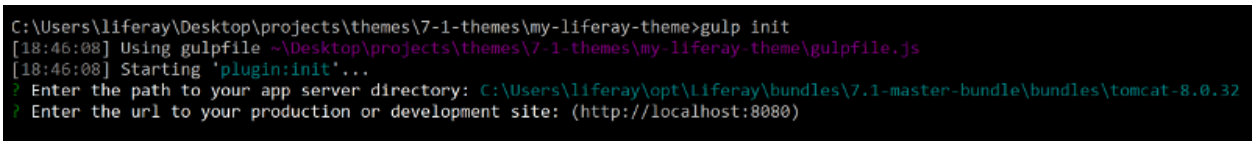


Figure 75.12: You can also run the `gulp init` task to update your site's URL.

3. Your theme's `liferay-theme.json` file contains the updated server configuration information:

```
{
  "LiferayTheme": {
    "appServerPath": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\7.1-master-bundle\\bundles\\tomcat-8.0.32",
    "deployPath": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\7.1-master-bundle\\bundles\\deploy",
    "url": "http://localhost:8080",
    "appServerPathPlugin": "C:\\Users\\liferay\\opt\\Liferay\\bundles\\7.1-master-bundle\\bundles\\tomcat-8.0.32\\webapps\\my-
liferay-theme",
    "deployed": false,
    "pluginName": "my-liferay-theme"
  }
}
```

Related Topics

Automatically Deploying Theme Changes
Changing Your Base Theme
Listing Your Theme's Extensions

75.7 Listing Your Theme's Extensions

Do you need to know what base theme/themelets your theme extends? There's a gulp task for that. While you can manually check your theme's `package.json` for this information, the `gulp status` task displays this information for you. Navigate to your theme's root folder and run `gulp status` to display your theme's extensions.

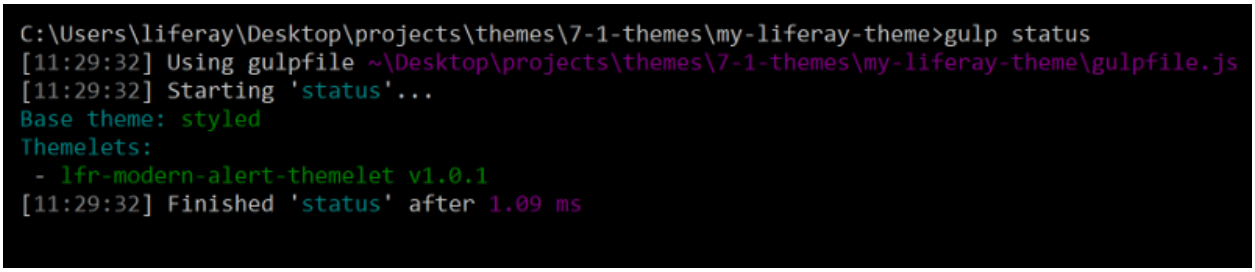


Figure 75.13: Run the `gulp status` task to list your theme's current extensions.

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.

Related Topics

Changing Your Base Theme
    Configuring Your Theme's App Server
    Creating Reusable Pieces of Code for Your Themes

75.8 Automatically Deploying Theme Changes

You may have noticed that you have to deploy your theme manually each time you make a change. This can become tedious during the development process. The `gulp watch` task lets you preview changes to your theme without requiring a full redeploy.

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.

Follow these steps to preview changes to your theme automatically:

1. Enable Developer Mode in your server. Without this enabled, the gulp watch task **will not work**.
2. Navigate to your theme's root folder and run `gulp watch`. This sets up a proxy for your app server (`http://localhost:9080`) and opens it in a new window in the browser. It also provides an IP address for you to view your app server across all devices connected to the local network. The browser is synced across all devices that use the given IP address.

```
**Note:** Live changes are only viewable on port `9080`
(`http://localhost:9080`). Live changes **are not viewable** on your app
server (e.g. `http://localhost:8080`).
```

![Run the `gulp watch` task to automatically deploy any changes to your theme.](./images/theme-dev-watching-themes-gulp-watch-startup.png)

3. Make a change to your theme and save the file. The updated files are built, compiled, and copied directly to port `9080`. CSS changes are deployed live, so no page reload is needed. For JS and template changes, **you must** reload the browser to see the changes.

```
[14:57:37] Finished 'build:remove-old-css-dir' after 356 ms
[14:57:37] Starting 'deploy:css-files'...
[14:57:37] gulp-debug: build\css\clay.css
[14:57:37] gulp-debug: build\css\font_awesome.css
[14:57:37] gulp-debug: build\css\main.css
[14:57:37] gulp-debug: 3 items
[Browsersync] 3 files changed (clay.css, font_awesome.css, main.css)
[14:57:37] Finished 'deploy:css-files' after 23 ms
```

Figure 75.14: The watch task notifies you that the changes are deployed.

4. Once you're happy with the previewed changes, deploy your theme to your app server to apply the changes.

Related Topics

Configuring Your Theme's App Server
    Copying an Existing Theme's Files
    Deploying Themes

75.9 Creating Reusable Pieces of Code for Your Themes

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 the admin tools, or a themelet that uses a custom JavaScript component for responsive embedded videos, and everything in between. For example, the Liferay Product Menu Animation Themelet simply alters the animation for the Product Menu.

If there is something you have to manually code for every theme you create, it's a good candidate for a themelet.

This tutorial demonstrates how to:

- Create a themelet to extend your theme
- Install a Themelet

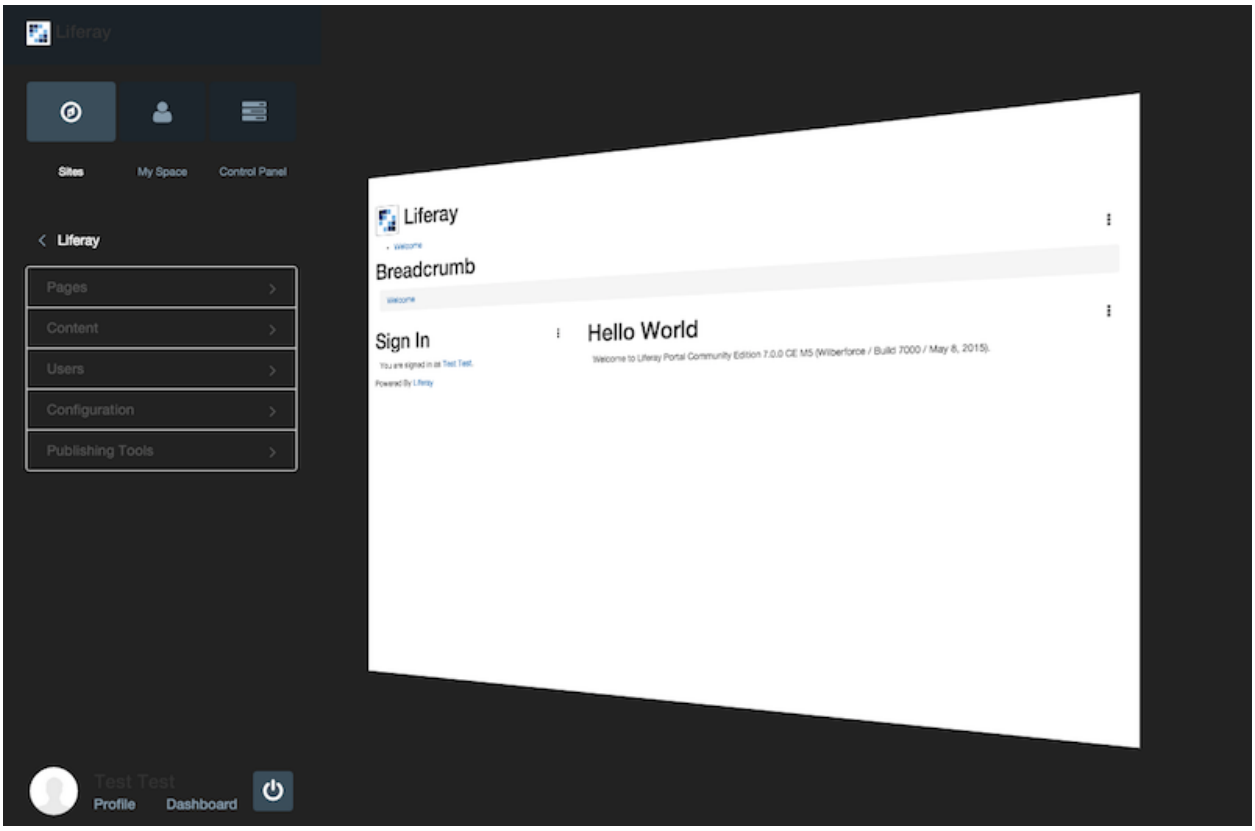


Figure 75.15: Themelets can be used to modify one aspect of the UI, that you can then reuse in your other themes.

To create a themelet, you need a theme to extend and the Liferay Theme Generator and dependencies installed, as explained in the Creating Themes tutorial.

Creating a Themelet

Follow these steps to create a themelet:

1. Open the Command Line and navigate to the folder 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.

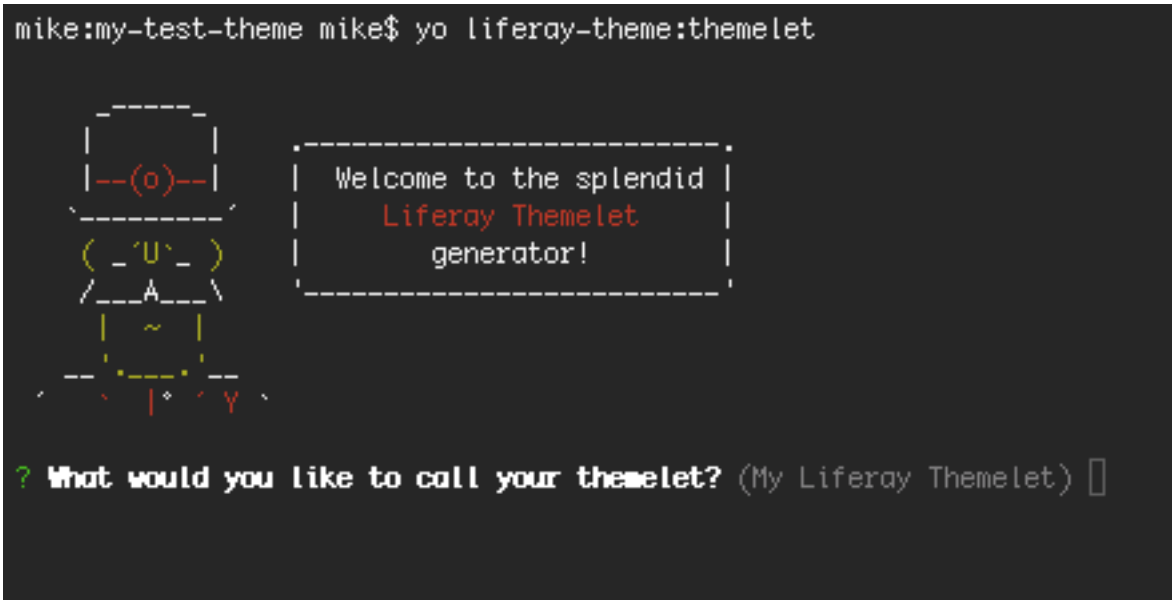


Figure 75.16: The Themelet sub-generator automates the themelet creation process, making it quick and easy.

Installing a Themelet

After you've developed your themelet, follow the steps below to install it into your theme.

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.

---

1. Navigate to your theme's root folder and run the following command:

   ```
   gulp extend
   ```

2. Choose *Themelet* as the theme asset to extend.

3. Select *Search globally installed npm modules*, *Search npm registry*, or *Specify a package URL* to locate the themelet.

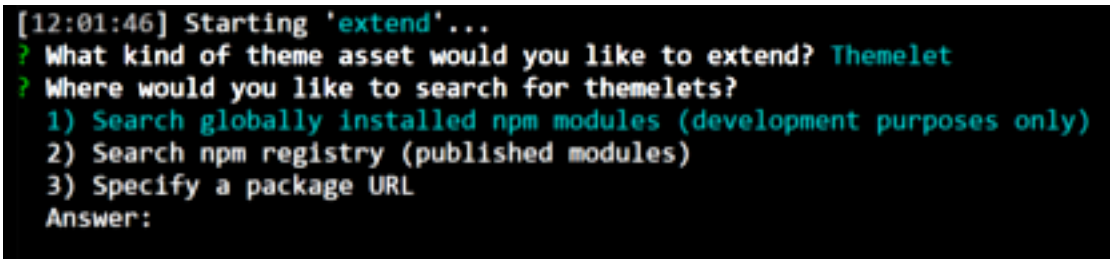


Figure 75.17: You can extend your theme using globally installed npm modules, published npm modules, or via a package URL.

```
**Note:** You can retrieve the URL for a package by running
`npm show package-name dist.tarball`.
```

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
Creating Themes

75.10 Creating a Thumbnail Preview for Your Theme

When you apply a theme to your site pages, you have to choose from the list of available themes in the site selector. The only identification for each theme is the theme's name, along with a small thumbnail preview image that gives a brief impression of the theme. By default, themes don't provide a thumbnail image, so you must create one. This is even more important when developing color schemes for a theme. Names are not displayed for color schemes, so a thumbnail preview is required to identify them.

Follow these steps to create a thumbnail preview for your theme:

1. With your theme applied to your site and an optional color scheme chosen, take a screenshot that best represents it.
2. Crop any unwanted areas and re-size the screenshot to 150 pixels high by 120 pixels wide. Your thumbnail *must be* these exact dimensions to display properly.
3. Save the image as a `.png` file named `thumbnail.png` and place it in the theme's `src/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.

Related Topics

Creating Themes
Creating Reusable Pieces of Code for Your Themes
Creating Color Schemes for Your Theme

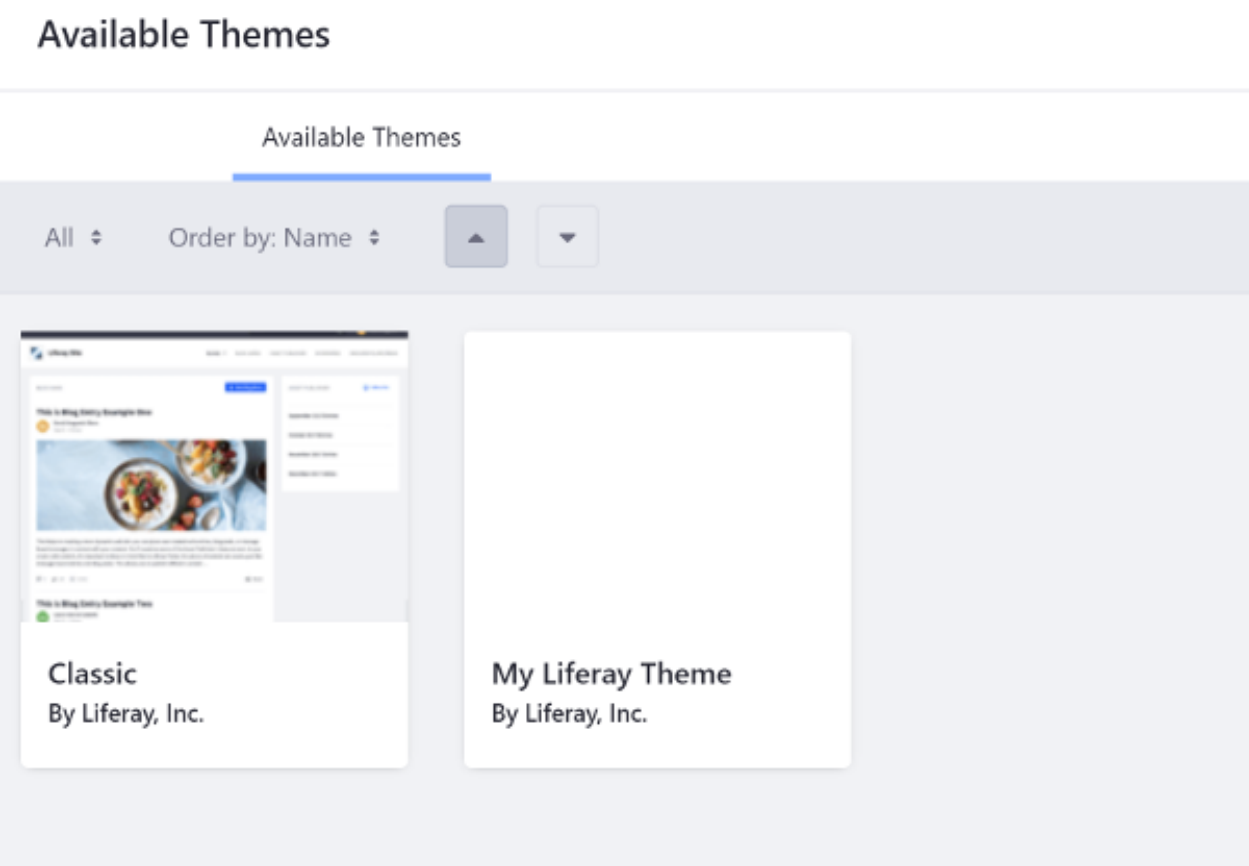


Figure 75.18: Your theme thumbnail is displayed with the rest of the available themes.

75.11 Creating Color Schemes for Your Theme

Color schemes give your theme additional color palettes. They only require a small amount of changes to your theme's CSS. This is an easy way to subtly change the look of your theme, while maintaining the same design and feel to it.

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 and add a `.scss` file to it for each color scheme your theme supports.

2. Choose a CSS class to identify each color scheme, and specify this class in the color scheme's styles. The color scheme's name is a good choice. This class is added to the site's `<body>` element when the color scheme is applied, so you *must* specify this class in the color schemes styles for them to work. For example, you could specify `.day` for a color scheme CSS file named `_day.scss`:

```
body.day { background-color: #DDF; }
.day a { color: #66A; }
```

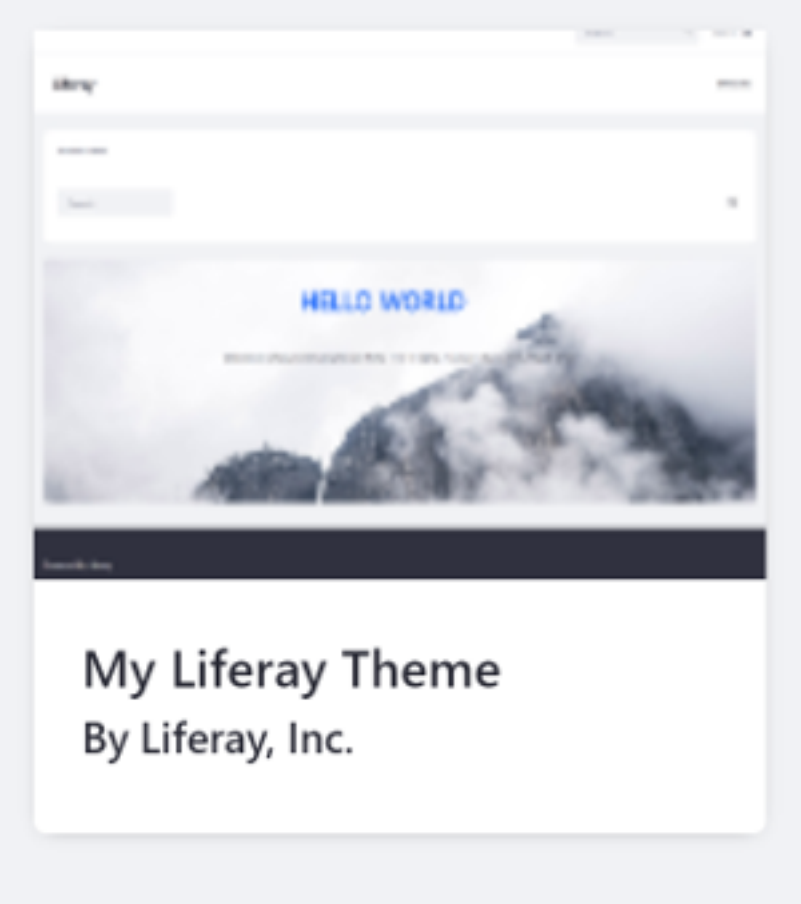


Figure 75.19: Your theme thumbnail is displayed with the rest of the available themes.

```
**Note:** The default color scheme uses the theme's `_custom.scss` for
styling, so you don't need to specify its name in its styles.
```

---

3. Import the color scheme `.scss` files into the theme's `_custom.scss` file. The example below imports `_day.scss` and `_night.scss` files:

   ```
   @import "color_schemes/day";
   @import "color_schemes/night";
   ```

4. Create a folder for each color scheme in your theme's `images` folder, and add a thumbnail preview for each color scheme. The folder name *must match* the color scheme's CSS class name you specified in step 2.

5. Open the theme's `liferay-look-and-feel.xml` file and follow the pattern below to add the default color scheme. A default color scheme is required so users can return to the theme's default look and feel after choosing a different color scheme. Note that the color scheme's `name` value is arbitrary, but the `<css-class>` element's value *must match* the CSS class you specified

in step 2. If the default color scheme styling is in the theme's `_custom.scss` file, use `default` for the `<css-class>`. Also note the inclusion of the `<color-scheme-images-path>` element. This specifies the theme thumbnail image location. Place this element in the first color scheme to configure the images path for all the color schemes:

```
<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>
```

```
**Note:** Color schemes are sorted alphabetically by `name` rather than `id`.
For example, a color scheme named `Clouds` and `id` `02` would be selected
by default over a color scheme named `Day` 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.
```

---

6. Add the remaining color schemes below the default color scheme, using the pattern below:

```
<color-scheme id="id-number" name="Color Scheme Name">
   <css-class>color-scheme-css-class</css-class>
</color-scheme>
```

An example `liferay-look-and-feel.xml` file is shown below:

```
<look-and-feel>
    <compatibility>
        <version>7.1.0+</version>
    </compatibility>
    <theme id="my-liferay-theme" name="My Liferay Theme">
        <template-extension>ftl</template-extension>
        <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="Red">
            <css-class>red</css-class>
        </color-scheme>
        <color-scheme id="03" name="Green">
            <css-class>green</css-class>
        </color-scheme>
    </theme>
</look-and-feel>
```

There you have it. Now you can go color scheme crazy with your themes!

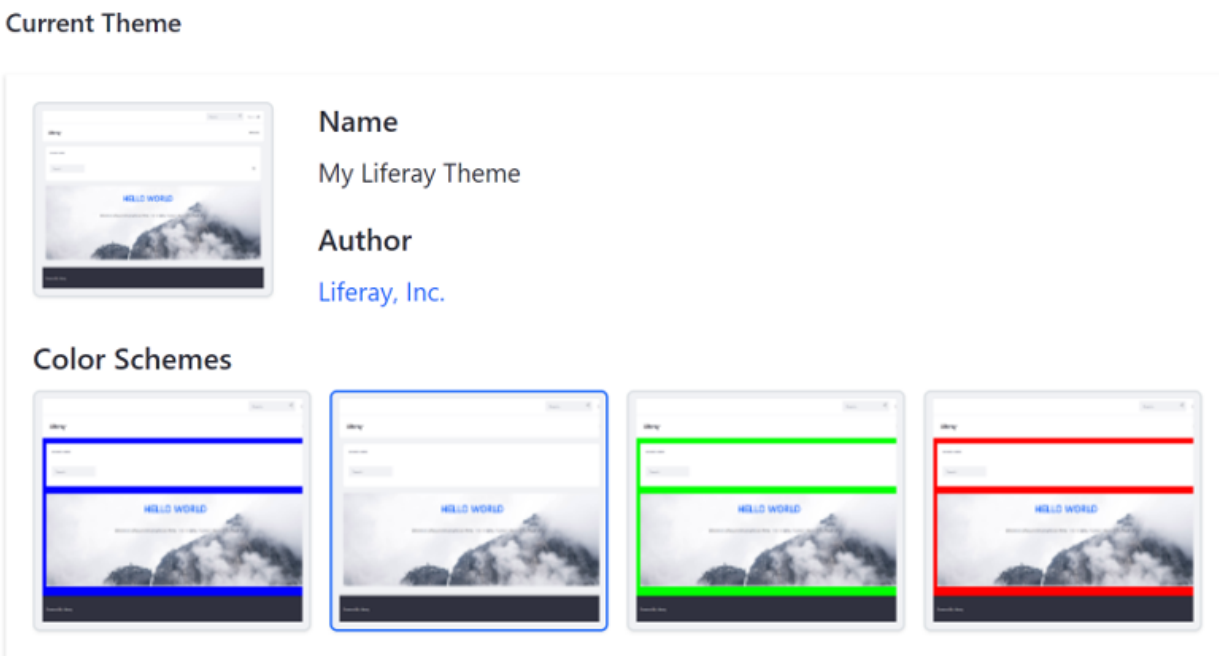


Figure 75.20: Color schemes give administrators some choices for your theme's look.

Related Topics

Creating Layout Templates

Creating a Thumbnail Preview for Your Theme

75.12 Making Configurable Theme Settings

Every time you want to make a change to a theme, you must make the change and then deploy it to your server. For a Site Administrator, this process is tedious, especially if it's a minor change to a theme, such as a banner color change. What happens when it must be changed back? It must be deployed again. For larger theme changes, this process is unavoidable, but for smaller changes, there's a better way: configurable theme settings. The Theme Developer can control the visibility and value of theme elements and provide variations of theme elements for the Site Administrator to configure and choose in the Control Panel. This tutorial covers how to create configurable theme settings for your theme.

Follow these steps:

1. Open your theme's `WEB-INF/liferay-look-and-feel.xml` file, or create it if it doesn't exist, and follow the pattern below to nest a `<setting/>` element inside the parent `<settings>` element for each setting the theme requires:

```
<look-and-feel>
    <compatibility>
        <version>7.1.0+</version>
    </compatibility>
    <theme id="your-theme-name" name="Your Theme Name">
```

```
        <template-extension>ftl</template-extension>
    <settings>
      <setting configurable="true" key="theme-setting-key"
      options="true,false" type="select" value="true" />
      <setting configurable="true" key="theme-setting-key"
      type="text" value="My placeholder text" />
    </settings>
    <portlet-decorator>
      portlet decorators...
        </portlet-decorator>
    </theme>
</look-and-feel>
```

The example configuration below adds a text input setting for custom text:

```
<settings>
  <setting configurable="true" key="my-custom-text"
  type="text" value="Liferay rocks!" />
</settings>
```

The available theme `<setting>` attributes are shown below:

`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.

2. Add `init_custom.ftl` to your theme templates if it doesn't already exist, and follow the patterns below to define your theme setting variables in it:

   Booleans (a select box with the options `true` and `false` or a checkbox with values `yes` and `no`):

   ```
   <#assign my_variable_name =
   getterUtil.getBoolean(themeDisplay.getThemeSetting("theme-setting-key"))/>
   ```

   Strings (a text input or text area input):

   ```
   <#assign my_variable_name =
   getterUtil.getString(themeDisplay.getThemeSetting("theme-setting-key"))/>
   ```

   The example configuration below adds the custom text setting:

   ```
   <#assign my_custom_text =
   getterUtil.getString(themeDisplay.getThemeSetting("my-custom-text"))/>
   ```

3. Apply your theme settings to the theme template. The example configuration below prints the value for the `my_custom_text` variable in a `<p>` element:

   `portal_normal.ftl`:

```
<p>${my_custom_text}</p>
```

4. Deploy and install the theme, and open the *Control Menu → Site Administration → Build → Pages*. From this point, you can configure theme settings for two scopes:

    - **Public or private page set:** *Click* the gear icon next to the page set you want to configure. Then, under the *Look and Feel* tab, choose your settings and click *Save* to apply the changes to the page set.
    - **An individual page:** Open the Actions menu next to the page and select *Configure*. Then, click the *Look and Feel* tab and select the *Define a Specific look and feel for this page* option. Choose your settings and click *Save* to apply the changes to the page.

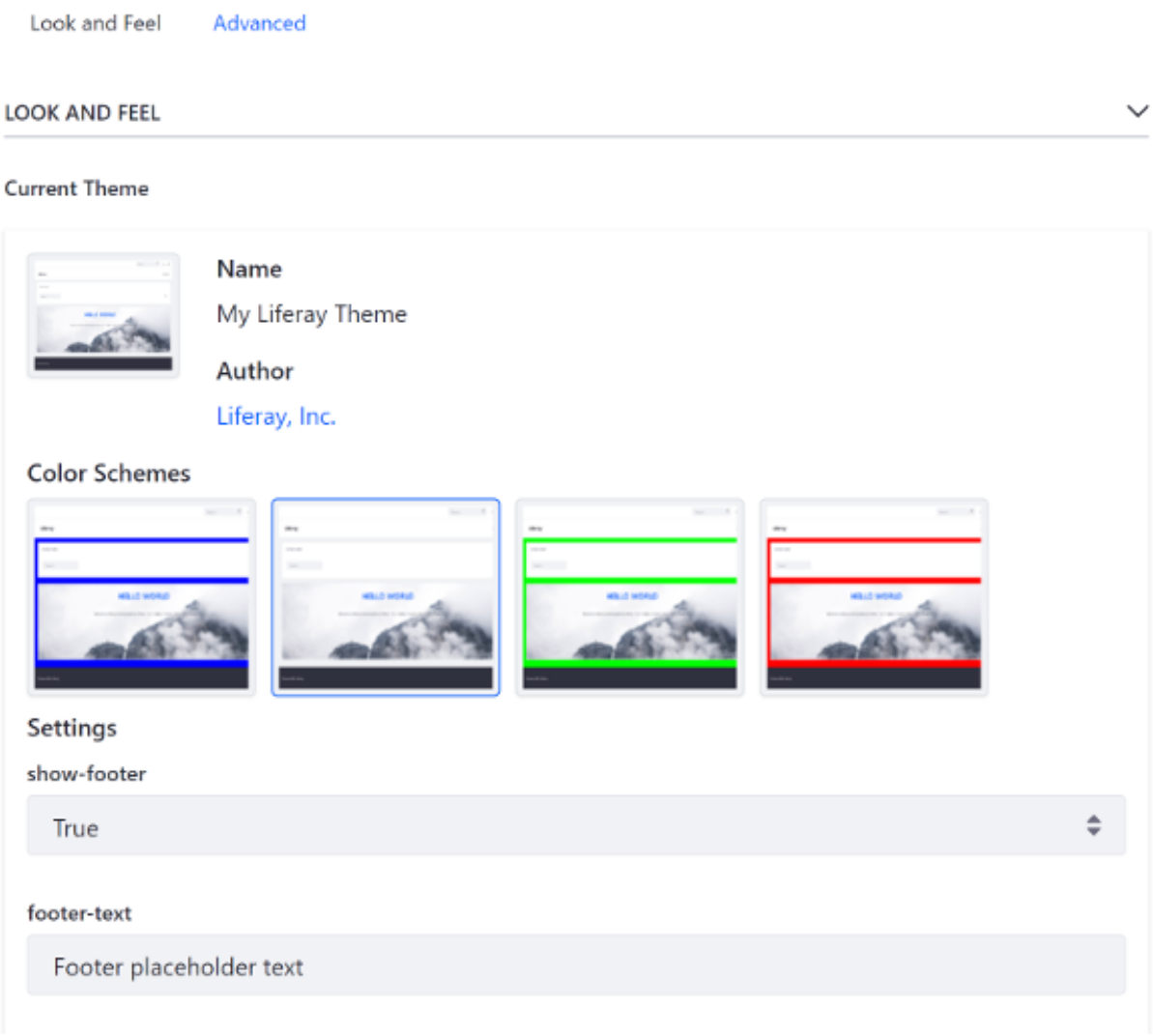


Figure 75.21: Here are examples of configurable settings for the site Admin.

Now you know how to make configurable theme settings for your themes!

Related Topics

Creating Reusable Pieces of Code for Your Themes
    Listing Your Theme's Extensions
    Importing Resources with a Theme

75.13 Overwriting and Extending Liferay Theme Tasks

Themes created with the Liferay Theme Generator have access to several default gulp theme tasks that provide the standard features required to develop and build your theme (build, deploy, watch, etc.). You may, however, want to run additional processes—such as minifying your JavaScript files—on your theme's files prior to deploying the theme to the server. The Liferay Theme Generator's APIs expose a `hookFn` property that lets you hook into the default gulp theme tasks to inject your own logic.

Follow these steps to hook into the default Liferay theme tasks:

1. Identify the gulp task or sub task that you want to hook into or overwrite. The tasks and their sub tasks are listed in their `[task-name].js` file in the `tasks` folder of the `liferay-theme-tasks` package. For example, the gulp `build` task and sub tasks are defined in the `build.js` file:

```
'build:clean',
'build:base',
'build:src',
'build:web-inf',
'build:liferay-look-and-feel',
'build:hook',
'build:themelets',
'build:rename-css-dir',
'build:prep-css',
'build:compile-css',
'build:fix-url-functions',
'build:move-compiled-css',
'build:remove-old-css-dir',
'build:fix-at-directives',
'build:r2',
'build:war',
```

2. Open your theme's `gulpfile.js` file and locate the `liferayThemeTasks.registerTasks()` method. This method registers the default gulp theme tasks. Add the `hookFn` property to the `registerTasks()` method's configuration object, making sure to pass in the `gulp` instance:

```
liferayThemeTasks.registerTasks({
  gulp: gulp,
  hookFn: function(gulp) {

  }
});
```

3. Inside the `hookFn()` function, use the `gulp.hook()` method to specify the theme task or sub task that you want to hook into. You can inject your code before or after a task by prefixing it with the `before:` or `after:` keywords. Alternatively, you can use the `gulp.task()` method to overwrite a gulp task. Both methods have two parameters: the task or sub task you want to hook into and a callback function that invokes `done` or returns a stream with the logic that you want to inject. A few example configuration patterns are shown below:

```
liferayThemeTasks.registerTasks({
  gulp: gulp,
  hookFn: function(gulp) {
    gulp.hook('before:build:src', function(done) {
      // Fires before build:src task
    });

    gulp.hook('after:build', function(done) {
      // Fires after build task
    });

    gulp.task('build:base', function(done) {
      // Overwrites build:base task
    });
  }
});
```

The example below fires before the `build:war` sub-task and reads the JavaScript files in the theme's `build` folder, minifies them with the `gulp-uglify` module, places them back in the `./build/js` folder, invokes `done`, and finally logs that the JavaScript was minified. To follow along, replace your theme's `gulpfile.js` with the contents shown below, install the gulp-uglify module and the fancy-log module, and run `gulp deploy`:

```
'use strict';

var gulp = require('gulp');
var log = require('fancy-log');
var uglify = require('gulp-uglify');
var liferayThemeTasks = require('liferay-theme-tasks');

liferayThemeTasks.registerTasks({
   gulp: gulp,
   hookFn: function(gulp) {
     gulp.hook('before:build:war', function(done) {
      // Fires before build `war` task
      gulp.src('./build/js/*.js')
      .pipe(uglify())
      .pipe(gulp.dest('./build/js'))
      .on('end', done);
      log('Your JS is now minified...');
      });
   }
});
```

You should see something similar to the output shown below:

```
[15:58:07] Finished 'build:r2' after 198 ms
[15:58:07] Starting 'build:war'...
[15:58:07] Your JS is now minified...
[15:58:07] Starting 'plugin:version'...
[15:58:07] Finished 'plugin:version' after 2.52 ms
```

Note: The `hook` callback function must invoke the `done` argument or return a stream.

Now you know how to hook into and overwrite the default Liferay theme tasks!

Related Topics

Creating Themes

Creating Reusable Pieces of Code for Your Themes

Using Developer Mode with Themes

75.14 Compiling and Building Themes with Ant, Gradle, and Maven

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.

Related Topics

Creating Themes
    Theme Template

75.15 Injecting Additional Context Variables and Functionality into Your Templates

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), Liferay DXP 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 DXP-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 DXP, 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 DXP 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 DXP'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, examine the `ProductMenuTemplateContextContributor` class used by Liferay DXP by default. This class injects variables into Liferay DXP'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.

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 DXP, you're free to fully harness the power of your chosen templating language.

Related Topics

Customizing the Product Menu
    Creating Themes
    Theme Contributors

75.16 Packaging Independent UI Resources for Your Site

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

The standard UI for User menus and navigation are packaged as Theme Contributors. For example, the Control Menu, Product Menu, and Simulation Panel are packaged as Theme Contributor modules in Liferay DXP, 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.

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 module's `bnd.bnd`:

   ```
   Bundle-Name: Liferay Product Navigation Product Menu Theme Contributor
   Bundle-SymbolicName: com.liferay.product.navigation.product.menu.theme.contributor
   Bundle-Version: 3.0.4
   Liferay-Theme-Contributor-Type: product-navigation-product-menu
   Web-ContextPath: /product-navigation-product-menu-theme-contributor
   ```

   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]
   ```

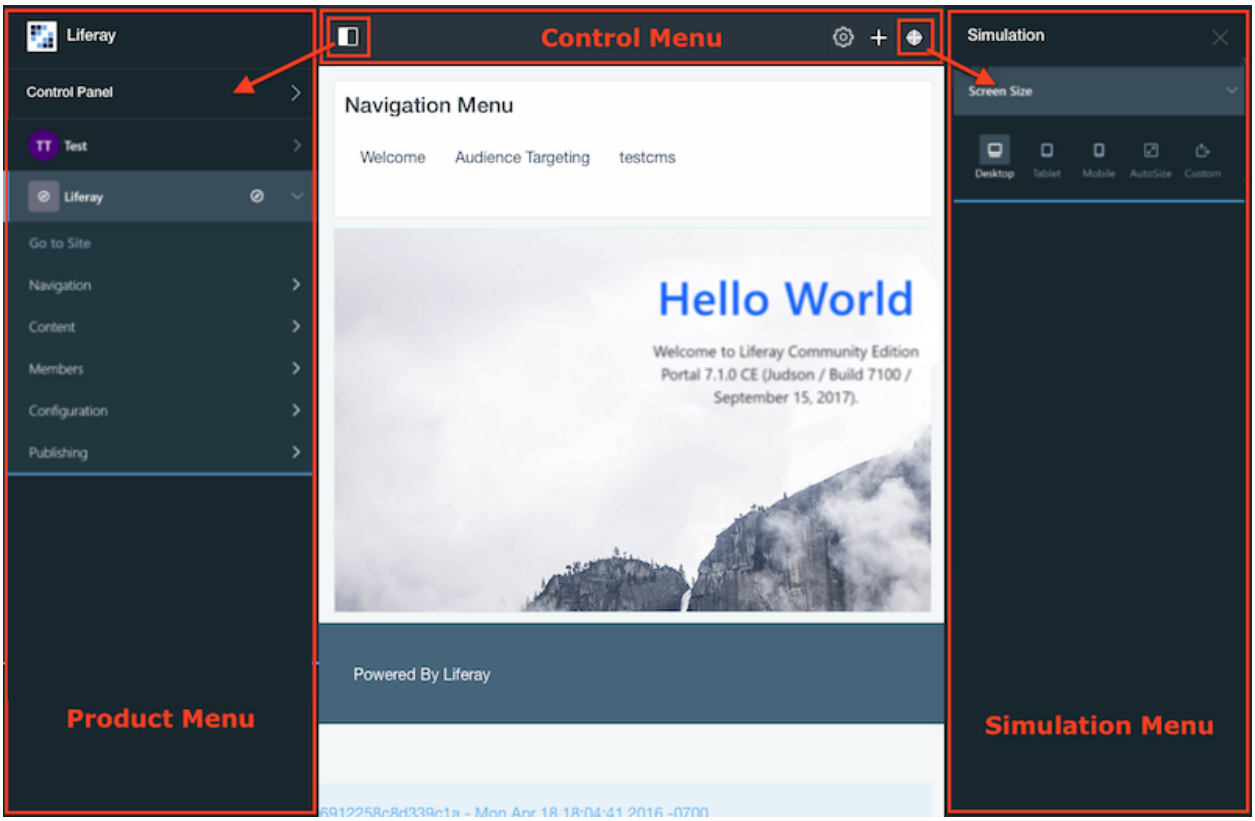


Figure 75.22: The Control Menu, Product Menu, and Simulation Panel are packaged as Theme Contributor modules.

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

Creating Themes
    Themelets
    Importing Resources with Your Themes
    Theme Contributor Template

75.17 Using Liferay DXP's Macros in Your Theme

Macros can 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 macros in `FTL_Liferay.ftl` template that you can use in your FreeMarker theme templates to include theme resources, standard portlets, and more. 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 macros in your FreeMarker theme templates.

Follow these steps:

1. Select one of the macros shown in the table below:

```
Macro | Parameters | Description | Example |
--- | --- | --- | --- |
breadcrumbs | default_preferences | Adds the Breadcrumbs portlet with optional preferences | `<@liferay.breadcrumbs />` |
control_menu | N/A | Adds the Control Menu portlet | `<@liferay.control_menu />` |
css | file_name | Adds an external stylesheet with the specified file name location | `<@liferay.css file_name="${css_folder}/mycss.css"/>` |
date | format | Prints the date in the current locale with the given format | `<@liferay.date format="/yyyy/MM/dd/HH/" />` |
js | file_name | Adds an external JavaScript file with the specified file name source | `<@liferay.js file_name="${javascript_folder}/myJs.js"/>` |
language | key | Prints the specified language key in the current locale | `<@liferay.language key="last-modified" />` |
language_format | arguments<br/>key | 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*. | `<@liferay.language_format arguments="${site_name}" key="go-to-
x" />` |
languages | default_preferences | Adds the Languages portlet with optional preferences | `<@liferay.languages />` |
navigation_menu | default_preferences<br/>instance_id | Adds the Navigation Menu portlet with optional preferences and instance ID. | `<@liferay.nav
search | default_preferences | Adds the Search portlet with optional preferences | `<@liferay.search />` |
search_bar | default_preferences | Adds the Search Bar portlet with optional preferences | `<@liferay.search_bar />` |
user_personal_bar | N/A | Adds the User Personal Bar portlet | `<@liferay.user_personal_bar />` |
```

---

2. Call the macro in your theme template. Liferay DXP's default FreeMarker macro calls are namespaced with `liferay`. For example, to include the Search Bar portlet, you would add the macro call shown below:

```
<@liferay.search_bar>
```

3. Include any required or optional arguments, such as portlet preferences, in the macro call. For example, Liferay DXP's `language` macro directive includes a language key parameter:

```
<#macro language
  key
>
  ${languageUtil.get(locale, key)}
</#macro>
```

   You can pass an argument in the macro call like this:

```
<@liferay.language key="powered-by" />
```

   The example below sets the Search portlet's Portlet Decorator to barebone:

```
<@liferay.search default_preferences=
  freeMarkerPortletPreferences.getPreferences(
    "portletSetupPortletDecoratorId", "barebone"
  )
/>
```

You can also pass multiple portlet preferences in an object, as shown in the example below for the Navigation Menu portlet:

```
<#assign secondaryNavigationPreferencesMap =
  {
    "displayStyle": "ddmTemplate_NAVBAR-BLANK-JUSTIFIED-FTL",
    "portletSetupPortletDecoratorId": "barebone",
    "rootLayoutType": "relative",
    "siteNavigationMenuId": "0",
    "siteNavigationMenuType": "1"
  }
/>

<@liferay.navigation_menu
  default_preferences=
  freeMarkerPortletPreferences.getPreferences(secondaryNavigationPreferencesMap)
  instance_id="main_navigation_menu"
/>
```

Note: Portlet preferences are unique to each portlet, so first you must determine which preferences you want to configure. There are two ways to determine the proper key/value pair for a portlet preference. The first is to set the portlet preference manually, and then check the values in the `portletPreferences.preferences` column of the database as a hint for what to configure.

Another approach is to search each app in your bundle for the keyword `preferences--`. This returns app JSPs that have the portlet preferences defined for the portlet.

Now you know how to use Liferay DXP's macros in your theme templates!

Related Topics

Creating Themes

Creating Reusable Pieces of Code for Your Themes

Theme Reference Guide

CHAPTER 76

IMPORTING RESOURCES WITH A THEME

To truly appreciate a theme, you must view it with content. Showcasing a theme in the proper context is key to communicating the true intentions of its design. Who better to do this than the theme's designer? Designers can provide a sample context that optimizes the design of their themes. The Resources Importer does this for you.

Note: The Resources Importer is deprecated as of 7.0.

The Resources Importer module lets theme developers import files and web content with a theme. Administrators can use the site or site template created by the Resources Importer to showcase the 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, go to Sites or Site Templates in the Control Panel and immediately see their new theme in action.

Using the Resources Importer involves the following steps:

- Preparing and Organizing Resources
- Creating a Sitemap for the Resources Importer
- Defining Assets for the Resources Importer (optional)
- Specifying Where to Import Your Theme's Resources

This section of tutorials explains how to use the Resources Importer to import resources with your theme.

76.1 Preparing and Organizing Web Content for the Resources Importer

You must create the resources to import with your theme. You can create resources from scratch and/or bring in resources that you've already created. This tutorial covers how to prepare and organize your resources for the Resources Importer.

First, you must prepare your web content for the Resources Importer.

Preparing Your Web Content

You can leverage your HTML (basic web content), JSON (structures), or VM or FTL (templates) files with the Resource Importer. All web content articles require a structure and template. Note that some articles may share the same structure and perhaps even the same template—this is the case for all basic web content articles. Follow these steps to prepare your web content articles:

1. Select *Edit* from the article's options menu, click the *Options* icon at the top right of the page and select *View Source*. Copy the article's raw XML into an XML file locally. Create a folder for the article under `resources-importer/journal/articles/` and rename it as desired. The web content article's XML fills in the data required by the structure. An example web content article's XML is shown below:

```
<?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>
```

2. Download the web content article's structure. Open the structure and click the *Source* tab to view the structure's file. Copy and paste its contents into a new JSON file in the `resources-importer/journal/structures/` folder. The structure JSON sets a wireframe, or blueprint, for an article's data. If you're saving a basic web content article, you can copy the structure below (replace `en_US` with your language):

```
{
    "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": "text",
            "localizable": true,
            "name": "content",
            "readOnly": false,
            "repeatable": false,
            "required": false,
            "showLabel": true,
            "type": "ddm-text-html"
        }
    ]
}
```

3. Download the structure's matching template. Open the Actions menu for the structure and select *Manage Templates* to view the templates that use it. Create a folder for the template under `resources-importer/journal/templates/` and copy and paste its contents into a new FTL file. The template defines how the data should be displayed. If you're saving a basic web content article, you can copy the FreeMarker template below:

```
${content.getData()}
```

Repeat the steps above for each web content article you have. Note that some web content articles may share the same structure and template; In these cases, only one copy of the structure and template is required for all web content articles that use them. Once your web content articles are saved, you can place them in their proper folder structure.

Organizing Your Resources

Add your resources under the `[theme-name]/src/WEB-INF/src/resources-importer` folder and its sub-folders. Place your resources in the folders outlined below:

- `[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 (assets)
    - `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`.

Related Topics

Creating a Sitemap for the Resources Importer
    Defining Assets for the Resources Importer
    Specifying Where to Import Your Theme's Resources

CHAPTER 77

CREATING A SITEMAP FOR THE RESOURCES IMPORTER

You have two options for specifying resources to be imported with your theme: a sitemap or an archive LAR file. Using a `sitemap.json` file is the most flexible approach, so we recommend it; unlike LAR files, a `sitemap.json` can be created in one version of Liferay DXP and used in another. LAR files are version-specific, and can only be imported in the same version in which they were created.

The `sitemap.json` 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 to import as a site or site template. If you're developing themes for Liferay Marketplace, you must use the `sitemap.json` to specify resources to be imported with your theme. Even if you're not familiar with JSON, the `sitemap.json` file is easy to understand. An example `sitemap.json` file is shown below:

```
{
    "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"
        }
    ]
}
```

If you don't understand the sitemap at this point, don't worry. This section of tutorials covers how to create a sitemap for your theme, from defining pages to defining portlets.

77.1 Defining Layout Templates in a Sitemap

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. When defined outside the scope of a page, the `layoutTemplateId` sets the default layout template for the theme's pages:

```
{
  "layoutTemplateId": "2_columns_ii",
  "publicPages": [
    {
      "friendlyURL": "/my-page",
      "name": "My Page",
      "title": "My Page"
    }
  ]
}
```

When placed inside a page configuration, the `layoutTemplateId` sets the layout template for the individual page. In the example below, the Hidden Page and the Welcome page both use the default `2_columns_ii` layout template, while the Custom Layout Page overrides the default layout template and uses the `2_columns_i` layout template instead:

```
{
  "layoutTemplateId":"2_columns_ii",
  "publicPages": [
      {
        "friendlyURL": "/welcome-page",
        "name": "Welcome",
        "title": "Welcome"
      },
      {
        "friendlyURL": "/custom-layout-page",
        "name": "Custom Layout Page",
```

```
      "title": "Custom Layout Page",
      "layoutTemplateId": "2_columns_i"
    },
    {
      "friendlyURL": "/hidden-page",
      "name": "Hidden Page",
      "title": "Hidden Page",
      "hidden": "true"
    }
  ]
}
```

Related Topics

Preparing and Organizing Web Content for the Resources Importer
    Defining Portlets in a Sitemap
    Specifying Where to Import Your Theme's Resources

77.2 Defining Pages in a Sitemap

A sitemap defines the layouts—pages—that your site or site template uses. This tutorial covers the configuration options that are available for pages in a sitemap.

Note: Pages are imported into a site template by default. Site templates only support the importing of either public page sets or private page sets, not both.

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.

You can specify a name for a page, title, friendly URL, whether it is hidden, and much more. The example below defines a default layout template and both public and private page sets to import into a site:

```
{
  "layoutTemplateId": "2_columns_ii",
  "privatePages": [
      {
        "friendlyURL": "/private-page",
            "name": "Private Page",
            "title": "Private Page"
      }
  ],
  "publicPages": [
      {
        "friendlyURL": "/welcome-page",
        "nameMap": {
            "en_US": "Welcome",
            "fr_FR": "Bienvenue"
        },
        "title": "Welcome"
      },
      {
        "friendlyURL": "/custom-layout-page",
        "name": "Custom Layout Page",
        "title": "Custom Layout Page",
        "layoutTemplateId": "2_columns_i"
      },
      {
        "friendlyURL": "/hidden-page",
```

```
        "name": "Hidden Page",
        "title": "Hidden Page",
        "hidden": "true"
      }
  ]
}
```

You can create child pages by configuring the `layouts` element for a page configuration:

```
{
"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"
}
```

These examples use some of the available page configuration attributes. The full list is shown below.

colorSchemeId: Specifies a different color scheme (by ID) than the default color scheme to use for the page.

columns: Specifies the column contents for the page.

friendlyURL: Sets the page's friendly URL.

hidden: Sets whether the page is hidden.

layoutCss: Sets custom CSS for the page to load after the theme.

layoutPrototypeLinkEnabled: Sets whether the page inherits changes made to the page template (if the page has one).

layoutPrototypeName: Specifies the page template (by name) to use for the page. 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 page. If `layoutPrototypeName` is defined, this is not required.

layouts: Specifies child pages for a page set.

name: The page's name.

nameMap: Passes a name object with multiple name key/value pairs. You can use this to pass translations for a page's title, as shown in the example above.

privatePages: Specifies private pages.

publicPages: Specifies public pages.

themeId: Specifies a different theme (by ID) than the default theme bundled with the `sitemap.json` to use for the page.

title: The page's title.

type: Sets the page 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 page `type`.

Related Topics

Preparing and Organizing Web Content for the Resources Importer
    Defining Layout Templates in a Sitemap
    Specifying Where to Import Your Theme's Resources

77.3 Defining Portlets in a Sitemap

You can embed portlets in a sitemap for the pages you define. You can embed them with the default settings or provide portlet preferences for a more custom look and feel. This tutorial covers both these options.

Follow these steps:

1. Note the portlet's ID. This is the `javax.portlet.name` attribute of the portlet spec. For convenience, The IDs for portlets available out-of-the-box are listed in the Fully Qualified Portlet IDs reference guide. For custom portlets, this property is listed in the portlet class as the `javax.portlet.name=` service property.

2. List the portlet ID in the column of the layout you want to display the portlet on. An example configuration is shown below:

```
{
    "layoutTemplateId": "2_columns_ii",
    "publicPages": [
        {
            "columns": [
                [
                    {
                        "portletId": "com_liferay_login_web_portlet_LoginPortlet"
                    },
                    {
                        "portletId":
                        "com_liferay_site_navigation_menu_web_portlet_SiteNavigationMenuPortlet"
                    }
                ],
                [
                    {
                        "portletId": "com_liferay_hello_world_web_portlet_HelloWorldPortlet"
                    }
                ]
            ],
            "friendlyURL": "/home",
            "nameMap": {
                "en_US": "Welcome",
                "fr_FR": "Bienvenue"
            },
            "title": "Welcome"
        }
    ]
}
```

   This approach embeds the portlet with its default settings. To customize the portlet, you must configure the portlet's preferences, as described in the next step.

3. Optionally specify portlet preferences for a portlet with the `portletPreferences` key. Below is an example for the Web Content Display portlet:

```
{
    "portletId": "com_liferay_journal_content_web_portlet_JournalContentPortlet",
    "portletPreferences": {
        "articleId": "Custom Title.xml",
        "groupId": "${groupId}",
        "portletSetupPortletDecoratorId": "decorate",
        "portletSetupTitle_en_US": "Web Content Display with Custom Title",
        "portletSetupUseCustomTitle": "true"
    }
}
```

portletSetupPortletDecoratorId: Specifies the portlet decorator to use for the portlet (`borderless` || `barebone` || `decorate`). See the Applying Portlet Decorators to Embedded Portlets tutorial for more info.

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, as shown in the example below:

```
"portletId": "com_liferay_asset_publisher_web_portlet_AssetPublisherPortlet",
    "portletPreferences": {
        "displayStyleGroupId": "10197",
        "displayStyle": "ddmTemplate_6fe4851b-53bc-4ca7-868a-c836982836f4"
}
```

Portlet preferences are unique to each portlet, so first you must determine which preferences you want to configure. There are two ways to determine the proper key/value pair for a portlet preference. The first is to set the portlet preference manually, and then check the values in the `portletPreferences.preferences` 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"
```

Another approach is to search each app in your bundle for the keyword `preferences--`. This returns some of the app's JSPs that have the portlet preferences defined for the portlet.

Note: 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.

Related Topics

Preparing and Organizing Web Content for the Resources Importer

Defining Pages in a Sitemap

Specifying Where to Import Your Theme's Resources

77.4 Retrieving Portlet IDs with the Gogo Shell

To include a portlet in a sitemap, you must have its ID. For convenience, IDs for out-of-the-box portlets appear in the Fully Qualified Portlet IDs reference guide. If you've installed purchased or developed portlets, you can retrieve their IDs using the Gogo Shell, as this tutorial instructs.

Follow these steps to use the Gogo Shell to retrieve a portlet ID:

1. Open the Control Panel and go to Configuration → Gogo Shell.
2. Run the command `lb [app prefix]`, and locate the app's web bundle. For example, run `lb blogs` to find the blogs web bundle.

   ```
   100|Active     |   10|Liferay Blogs Web (3.0.7)
   ```

3. Run the command `scr:list [bundle ID]`, and locate the app's component ID. The blogs portlet entry is shown below. The last number preceding the bundle's state is the component ID:

   ```
   [ 100] com.liferay.blogs.web.internal.portlet.BlogsPortlet enabled
   [ 196] [active]
   ```

4. Run the command `scr:info [component ID]` to list the portlet's information. For example, to list the info for the blogs portlet component, run `scr:info 196`. Note that the bundle and/or component ID may be different for your instance.
5. Search for `javax.portlet.name` in the results. `javax.portlet.name`'s value is the portlet ID required for the sitemap. The blogs portlet's ID is shown below:

   ```
   javax.portlet.name = com_liferay_blogs_web_portlet_BlogsPortlet
   ```

```
    javax.portlet.init-param.template-path = /
    javax.portlet.name = com_liferay_blogs_web_portlet_BlogsPortlet
    javax.portlet.resource-bundle = content.Language
    javax.portlet.security-role-ref = guest,power-user,user
    javax.portlet.supported-public-render-parameter = [categoryId, resetCur, tag]
    javax.portlet.supports.mime-type = text/html
Component Configuration:
  ComponentId: 196
  State: active
```

Figure 77.1: Portlet IDs can be found via the Gogo Shell.

Related Topics

Defining Pages in a Sitemap
    Defining Portlets in a Sitemap
    Preparing and Organizing Web Content for the Resources Importer

77.5 Defining Assets for the Resources Importer

The `sitemap.json` file defines the pages of the site or site template to import—along with the layout templates, portlets, and portlet preferences of these pages. You may also want to provide details about the assets you include with the theme. An `assets.json` file lets you provide this information. Create the `assets.json` in your theme's `[theme-name]/src/WEB-INF/src/resources-importer` folder.

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.json",
            "smallImage": "company_logo.png"
        }
    ]
}
```

Now you know how to configure assets for your web content!

Related Topics

Preparing and Organizing Web Content for the Resources Importer
Creating a Sitemap for the Resources Importer
Specifying Where to Import Your Theme's Resources

77.6 Specifying Where to Import Your Theme's Resources

By default, resources are imported into a new site template named after the theme, but you can also import resources into a new site or existing sites or site templates. This tutorial covers all these options.

First you must enable Developer Mode for the Resources Importer.

Enabling Developer Mode

Before specifying where to import your resources, you must enable Developer Mode in your theme. To do this, add the following property to your theme's `liferay-plugin-package.properties` file:

`resources-importer-developer-mode-enabled=true`

This is enabled by default for themes generated with the Liferay Theme Generator. This is a convenience feature for theme developers. With this setting enabled, importing resources into 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 must 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.**

With Developer Mode enabled in the Resource Importer, you can choose where you want to import your theme's resources.

Importing Resources into Existing Site Templates and Sites

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 template, you must specify a value for the `resources-importer-target-value` property in your theme's `liferay-plugin-package.properties` file:

```
#resources-importer-target-class-name

resources-importer-target-value=[site-template-name]
```

You must import your resources into a site if you define both public and private page sets in your `sitemap.json`. To import resources into an existing site, uncomment the `resources-importer-target-class-name` property and set it to `com.liferay.portal.kernel.model.Group`:

```
resources-importer-target-class-name=com.liferay.portal.kernel.model.Group

resources-importer-target-value=[site-name]
```

Double check the name that you're specifying. If you specify the wrong value, you could end up deleting (and re-creating) the wrong site or site template!

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 deployed to a production Liferay instance or a theme 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.

To view your theme and its resources, deploy the theme, log in as an administrator, and check the Sites or Site Templates section of the Control Panel to make sure 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, open its actions menu and select *View Pages* to see it.
- If you imported directly into a site, open its actions menu and select *Go to Public Pages* to see it.

It's just that easy to import resources with your theme!

Related Topics

Preparing and Organizing Web Content for the Resources Importer
Creating a Sitemap for the Resources Importer
Defining Assets for the Resources Importer

77.7 Archiving Site Resources

Although a `sitemap.json` is the recommended approach for including resources with a theme, you can also export your site's data in a LAR (Liferay Archive) file. A LAR file is version-specific; it won't work on any version of Liferay DXP other than the one from which it was exported. This approach does, however, require less configuration, since it does not require a sitemap or other files. So, if you're using the exported resources in the same version of Liferay DXP and it's not for a theme on Liferay Marketplace, you may prefer a LAR file.

To create an `archive.lar`, export the contents of a site using the site scope. Then place the `archive.lar` file in your theme's `[theme-name]/src/WEB-INF/src/resources-importer` folder. A LAR archive does not require a `sitemap.json` file or any other files in your `[theme-name]/src/WEB-INF/src/resources-importer` folder.

Related Topics

CHAPTER 78

UPGRADING YOUR THEME FROM LIFERAY PORTAL 6.2 TO 7.1

This section 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 4 and Lexicon 2.0.

Theme upgrades involve these steps:

- Updating project metadata
- Updating CSS
- Updating theme templates
- Updating resources importer configuration and content
- Applying Clay design patterns

As an example, these tutorials apply the steps to a theme called the Lunar Resort theme developed in the Liferay Portal 6.2 Developing a Liferay Theme Learning Path. It's similar to many Liferay Portal 6.2 themes, as it extends the `_styled` theme, adds configurable settings, and incorporats 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. 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 features, such as Themelets.

If your theme uses Bootstrap 3 and Lexicon CSS, you can still use Bootstrap 3 and Lexicon CSS alongside Bootstrap 4 and Clay CSS markup, thanks to Liferay DXP's compatibility layer.

Note: The compatibility layer is meant as a short-term solution to ensure that your Bootstrap 3 and Lexicon CSS components aren't broken while you update your theme to use Bootstrap 4 and Clay CSS. It will be disabled in a future release. Migrate your theme to use Bootstrap 4 and Clay CSS as soon as you're able to.

Follow the steps in the Running the 6.2 theme upgrade task tutorial to learn how to migrate your theme to use the Liferay JS Theme Toolkit, including its Gulp `upgrade` task. Otherwise, you

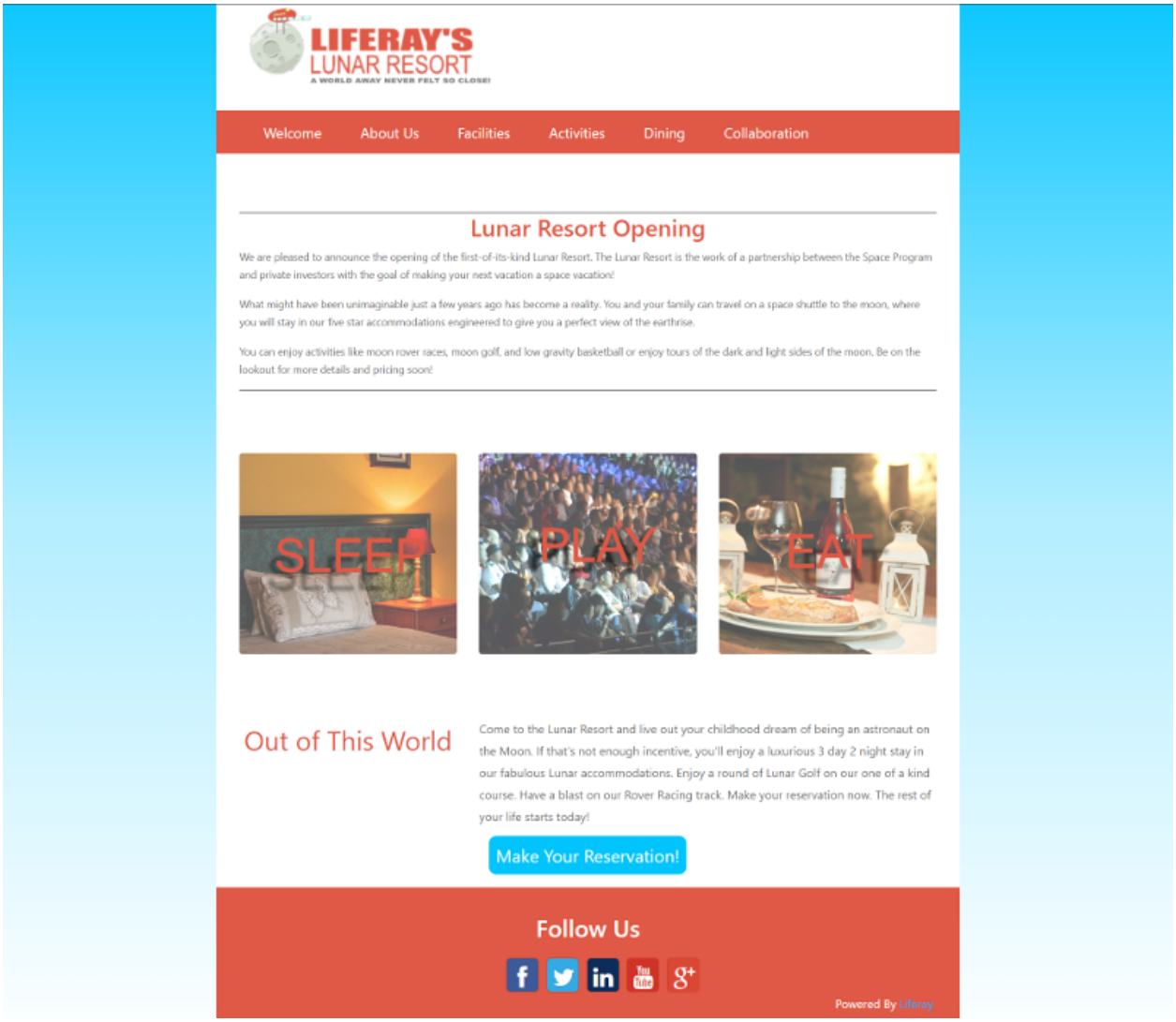


Figure 78.1: The Lunar Resort example theme upgraded in this tutorial uses a clean, minimal design.

must follow the steps in the remaining tutorials in this section to upgrade your 6.2 theme to 7.1 manually.

78.1 Running the Gulp Upgrade Task for 6.2 Themes

You can upgrade Liferay Portal 6.2 theme 7.0, regardless of the development environment (Plugins SDK, Maven, etc.) you used. If you migrate your theme to use the Liferay JS Theme Toolkit, you can leverage the theme's Gulp `upgrade` task.

Without the Liferay JS Theme Toolkit, you must follow the directions in the remaining tutorials in this section to upgrade your theme manually.

Here's what the Upgrade Task does:

- Updates the theme's Liferay version

- Updates the theme's Bootstrap version
- Updates the theme's Lexicon version
- Suggests specific code updates

Here are the steps for using the Gulp `upgrade` task:

1. Migrate your 6.2 theme to use the Liferay JS Theme Toolkit. Note that **you must** have the Liferay Theme Generator installed to migrate your theme to use the Liferay JS Theme Toolkit:

   ```
   yo liferay-theme:import
   ```

   You must provide the absolute path to your 6.2 theme's root folder. The import task does not work for relative paths.

2. Navigate to your theme's root directory and run the command below to initially upgrade your 6.2 theme to 7.0:

   ```
   gulp upgrade
   ```

   Here's what the 6.2 to 7.0 upgrade task does:

   - Updates the theme's Liferay version
   - Updates the CSS
   - Suggests specific code updates

   The task continues upgrading CSS files, prompting you to update CSS file names. 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.

```
**Note**: The `gulp upgrade` task overwrites the theme's files. We recommend
that you backup your theme's files before running it.
```

```
The upgrade task automatically upgrades CSS code that it can identify. For
everything else, it suggests upgrades.
```

3. Run the `gulp upgrade` command again to upgrade the 7.0 theme to 7.1.

   Here's what it does:

   - Creates core code for generating theme base files
   - Collects removed Bootstrap and Lexicon variables
   - Updates Bootstrap version references
   - Updates Lexicon version references
   - Updates Liferay version references

The Gulp `upgrade` task lists any deprecated or removed variables. For other areas of the code it suspects might need updates, it logs suggestions. The task also reports changes that may affect theme templates.

The Gulp `upgrade` task jump-starts the upgrade process, but it doesn't complete it. Manual updates are required.

The rest of the tutorials in this section explain all the 6.2 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.

Related Topics

Creating Themes
    Updating 6.2 Project Metadata

78.2 Updating 6.2 Project Metadata

If your theme uses the Liferay JS Theme Toolkit, the Gulp `upgrade` task automatically updates some of your theme's metadata as part of the upgrade process. Follow the steps below to update your theme's metadata manually:

1. Open your theme's `liferay-plugin-package.properties` file and change the `liferay-versions` property value to `7.1.0+`:

    ```
    liferay-versions=7.1.0+
    ```

2. Open the `liferay-look-and-feel.xml` file and specify `7.1.0+` as the compatibility version:

    ```
    <look-and-feel>
            <compatibility>
                    <version>7.1.0+</version>
            </compatibility>

            ...

    </look-and-feel>
    ```

3. While you're updating `liferay-look-and-feel.xml`, enable your theme to use Portlet Decorators.

4. If your theme uses the Liferay JS Theme Toolkit, the Gulp `upgrade` task updates the `package.json` file's Liferay version references to `7.1`, and it updates the `liferayTheme`'s `templateLanguage` to `ftl` (since Velocity theme templates are no longer supported):

    ```
    "liferayTheme": {
        ...
        "templateLanguage": "ftl",
        "version": "7.1"
    },
    ```

    Your theme's Liferay version references are updated for 7.0.

Related Topics

Updating 6.2 CSS Code
    Developing Themes

CHAPTER 79

UPDATING 6.2 CSS CODE

7.0's UI improvements required these CSS-related changes:

- Renaming CSS files
- Removing unneeded CSS files
- Updating CSS rules and imports
- Modifications to CSS responsiveness tokens

The theme upgrade process involves conforming to these changes. Here's how to update your theme's CSS to leverage the styling improvements.

79.1 Updating CSS File Names for Clay and 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`, and `aui` filename prefixes have been replaced with `clay` to reflect the introduction of Clay (previously Lexicon CSS).

Note: The Gulp `upgrade` task renames CSS files automatically.

Follow these steps to update your CSS file names manually:

1. Change each CSS file name's suffix in your theme from `.css` to `.scss`, then prepend an underscore (_) to all Sass partial file names, except `main.scss` and `aui.scss`. The Lunar Resort's updated files are shown below:

```
- `css/`
  - `_aui_variables.scss`
  - `_custom.scss`
```

2. Rename any CSS files that use the name `aui` in your theme to use the name `clay` instead. Below are the Lunar Resort's updated CSS file names:

```
- `css/`
  - `_clay_variables.scss`
  - `_custom.scss`
```

The full list of core files to rename is shown below for reference. If you modified a CSS file shown in the table below, **you must** update its name. If you didn't modify the file, no action is required:

CSS File Name Updates

Original CSS File Name	Updated CSS File Name
application.css	_application.scss
aui.css	clay.scss
base.css	_base.scss
custom.css	_custom.scss
dockbar.css	removed
extras.css	_extras.scss
layout.css	_layout.scss
main.css	main.scss
navigation.css	_navigation.scss
portal.css	_portal.scss
portlet.css	_portlet.scss
taglib.css	_taglib.scss

Related Topics

Running the Gulp Upgrade Task for 6.2 Themes
Updating 6.2 CSS Rules

79.2 Updating 6.2 CSS Rules and Imports

7.0 uses Bootstrap 4's CSS rule syntax. Font Awesome icons have also been moved, requiring changes to your imports. If your theme uses the Liferay JS Theme Toolkit, the Gulp `upgrade` task reports automatic CSS updates and suggests manual updates. For example, here is part of the task log for the Lunar Resort theme upgrade from 6.2 to 7.0. For each update performed and suggested, the task reports a file name and line number range:

```
-----------------------------------------------------------------
 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"
    ...
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"
    ...
```

Manually Updating CSS Rules

Follow these steps to update your theme's CSS rules manually:

1. Since Bootstrap 3 adopted 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. 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 for the Lunar Resort theme's `_custom.scss` file:

   Original:

```
#reserveBtn {
    background-color: #00C4FB;
    border-radius: 10px;
    color: #FFF;
    font-size: 1.5em;
    height: 62px;
    margin: 30px;
    padding: 10px 0;
    ...
}
```

Updated:

```
#reserveBtn {
    background-color: #00C4FB;
    border-radius: 10px;
    color: #FFF;
    font-size: 1.5em;
    height: 82px;
    margin: 30px;
    padding: 10px 0;
    ...
}
```

```
**Note:** For individual elements, you can overwrite the
`box-sizing:border-box` rule with `box-sizing:content-box`.
```

---

2. The following variables are removed in Bootstrap 4. Remove any of these if they are used in your theme:

```
$line-height-computed
$padding-base-horizontal
$padding-base-vertical
$padding-large-horizontal
$padding-large-vertical
$padding-small-horizontal
$padding-small-vertical
$padding-xs-horizontal
$padding-xs-vertical
$gray-base
$gray-darker
$gray-dark
$gray
$gray-light
$gray-lighter
$brand-primary
$brand-success
$brand-info
$brand-warning
$brand-danger
$state-success-text
$state-success-bg
$state-success-border
$state-info-text
$state-info-bg
$state-info-border
$state-warning-text
$state-warning-bg
$state-warning-border
```

```
$state-danger-text
$state-danger-bg
$state-danger-border
```

See the Migrating from 2.x to 3.0 guide for CSS rules that changed in Bootstrap 3. Likewise, you can refer to the Migrating to v4 guide for updating CSS rules to Bootstrap 4.

Updating Font Awesome Imports

Font Awesome icons were moved to their own file (`font-awesome.scss`) to avoid the IE9 CSS selector limitation. If you include these imports in your `_custom.scss` file, you must remove them:

```
@import "aui/alloy-font-awesome/scss/mixins-alloy";
@import "aui/alloy-font-awesome/scss/variables";
```

Related Topics

Updating 6.2 CSS Responsiveness
Copying an Existing Theme's Files

79.3 Updating the Responsiveness

Bootstrap 4 explicit media queries replaced the Bootstrap 2 `respond-to` mixins for CSS responsiveness. Follow these steps to update your theme's responsiveness:

1. Open your `_custom.scss` file.
2. Replace all `respond-to` mixins with corresponding media queries shown below. Note that some of the dimensions are slightly different:

   Media Query Replacements

Liferay Portal 6.2 Mixin	7.0 Media Query
`@include respond-to(phone)`	(max-width: 767px)
`@include media-breakpoint-down(sm)`	(max-width: 767px)
`@include respond-to(tablet)`	(min-width: 768px, max-width: 979px)
`@include media-breakpoint-only(md)`	(min-width: 768px, max-width: 991px)
`@include respond-to(phone, tablet)`	(max-width: 979px)
`@include media-breakpoint-down(md)`	(max-width: 991px)
`@include respond-to(desktop, tablet)`	(min-width: 768px)
`@include media-breakpoint-up(md)`	(min-width: 768px)
`@include respond-to(desktop)`	(min-width: 980px)
`@include media-breakpoint-up(lg)`	(min-width: 992px)

For example, here is a comparison between the Lunar Resort theme's original and updated syntax:

Original:

```
@include respond-to(phone, tablet) {
    html #wrapper #banner #navigation {
    …
    }
    …
}
```

Updated:

```
@include media-breakpoint-down(md) {
    html #wrapper #banner #navigation {
    ...
    }
    ...
}
```

Related Topics

Updating 6.2 Theme Templates
    Updating CSS File Names for Clay and Sass

79.4 Updating 6.2 Theme Templates

7.0 theme templates are essentially the same as Liferay Portal 6.2 theme templates. Here are the main changes:

- Velocity templates were deprecated in Liferay Portal CE 7.0 and are now removed in favor of FreeMarker templates in Liferay DXP. 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.

- The Dockbar has been replaced and reorganized into a set of three distinct 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.

Start by converting your Velocity theme templates to FreeMarker. You can refer to Apache's FreeMarker documentation for help. Common Liferay DXP FreeMarker variables and macros can be found in `FTL_liferay.ftl`

If you used the Gulp `upgrade` task, it reports the required theme template changes in the log. For example, here are the 6.2 to 7.0 upgrade log and 7.0 to 7.1 upgrade log for the Lunar Resort theme:

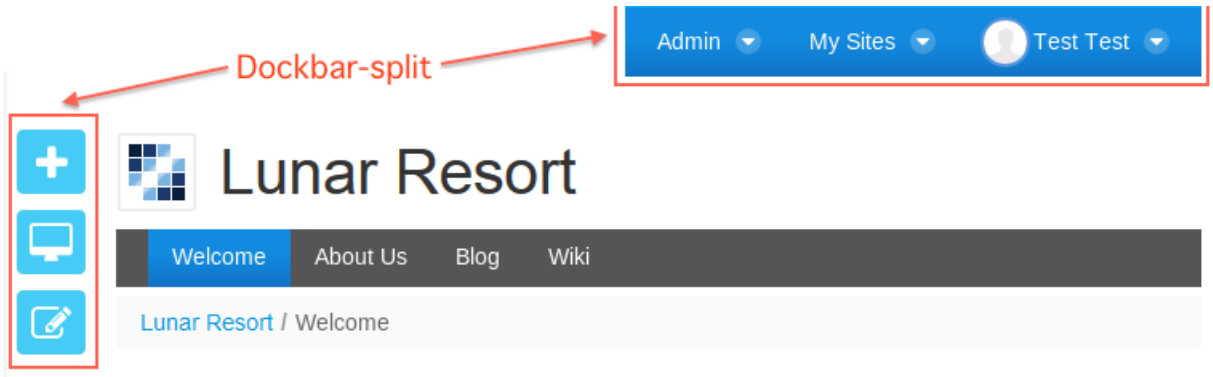


Figure 79.1: The Dockbar was removed in 7.0 and must be replaced with the new Control Menu.

```
------------------------------------------------------------------
 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.
[18:57:23] Finished 'upgrade:log-changes' after 5.61 ms
[18:57:23] Finished 'upgrade' after 19 s

------------------------------------------------------------------
 Liferay Upgrade (7.0 to 7.1)
------------------------------------------------------------------

Renamed aui.scss to clay.scss
[19:16:54] Finished 'upgrade:log-changes' after 2.53 ms
[19:16:54] Finished 'upgrade' after 16 min
```

The log warns about removed and deprecated code and suggests replacements when applicable.

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

If you didn't customize `portal_normal.ftl`, you can skip this section. Follow these steps to update `portal_normal.ftl`:

1. Open your modified `portal_normal.ftl` file and replace the following 6.2 directives with the updated syntax. This change is described in the 7.0 Breaking Changes reference document:

6.2	Updated
`${theme.include(top_head_include)}`	`<@liferay_util["include"] page=top_head_include />`
`${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 /> </@>`

6.2	Updated
${theme.include(body_bottom_include)}	<@liferay_util["include"] page=body_bottom_include />
${theme.include(bottom_include)}	<@liferay_util["include"] page=bottom_include />
${theme_settings["my-theme-setting"]}	${themeDisplay.getThemeSetting("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. Optionally remove the breadcrumbs and page title code:

```
<nav id="breadcrumbs">
    <@liferay.breadcrumbs />
</nav>
...
<h2 class="page-title">
    <span>${the_title}</span>
</h2>
```

3. If you used the split Dockbar in your Liferay Portal 6.2 theme, remove `dockbar-split` from the `body` element's `class` value so it matches the markup below:

```
<body class="${css_class}">
```

4. Find the `<a href="#main-content" id="skip-to-content"><@liferay.language key="skip-to-content" /></a>` element and replace it with the updated Liferay UI quick access macro shown below:

```
<@liferay_ui["quick-access"] contentId="#main-content" />
```

5. Replace the `<@liferay.dockbar />` macro with the updated Control menu macro:

```
<@liferay.control_menu />
```

6. Add the `<#if...></#if>` wrappers to the `navigation.ftl` theme template include:

```
<#if has_navigation && is_setup_complete>
    <#include "${full_templates_path}/navigation.ftl" />
</#if>
```

7. Replace the `content` `<div>` with an HTML 5 `section` element. The `section` element is more accurate and provides better accessibility for screen readers:

```
<section id="content">
```

8. Add the `<h1 class="hide-accessible">${the_title}</h1>` header element just inside the `content` `<section>` to support accessibility.

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() />
```

3. 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. You can update `portlet.ftl` next.

Updating Portlet FTL

Follow these steps to update your modified `portlet.ftl` file:

1. Find the `<a class="icon-monospaced portlet-icon-back text-default" href="${portlet_back_url}" title="<@liferay.language key="return-to-full-page" />">` element and add the `list-unstyled` class to it:

```
<a
  class="icon-monospaced list-unstyled portlet-icon-back text-default"
  href="${portlet_back_url}"
  title="<@liferay.language key="return-to-full-page" />"
>
```

2. Find the `<div class="autofit-float autofit-row">` element and add the `portlet-header` class to it:

```
<div class="autofit-float autofit-row portlet-header">
```

The portlet template is updated. You can update `init_custom.ftl` next.

Updating Init Custom FTL

If your theme uses configurable theme settings, update them to use the new syntax, following the patterns below.

Original syntax:

```
<#assign theme_setting_variable =
getterUtil.getBoolean(theme_settings["theme-setting-key"])>

<#assign theme_setting_variable =
getterUtil.getString(theme_settings["theme-setting-key"])>
```

Updated syntax:

```
<#assign theme_setting_variable =
getterUtil.getBoolean(themeDisplay.getThemeSetting("theme-setting-key"))/>

<#assign theme_setting_variable =
themeDisplay.getThemeSetting("theme-setting-key")/>
```

For example, here are the Lunar Resort theme's updated theme settings:

```
<#assign show_breadcrumbs =
getterUtil.getBoolean(themeDisplay.getThemeSetting("show-breadcrumbs"))/>

<#assign show_page_title =
getterUtil.getBoolean(themeDisplay.getThemeSetting("show-page-title"))/>
```

That covers most, if not all, of the theme template changes. If you modified any other FreeMarker theme templates, compare them with templates in the `_unstyled` theme. If your theme uses the Liferay JS Theme Toolkit, refer to the suggested changes that the Gulp `upgrade` task reports.

Related Topics

Updating CSS Code

Making Configurable Theme Settings

79.5 Updating the Resources Importer

The Resources Importer is now an OSGi module in Liferay's Web Experience application suite. Since the suite is bundled with Liferay DXP, you don't have to download the Resources Importer separately. The following components have been updated and are the focus of this tutorial:

- Plugin properties
- Web content article files and folder structure
- Sitemap

Note: Due to the page and article import order, articles that link to pages in the Site's layout cause a null pointer exception issue. These links have been removed from the example Lunar Resort theme's web content articles to avoid this issue.

Start updating the plugin properties for the Resources Importer.

Updating liferay-plugin-package.properties

Follow the steps in this section to upgrade Plugins SDK themes. Skip to the next section for all other themes.

1. Open the `src\WEB-INF\liferay-plugin-package.properties` file and remove the `required-deployment-contexts` property. This is no longer needed since the Resources Importer is bundled with Liferay DXP.

2. The group model class's fully-qualified class name has changed. Replace the `resources-importer-target-class-name` property's value with the updated one below:

   ```
   com.liferay.portal.kernel.model.Group
   ```

Now that your `liferay-plugin-package.properties` is updated, you can update your theme's web content.

Updating Web Content

All web content articles must be written in XML and have a structure for article creation and a template for rendering.

Note: The example Lunar Resort theme's updated XML articles are in the ZIP file's `/resources-importer/journal/articles/Basic Web Content/` folder for reference.

Follow these steps to update your web content:

1. Create a subfolder, for example `BASIC_WEB_CONTENT`, in the `/resources-importer/journal/articles/` folder, and move all the basic HTML articles (articles that did not require a structure or template previously) into it.

2. Create a subfolder in the `/resources-importer/journal/templates/` folder with the same name as the folder you created in step 1. The articles and template folder names must match for the web content to import properly.

3. XML article structures are now written in JSON. Create a file, for example `BASIC_WEB_CONTENT.json`, in the `/resources-importer/journal/structures/` folder. The structure name **must match** the folder names you created in the previous steps. To ensure the syntax is correct for web content articles that used a structure and template before, we recommend that you recreate the structure and template in Liferay DXP.

4. In the JSON file you just created, add a JSON structure for the basic web content that follows the pattern below:

   ```
   {
       "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"
        }
    ]
}
```

5. For basic web content, create a FreeMarker template file (e.g., `[template-folder-name].ftl`) in the template subfolder you created in step 2, and add the method below to retrieve the article's data:

```
${content.getData()}
```

6. Use this syntax to migrate basic web content articles from HTML to XML. **Remember** to change your `.html` file extension 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[
                        ORIGINAL HTML CONTENT GOES HERE
                        ]]>
                </dynamic-content>
        </dynamic-element>
</root>
```

7. 7.0's updated Bootstrap requires that you replace all `span[number]` classes with the updated syntax below:

```
col-[device-size]-[number]
```

   `[device-size]` can be `xs`, `sm`, `md`, or `lg`. `md` works for most cases. The original and updated classes for the Lunar Resort's `2 column description.xml` article are shown below for reference:

   Original:

```
<?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...
                    </div>
                </div>
            ]]>
        </dynamic-content>
    </dynamic-element>
</root>
```

Updated:

```
<?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...
                    </div>
                </div>
            ]]>
        </dynamic-content>
    </dynamic-element>
</root>
```

Bootstrap's documentation explains the updated grid system.

Next, you must update your theme's sitemap file.

Updating the Sitemap

In Liferay Portal 6.2, portlet IDs were incremental numbers. In 7.0, they're explicit class names. Update your `sitemap.json` file with the new portlet IDs.

Some common portlet IDs are specified in the `sitemap.json` example in the Creating a Sitemap for the Resources Importer tutorial.

You can also retrieve a portlet's ID from the UI:

1. In the portlet's *Options* menu, select *Look and Feel Configuration*.
2. Select the *Advanced Styling* tab.

   The `Portlet ID` value appears in the blue box.

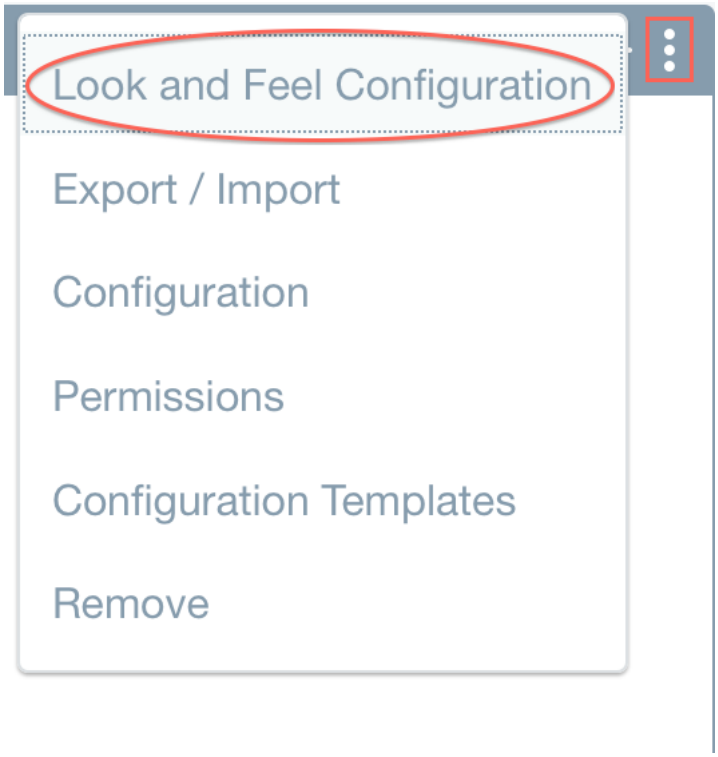


Figure 79.2: You can find the portlet ID in the *Look and Feel Configuration* menu.

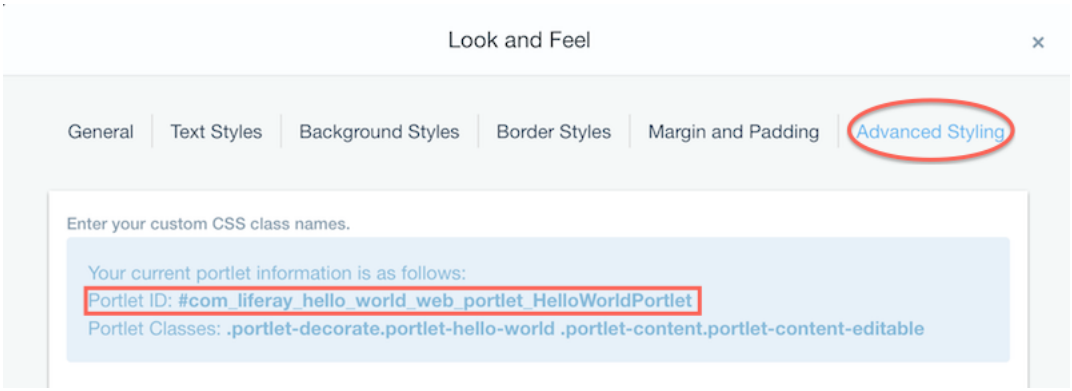


Figure 79.3: The portlet ID appears within the blue box in the *Advanced Styling* tab.

The Portlet ID Quick Reference Guide shows all the default portlet IDs. Check `liferay-portlet.xml` for the portlet ID number in 6.2 and replace it with the updated ID in the quick reference Guide.

Remember to use the updated `.xml` extension for your web content articles in your sitemap.

Related Topics

Updating 6.2 CSS Code

Applying Clay Design Patterns to 6.2 Themes

79.6 Applying Clay Design Patterns

7.0 uses Clay, a web implementation of Liferay's Lexicon Experience Language. The Lexicon Experience Language provides styling guidelines and best practices for application UIs. Clay's CSS, HTML, and JavaScript components enable developers to build fully-realized UIs quickly and effectively. Liferay DXP's compatibility layer let's you use Lexicon CSS markup alongside Clay CSS.

Note: The compatibility layer is meant as a short-term solution to ensure that your Bootstrap 3 and Lexicon CSS components aren't broken while you update your theme to use Bootstrap 4 and Clay CSS. It will be disabled in a future release. Migrate your theme to use Bootstrap 4 and Clay CSS as soon as you're able to.

This section demonstrates how to apply Clay to your HTML markup. 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. Here's the Lunar Resort form's updated Clay markup:

```
<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>
```

The Clay 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.

You can apply similar Clay patterns to your theme's HTML files.

You've updated your theme to 7.0! You can deploy it from your theme project. Now your users can continue enjoying the visual styles you've created in your upgraded themes.

Related Topics

Liferay Theme Generator

Upgrading to 7.0

CHAPTER 80

Upgrading Your Theme from Liferay Portal 7.0 to 7.1

This section of tutorials guides you through the process of upgrading your 7.0 theme to run on 7.0. While you're at it, you should leverage theme improvements, including support for Bootstrap 4 and Lexicon 2.0.

Theme upgrades involve these steps:

- Updating project metadata
- Updating CSS
- Updating theme templates

No matter the environment in which you're developing your theme, these tutorials explain everything required to upgrade it. The easiest option is to use the gulp `upgrade` task. This, however, is only available for themes created with the Liferay Theme Generator. If you're upgrading in an environment other than the themes generator, follow the other tutorials in this section to upgrade your theme manually.

80.1 Upgrading Themes Created With the Liferay Theme Generator

A Liferay Portal 7.0 theme can be upgraded to 7.0, regardless of its project environment (Liferay Theme Generator, Plugins SDK, Maven, etc.). But a theme created with the Liferay Theme Generator can leverage the theme's `gulp upgrade` task. If you're developing your theme in an environment other than the Themes Generator, the `gulp upgrade` task doesn't work for your theme. Please follow the manual directions in the remaining tutorials in this section to upgrade your theme manually.

Here's what the Upgrade Task does:

- Updates the theme's Liferay version
- Updates the theme's Bootstrap version
- Updates the theme's Lexicon version
- Suggests specific code updates

Here are the steps for using the theme Gulp `upgrade` task:

1. Navigate to your theme's root directory.
2. Update the path of your Liferay DXP server in your theme's `liferay-theme.json` file to point to your 7.1 Liferay DXP server. You can use the `gulp init` task to update the path to your server.
3. Run the command below to update liferay-theme-tasks to the latest version:

   ```
   npm update liferay-theme-tasks
   ```

4. Run the command below to upgrade the theme:

   ```
   gulp upgrade
   ```

   Here's what it does:

   - Copies the existing theme to a folder called `_backup`
   - Creates core code for generating theme base files
   - Collects removed Bootstrap and Lexicon variables
   - Updates Bootstrap version references
   - Updates Lexicon version references
   - Updates Liferay version references

Note: The `gulp upgrade` task overwrites the theme's files. We recommend that you backup your theme's files before running it.

The Gulp task list any deprecated or removed variables. For other areas of the code it suspects might need updates, it logs suggestions. The task also reports changes that may affect theme templates.

The Gulp `upgrade` task jump-starts the upgrade process, but it doesn't complete it. Manual updates are required.

The rest of the tutorials in this section explain 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.

Related Topics

Creating Themes

Updating Project Metadata

80.2 Updating Project Metadata

If your theme uses the Liferay Theme Generator, the gulp upgrade task automatically updates your theme's metadata as part of the upgrade process. If you're developing your theme in an environment other than the themes generator, follow the steps below to update your theme's metadata manually:

1. Open your theme's `liferay-plugin-package.properties` file and change the `liferay-versions` property value to `7.1.0+`:

```
liferay-versions=7.1.0+
```

2. Open the `liferay-look-and-feel.xml` file and specify `7.1.0+` as the compatibility version:

```
<look-and-feel>
        <compatibility>
                <version>7.1.0+</version>
        </compatibility>

        ...

</look-and-feel>
```

3. If your theme uses the Liferay Theme Generator, open the `package.json` file and update the file's Liferay version references to `7.1`. Update the `liferayTheme`'s `templateLanguage` to `ftl` (since Velocity theme templates are no longer supported), and update its `version` to `7.1`:

```
"liferayTheme": {
    ...
    "templateLanguage": "ftl",
    "version": "7.1"
},
```

4. Update the `liferay-theme-deps-7-0` dependency to `liferay-theme-deps-7.1` with the version below, and add the `liferay-theme-tasks` dependency as shown in the example configuration below:

```
"devDependencies": {
    "gulp": "3.9.1",
    "liferay-theme-tasks": "8.0.0-alpha.6",
    "liferay-theme-deps-7.1": "8.0.0-alpha.6"
},
```

Your theme's Liferay version references are updated for 7.0.

Related Topics

Updating CSS Code
    Developing Themes

80.3 Updating CSS Code

7.0's UI improvements required these CSS-related changes:

- Renaming CSS files
- Class variable changes
- Updating core imports

The theme upgrade process involves conforming to these changes. Now you'll update your theme's CSS files to reflect these changes. Start with updating CSS file names.

Updating CSS File Names for Clay

Some of the CSS filenames have changed to reflect the introduction of Clay (previously Lexicon CSS). The filename changes for the unstyled theme are listed below. Refer to the Theme Reference Guide for a complete list of expected theme CSS files.

The old 7.0 aui filenames are shown below:

- css/
    - _aui_custom.scss
    - _aui_variables.scss
    - aui.scss

Rename the aui files to match the updated 7.1 clay filenames shown below:

- css/
    - _clay_custom.scss
    - _clay_variables.scss
    - clay.scss

Next, you can note the removed and deprecated variables and mixins for Bootstrap 4 and Lexicon.

Class Variable Changes

7.0 uses Bootstrap 4's CSS rule syntax. The new syntax lets developers leverage Bootstrap 4 features and improvements. If your theme does not use the Liferay Theme Generator, you can refer to the Migrating to v4 guide for updating CSS rules to Bootstrap 4.

If your theme uses the Liferay Theme Generator, the `gulp upgrade` task suggests manual updates. For example, here is part of the task log for the 7.0 Westeros Bank theme:

```
---------------------------------------------------------------
 Lexicon Upgrade (1.0 to 2.0)
---------------------------------------------------------------

File: _variables_custom.scss
    $brand-default was deprecated in Lexicon CSS 1.x.x and has been removed
        in the new Clay 2.x.x version
```

The log lists removed and/or deprecated variables and suggests possible changes. For each update performed or suggested, the task reports a file name. For reference, here's the full list of variable changes:

The following variables were removed in Bootstrap 4:

```
$line-height-computed
$padding-base-horizontal
$padding-base-vertical
$padding-large-horizontal
$padding-large-vertical
$padding-small-horizontal
$padding-small-vertical
$padding-xs-horizontal
$padding-xs-vertical
$gray-base
```

```
$gray-darker
$gray-dark
$gray
$gray-light
$gray-lighter
$brand-primary
$brand-success
$brand-info
$brand-warning
$brand-danger
$state-success-text
$state-success-bg
$state-success-border
$state-info-text
$state-info-bg
$state-info-border
$state-warning-text
$state-warning-bg
$state-warning-border
$state-danger-text
$state-danger-bg
$state-danger-border
```

See Bootstrap Migration Guide for a full list of the changes.

The following Lexicon variables were deprecated in Lexicon CSS 1.x.x and are removed in the new Clay 2.x.x version:

```
$atlas-theme
$box-shadow-default
$box-shadow-default-bg
$box-shadow-default-blur
$box-shadow-default-spread
$box-shadow-default-x
$box-shadow-default-y
$brand-danger-active
$brand-danger-hover
$brand-default
$brand-default-active
$brand-default-hover
$brand-info-active
$brand-info-hover
$brand-primary-active
$brand-primary-hover
$brand-success-active
$brand-success-hover
$brand-warning-active
$brand-warning-hover
$inverse-active-bg
$inverse-active-border
$inverse-active-color
$inverse-bg
$inverse-border
$inverse-color
$inverse-disabled-color
$inverse-header-bg
$inverse-header-border
$inverse-header-color
$inverse-hover-bg
$inverse-hover-border
$inverse-hover-color
$inverse-link-color
$inverse-link-hover-color
$state-danger-bg
$state-danger-border
$state-danger-text
$state-default-bg
```

```
$state-default-border
$state-default-text
$state-info-bg
$state-info-border
$state-info-text
$state-primary-bg
$state-primary-border
$state-primary-text
$state-success-bg
$state-success-border
$state-success-text
$state-warning-bg
$state-warning-border
$state-warning-text
```

The following Lexicon mixins are removed in Clay 2.x.x:

`@mixin color-placeholder`:

Lexicon's `color-placeholder($element, $color: $input-color-placeholder)` mixin is deprecated as of v1.0.9 and was removed in v2.0.0. Please use `placeholder($color: $input-color-placeholder)` instead.

`@mixin select-box-icon`:

Lexicon's `select-box-icon($color)` mixin is deprecated as of v1.0.10 and was removed in v2.0.0. Please use `background-image: lx-icon($name, $color)` instead.

Note that if the `gulp upgrade` task detects any variables in your theme that have been removed in Clay from the previous LexiconCSS version, it adds the `_variables_deprecated.scss` file to your theme with the variables to make sure the theme compiles and to decouple it from future upgrades. If you wish to include this file manually, here are its contents for reference:

```
@warn "You're using deprecated variables. Please refer to the update guides
to remove its usage";

// Deprecated `$brand-*-*` as of v1.0.24, removed in v2.0.0
$brand-default: #869CAD !default; // Atlas
$brand-default-active: darken($brand-default, 12%) !default; // Atlas
$brand-default-hover: darken($brand-default, 8%) !default; // Atlas

$brand-primary: #65B6F0 !default;
$brand-primary-active: darken($brand-primary, 12%) !default; // Atlas
$brand-primary-hover: darken($brand-primary, 8%) !default; // Atlas

$brand-success: #76BD4A !default;
$brand-success-active: darken($brand-success, 12%) !default; // Atlas
$brand-success-hover: darken($brand-success, 8%) !default; // Atlas

$brand-info: #1E5EC8 !default;
$brand-info-active: darken($brand-info, 12%) !default; // Atlas
$brand-info-hover: darken($brand-info, 8%) !default; // Atlas

$brand-warning: #F5984C !default;
$brand-warning-active: darken($brand-warning, 12%) !default; // Atlas
$brand-warning-hover: darken($brand-warning, 8%) !default; // Atlas

$brand-danger: #C67 !default;
$brand-danger-active: darken($brand-danger, 12%) !default; // Atlas
$brand-danger-hover: darken($brand-danger, 8%) !default; // Atlas

// Box Shadow
// Defines global box-shadows

// Deprecated `$box-shadow-default-*` as of v1.0.24, removed in v2.0.0
$box-shadow-default-x: 0 !default; // Atlas
$box-shadow-default-y: 0 !default; // Atlas
```

```
$box-shadow-default-blur: 3px !default; // Atlas
$box-shadow-default-spread: 1px !default; // Atlas
$box-shadow-default-bg: rgba(0, 0, 0, 0.2) !default; // Atlas
$box-shadow-default: $box-shadow-default-x $box-shadow-default-y $box-shadow-default-blur $box-shadow-default-spread $box-
shadow-default-bg !default; // Atlas

// Inverse Colors

// Deprecated `$inverse-*` as of v1.0.24, removed in v2.0.0
$inverse-bg: #1F282E !default; // Atlas
$inverse-border: darken($inverse-bg, 10%) !default; // Atlas
$inverse-color: #869CAD !default; // Atlas

$inverse-active-bg: #3A4D5A !default; // Atlas
$inverse-active-border: darken($inverse-active-bg, 1%) !default; // Atlas
$inverse-active-color: #FFF !default; // Atlas

$inverse-disabled-color: #394956 !default; // Atlas

$inverse-hover-bg: #1C252C !default; // Atlas
$inverse-hover-border: darken($inverse-hover-bg, 1%) !default; // Atlas
$inverse-hover-color: $inverse-color !default; // Atlas

$inverse-header-bg: #1B2228 !default; // Atlas
$inverse-header-border: darken($inverse-header-bg, 1%) !default; // Atlas
$inverse-header-color: $inverse-color !default; // Atlas

$inverse-link-color: #FFF !default; // Atlas
$inverse-link-hover-color: $inverse-link-color !default; // Atlas

// States for Form and alert

// Deprecated `$state-*-*` as of v1.0.24, removed in v2.0.0
$state-default-text: $brand-default !default; // Atlas
$state-default-bg: lighten($state-default-text, 34%) !default; // Atlas
$state-default-border: $state-default-text !default; // Atlas

$state-primary-text: $brand-primary !default; // Atlas
$state-primary-bg: lighten($state-primary-text, 12%) !default; // Atlas
$state-primary-border: $state-primary-text !default; // Atlas

$state-success-text: $brand-success !default;

$state-success-bg: lighten($state-success-text, 34%) !default;
$state-success-border: $state-success-text !default;

$state-info-text: $brand-info !default;

$state-info-bg: lighten($state-info-text, 34%) !default;
$state-info-border: $state-info-text !default;

$state-warning-text: $brand-warning !default;

$state-warning-bg: #F8F4D5 !default;
$state-warning-border: $state-warning-text !default;

$state-danger-text: $brand-danger !default;

$state-danger-bg: lighten($state-danger-text, 34%) !default;
$state-danger-border: $state-danger-text !default;
```

After updating your theme's CSS variables and mixins, you should update the Font Awesome Icon imports.

Updating Font Awesome Icon Imports

Originally in Liferay Portal CE 7.0 and Liferay DXP, Font Awesome icons were imported in `_aui_variables.scss` (now renamed `_clay_variables.scss`). Font Awesome icons were later moved to the compatibility layer. If your 7.0 theme was made prior to this move and you modified `_aui_variables.scss`, you must remove the Font Awesome imports shown below:

```
// 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 can update the core imports.

Updating Core Imports

Update the old aui lexicon paths to use the clay paths instead, as shown in the table below:

Pattern	Replacement
`@import "/aui/lexicon/bootstrap/mixins/";`	removed
`@import "/aui/lexicon/lexicon-base/mixins/";`	removed
`@import "/aui/lexicon/atlas-theme/mixins/";`	removed
`@import "aui/lexicon/atlas-variables";`	`@import "clay/atlas-variables";`
`@import "aui/lexicon/atlas";`	`@import "clay/atlas";`

Related Topics

Updating Theme Templates
Copying an Existing Theme's Files

80.4 Updating Theme Templates

7.0 theme templates are essentially the same as Liferay Portal 7.0 theme templates. Here are the main changes:

- Velocity templates were deprecated in Liferay Portal CE 7.0 and are now removed in favor of FreeMarker templates in Liferay DXP.

Key reasons for using FreeMarker templates and removing 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.

You should start by addressing the Velocity theme templates. Since Velocity theme templates are no longer supported, **you must convert your Velocity theme templates to FreeMarker**.

If you're using the Liferay Theme Generator, the `gulp upgrade` command reports the required theme template changes in the log.

For example, here is the `gulp upgrade` log for the Westeros Bank theme:

```
---------------------------------------------------------------
 Liferay Upgrade (7.0 to 7.1)
---------------------------------------------------------------

Renamed aui.scss to clay.scss
File: footer.ftl
    Warning: .container-fluid-1280 has been deprecated. Please use
    .container-fluid.container-fluid-max-xl instead.
File: portal_normal.ftl
    Warning: .navbar-header has been removed. This container should be
    removed in most cases. Please, use your own container if necessary.
```

The log warns about removed and deprecated code and suggests replacements when applicable. For reference, the main changes appear below:

- List items inside a container with the `list-inline` class now require the `list-inline-item` class.
- The `container-fluid-1280` class has been deprecated. Please use `container-fluid container-fluid-max-xl` instead.
- Responsive navbar behaviors are now applied to the `navbar` class via the required `navbar-expand-{breakpoint}` class.
- The `navbar-toggle` class is now `navbar-toggler` and has different inner markup.
- The `navbar-header` class has been removed. This container should be removed in most cases. Please, use your own container if necessary.

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. Follow the steps below to update `portal_normal.ftl`:

1. Open your modified `portal_normal.ftl` file and remove the breadcrumbs:

```
<nav id="breadcrumbs">
    <@liferay.breadcrumbs />
</nav>
```

2. Remove `id="main-surface"` from the `body` tag. This is not needed for SPA to work properly:

```
<body class="${css_class}" id="main-surface">
```

If you modified the portlet template for your theme, follow the steps in the next section.

Updating Portlet FTL

Follow these steps to update your modified `portlet.ftl` file:

1. Find the `<a class="icon-monospaced portlet-icon-back text-default" href="${portlet_back_url}" title="<@liferay.language key="return-to-full-page" />">` element and add the `list-unstyled` class to it:

```
<a
  class="icon-monospaced list-unstyled portlet-icon-back text-default"
  href="${portlet_back_url}"
  title="<@liferay.language key="return-to-full-page" />"
>
```

2. Find the `<div class="autofit-float autofit-row">` element and add the `portlet-header` class to it:

```
<div class="autofit-float autofit-row portlet-header">
```

The portlet 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. If your theme uses the Liferay Theme Generator, refer to the suggested changes that the `gulp upgrade` task reports.

Related Topics

Updating CSS Code

Making Configurable Theme Settings

80.5 Using the Bootstrap 3 Lexicon CSS Compatibility Layer

By default, Liferay DXP includes Bootstrap 4 out-of-the-box. Bootstrap 4 has been completely rewritten and therefore includes some notable changes and compatibility updates that may be cause for concern if your theme uses Bootstrap 3 or Lexicon CSS. Not to worry though. To ensure that your upgrade runs smoothly, Liferay DXP includes a compatibility layer so you can use Bootstrap 3 markup and Lexicon CSS markup alongside the new Bootstrap 4 and Clay CSS. The bundled icon fonts (Font Awesome v3.2.1 and Glyphicons 3) were moved to the compatibility layer's `_components.scss` file as well. If your theme extends the Styled base theme, this compatibility layer is included by default.

Note: The compatibility layer is meant as a short-term solution to ensure that your Bootstrap 3 and Lexicon CSS components aren't broken while you update your theme to use Bootstrap 4 and Clay CSS. It will be disabled in a future release. Migrate your theme to use Bootstrap 4 and Clay CSS as soon as you're able to.

Follow these guidelines to update your markup:

1. See how your theme looks with the compatibility layer enabled (it's enabled by default).

2. Individually disable the component(s) in the compatibility layer that you don't need. These are listed in the `css/compat/_variables.scss` file. For convenience, the components are listed below:

```
// Compatibility layer components config

$compat-alerts: true !default;
$compat-basic_search: true !default;
$compat-breadcrumbs: true !default;
$compat-button_groups: true !default;
$compat-buttons: true !default;
$compat-cards: true !default;
$compat-component_animations: true !default;
$compat-dropdowns: true !default;
$compat-figures: true !default;
$compat-form_validation: true !default;
$compat-forms: true !default;
$compat-grid: true !default;
$compat-icons: true !default;
$compat-labels: true !default;
$compat-liferay: true !default;
$compat-list_groups: true !default;
$compat-management_bar: true !default;
$compat-modals: true !default;
$compat-nav_tabs: true !default;
$compat-navbar: true !default;
$compat-navs: true !default;
$compat-pager: true !default;
$compat-pagination: true !default;
$compat-panels: true !default;
$compat-progress_bars: true !default;
$compat-responsive_utilities: true !default;
$compat\noindent\hrulefill: true !default;
$compat-simple_flexbox_grid: true !default;
$compat-stickers: true !default;
$compat-tables: true !default;
$compat-toggle_card: true !default;
$compat-toggle_switch: true !default;
$compat-toolbar: true !default;
$compat-user_icons: true !default;
$compat-utilities: true !default;
```

   To disable a component, add the component you want to remove compatibility for to `/src/css/_clay_custom.scss` (create this file if it doesn't exist) and set its value to `false`. The example below removes compatibility for alerts and cards:

```
$compat-alerts: false !default;
$compat-cards: false !default;
```

```
**Note:** Some Liferay DXP components haven't been migrated to Bootstrap 4.
Disabling certain components might cause portions of the UI to break.
Therefore, after upgrading your markup, we recommend that you re-enable any
components you disable. Proceed with caution.
```

---

3. Update your markup to Bootstrap 4 and Clay CSS until you're satisfied with the result.

4. Re-enable any components you disabled in the compatibility layer by removing any components you set to false in `/src/css/_clay_custom.scss`. This ensures that Liferay DXP's UI isn't broken.

Now you know how to use the Bootstrap 3 and Lexicon CSS compatibility layer to provide a smooth transition during your theme upgrade.

Related Topics

Updating CSS Code
    Updating Project Metadata

CHAPTER 81

LAYOUT TEMPLATES

Layout templates define how content can be placed on a page. Liferay DXP includes several default layout templates out-of-the-box that you can use to organize content on your pages:

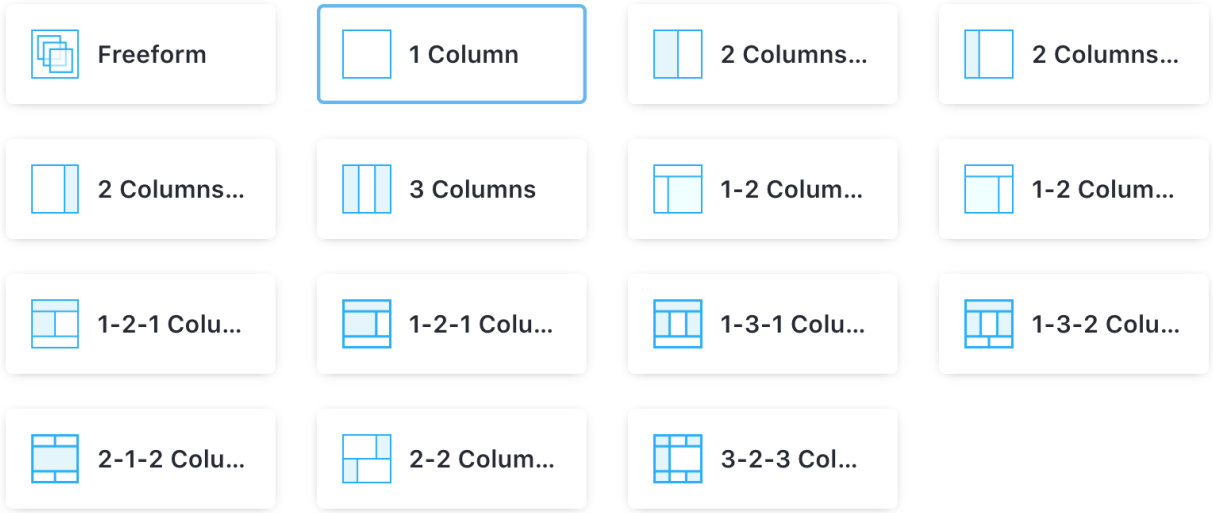


Figure 81.1: There are many default layout templates to choose from.

If you require a layout for your content that's not provided, you can create your own layout templates. In this section of tutorials, you'll learn how to develop layout templates for Liferay DXP.

81.1 Creating Layout Templates

Layout Templates specify how content is arranged on your site pages, as shown in the *1-2-1 Columns* layout below:

The Liferay Theme Generator provides a Layouts sub-generator that helps automate layout template creation. This tutorial covers how to use this tool to create layout templates. Install the

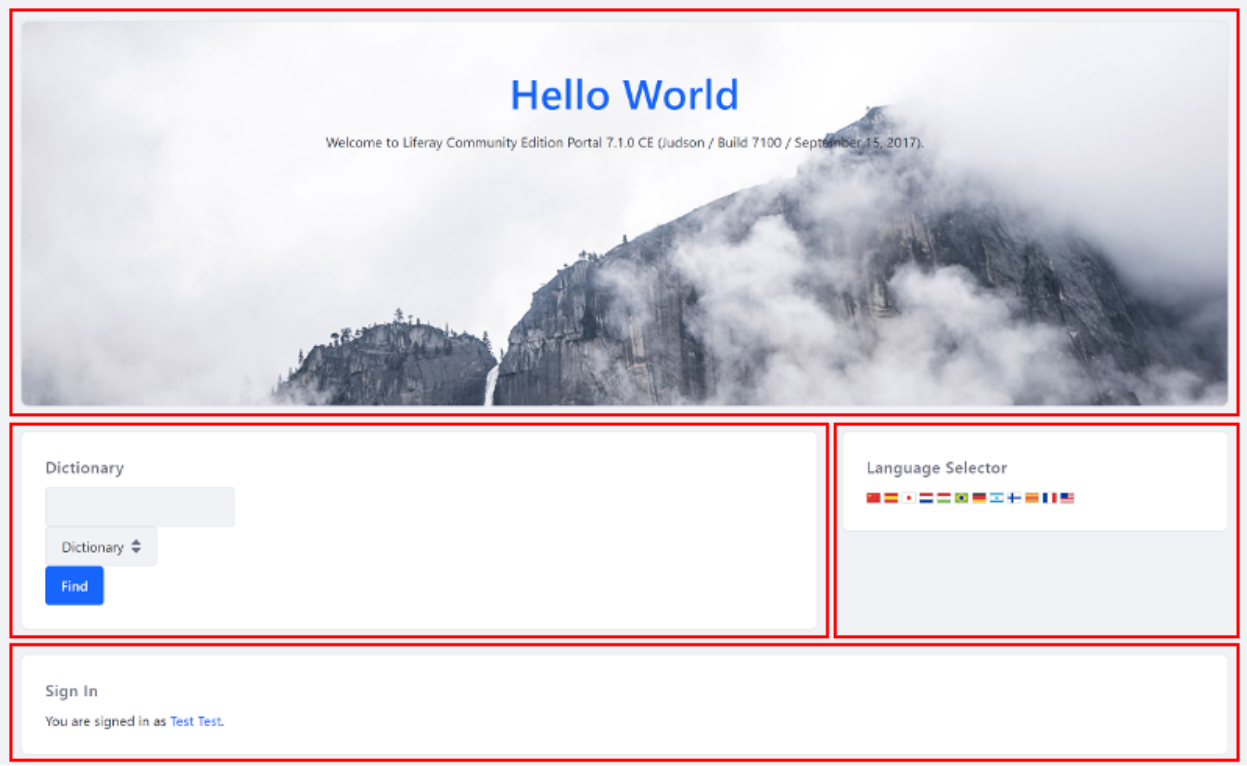


Figure 81.2: The *1-2-1 Columns* page layout creates a nice flow for your content.

Liferay Theme Generator if it's not already installed, then follow these steps to create a layout template:

1. Open the Command Line and navigate to the folder you want to create your layout template in.
2. Run The Layouts sub-generator and follow the prompts to create your layout:

```
`yo liferay-theme:layout`
```

```
**Note:** Run the Layouts sub-generator from the theme's root folder to
bundle it with the theme. This adds the layout template to the theme's
`src/layouttpl/custom` folder. This **only works** for generated themes.
```

```
Every row consists of 12 sections, so columns can range in size from 1 to 12.
Once you've entered a value, the generator asks the size you want your row
and column to be and presents you with the available width(s).

![ You must specify the width for each column in the row.](./images/layout-column-widths.png)

Choose from the available option(s) with your arrow keys and press Enter to
make your selection. Repeat this process for the remaining columns.
```

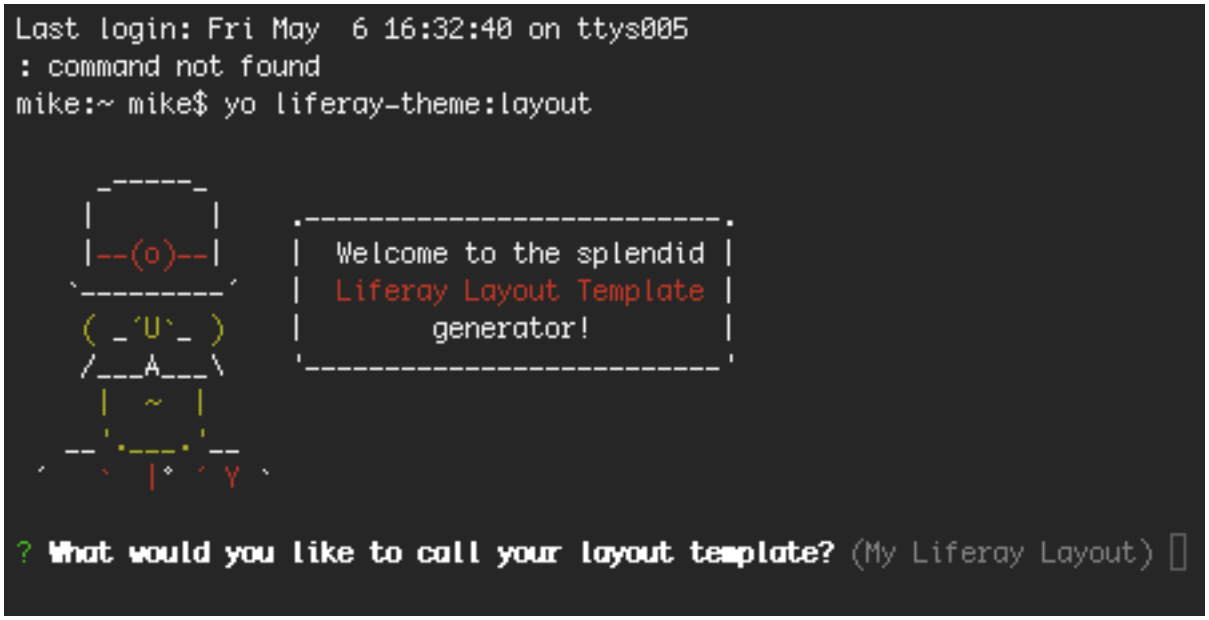


Figure 81.3: The Layouts sub-generator automates the layout creation process.

```
The Layouts sub-generator provides the following options for layout
templates:

- *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.

![ Rows can be inserted using the layout vi.](./images/insert-row.png)

- *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.

![ Rows are removed using the layout vi.](./images/remove-row.png)

![ Select the *Finish layout* option to complete your design.](./images/finish-layout.png)
```

3. Run `gulp deploy` to deploy your layout template to the server you specified. If the layout is bundled with your theme, deploy the theme to deploy the layout template.

Related Topics

Importing Resources with Your Themes
Creating Themes

81.2 Creating Layout Templates Manually

Although you can generate layout templates with the Liferay Theme Generator, you may prefer to create or modify them manually using your own tools. In this tutorial you'll

learn the anatomy of Liferay DXP's layout templates so you can create layout templates manually or modify existing ones.

Understanding the Anatomy

Layout templates are made of rows and columns. The design you create specifies where users can place portlets on the page. An example row's HTML markup is 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>
```

Each row has a `div`, with the classes `portlet-layout` and `row`, that contains child `div`s for each column. Each column is indicated with the class `portlet-column`, as well as a class that specifies whether it is the first (`portlet-column-first`), last (`portlet-column-last`), or only column in the row (`portlet-column-only`).

Columns use the Bootstrap grid system and can therefore range in width from 1 to 12. Sizes are indicated with the number that follows the `col-[breakpoint]` class prefix (e.g. `col-md-6`). These specify two things: the percentage based width of the element and the media query breakpoint (`xs`, `sm`, `md`, or `lg`) for when this element expands to 100% width. 12 is the maximum amount, so `col-md-6` indicates `6`/12 width, or `50%`. These classes can also be mixed to achieve more advanced layouts, as shown above. In the example, medium sized viewports display `column-1` at 33.33% width and `column-2` at 66.66% width, but on small sized view ports both `column-1` and `column-2` are 50% width.

The processor (`${processor.processColumn()}`) processes each column's content, taking two arguments: the column's `id`, and the classes `portlet-column-content` and `portlet-column-content-[case]` (if applicable), where `[case]` refers to the `first`, `last`, or `only` column in the row.

Note: Velocity layout templates are supported, but deprecated as of 7.0. We recommend that you convert your Velocity layout templates to FreeMarker at your earliest convenience.

Now that you understand a layout template's anatomy, you can write your own Liferay DXP layout templates!

Related Topics

Layout Templates with the Liferay Theme Generator

Creating Themes

Creating Custom Layout Template Thumbnail Previews

81.3 Creating Custom Layout Template Thumbnail Previews

If you created a layout template with the Theme Generator, it generated a default thumbnail preview for your layout template. Follow these steps to create a custom preview thumbnail for a layout template:

1. Open the Command-line and navigate to the `docroot/layouttpl/custom` folder of your layout template. If you created the layout template in a theme created with the Liferay Theme Generator, the thumbnail is located in your theme's `src/layouttpl/custom` folder.
2. Replace the thumbnail PNG file, if it exists, with a custom thumbnail PNG with the same dimensions (120 x 120 px), or create a new one.
3. If including the layout template with a theme, specify the thumbnail's location in your theme's `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>
        /layouttpl/custom/porygon_50_50_width_limited.ftl
    </template-path>
    <thumbnail-path>
        /layouttpl/custom/porygon_50_50_width_limited.png
    </thumbnail-path>
</layout-template>
```

Deploy your layout template to your app server to use it. If your layout template is bundled with a theme, it deploys when the theme is deployed. Now you know how to create a custom thumbnail preview for your Liferay DXP layout templates!

Related topics

Layout Templates with the Liferay Theme Generator
    Creating Layout Templates Manually
    Creating Themes

81.4 Including Layout Templates with a Theme

Although you can deploy a layout template by itself, you can also bundle it with a theme. To include a layout template with a theme, follow these steps:

1. Open your theme's `liferay-look-and-feel.xml` file, and nest `<layout-templates>` tags in between the `<theme>...</theme>` tags so it matches the configuration below:

```
<theme id="my-theme-name" name="My Theme Name">
    ...
      <layout-templates>
        <custom>
            //layout template code goes here
        </custom>
      </layout-templates>
    ...
</theme>
```

2. Place the layout template in between the `<custom>...</custom>` tags, using the `<layout-template>` tag. The `<layout-template>` tag's `id` attribute **must match** the layout template's filename. Below is an example configuration:

```
<layout-template id="my_liferay_layout_template"
name="My Liferay Layout Template">
```

3. Specify the layout template's path with a `<template-path>` tag, as shown below:

```
<template-path>
    /layoutttpl/custom/my_liferay_layout_template.ftl
</template-path>
```

4. Specify the layout template thumbnail's path with a `<thumbnail-path>` tag, as shown below:

```
<thumbnail-path>
    /layoutttpl/custom/my_liferay_layout_template.png
</thumbnail-path>
```

5. Place the completed layout in your theme's `src/layouttpl` folder. Below is an example `liferay-look-and-feel` configuration:

```
<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.ftl
            </template-path>
            <thumbnail-path>
                /layoutttpl/custom/my_liferay_layout_template.png
            </thumbnail-path>
          </layout-template>
        </custom>
      </layout-templates>
    ...
</theme>
```

Now you know how to include layout templates with your Liferay DXP themes!

Related topics

Creating Custom Layout Template Thumbnail Previews
Layout Templates with the Liferay Theme Generator
Creating Layout Templates Manually

81.5 Upgrading 6.2 Layout Templates to 7.1

Upgrading your Liferay DXP 6.2 layout template to 7.0 a few updates:

1. Open your layout template's `liferay-plugin-package.properties` file and update the `liferay-versions` property to `7.1.0+`:

```
liferay-versions=7.1.0+
```

2. Update the Bootstrap `span[number]` classes to use the newer `col-[size]-[number]` classes. See Creating Layout Templates Manually for more information on Bootstrap's grid system.

3. Save the changes.

Note: Velocity layout templates are supported, but deprecated as of 7.0. We recommend that you convert your Velocity layout templates to FreeMarker at your earliest convenience. See Creating Layout Templates Manually for an example of the updated syntax.

Related Topics

Layout Templates with the Liferay Theme Generator
    Creating Layout Templates Manually
    Including Layout Templates with a Theme

81.6 Upgrading 7.0 Layout Templates to 7.1

If you're upgrading your Liferay DXP 7.0 layout template to 7.0, you must upgrade your layout template version to 7.1:

1. Open your layout template's `liferay-plugin-package.properties` file.

2. Update the `liferay-versions` property to `7.1.0+`:

```
liferay-versions=7.1.0+
```

3. Save the changes.

Note: Velocity layout templates are supported, but deprecated as of 7.0. We recommend that you convert your Velocity layout templates to FreeMarker at your earliest convenience. See Creating Layout Templates Manually for an example of the updated syntax.

Related Topics

Layout Templates with the Liferay Theme Generator
    Creating Layout Templates Manually
    Including Layout Templates with a Theme

CHAPTER 82

PORTLETS AND THEMES

The default theme sets the basic look and feel for all your portlets, and, through Portlet Decorators, gives you a way to fine-tune the look of individual portlets with the click of a mouse. But you aren't limited to defaults. The following sections explain

- how to modify themes to create custom templates for your portlets
- how to create your own decorators to customize the look of your portlets individually
- how to embed portlets in your themes, a function that lets you choose portlets to deploy automatically on any page where a given theme is used.

82.1 Theming Portlets

Themes can provide additional styles for your apps. You can change the markup for portlet containers by modifying the theme's `portlet.ftl` file.

This tutorial demonstrates how to style portlets with your themes.

Portlet FTL

Although you can individually style a portlet via the theme's CSS or the portlet's Look and Feel Configuration menu, you may want to modify the default look and feel for all portlets in your site. A portlet's template–its container, CSS classes, and overall HTML Markup–is defined via the theme's `portlet.ftl` file. To provide a custom style for all portlets, use the CSS classes in this file for the various container elements, in conjunction with the portlet decorators to achieve the desired look and feel. Be cautious: changes to `portlet.ftl` affect all the portlets in your site when the theme is applied.

To help you with your bearings as you modify your portlet's template, below is a quick look at the `portlet.ftl` file 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())
/>
```

The variables shown above are used throughout the template, so its important that you understand them before modifying the file:

- `portletDisplay`: is fetched from the `themeDisplay` object and contains information about the portlet.
- `portlet_back_url`: URL to return to the previous page when the portlet `WindowState` is maximized.
- `portlet_display_name`: The "friendly" name of the portlet as displayed in the GUI.
- `portlet_display_root_portlet_id`: The root portlet ID of the portlet.
- `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).

Next, a 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 and the header is displayed:

```
<#if portlet_display.isPortletDecorate() && !portlet_display.isStateMax()
&& portlet_display.getPortletConfigurationIconMenu()??
&& portlet_display.getPortletToolbar()??>
```

You can use a similar pattern if you want to dynamically show portions of the portlet's UI.

Next, the portlet title menus are defined. These are used in portlets that let you add resources (Web Content Display, Media Gallery, Documents and Media):

```
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, renderResp
```

The rest of the file contains the HTML markup for the portlet topper and the portlet content. This section barely scratches the surface of the `portlet.ftl` file. You must examine the `portlet.ftl` file yourself and determine what elements and classes need updated for your theme and site.

Now that you are more familiar with your theme's `portlet.ftl` file, you can learn the role Portlet Decorators play in the overall look and feel of your portlets.

Portlet Decorators

With Portlet Decorators, you can customize the style of the application wrapper. Themes come bundled with three default portlet decorators in their `liferay-look-and-feel.xml`:

- `Barebone`: this decorator displays the bare application content, showing neither the wrapping box nor the custom application title.

- `Borderless`: this decorator shows the title at the top, but does not display a wrapping box.

- `decorate`: this is the default Portlet Decorator when using the Classic theme. It wraps the application in a white box with a border, and displays the title at the top.

Now you know how to make your portlets stylish! But if default decorators are not stylish enough for you, [EDITOR: Our defaults are always stylish!] go to Portlet Decorators to learn how make and apply your own.

Related Topics

Look and Feel Configuration
    Creating Configurable Styles For Portlet Wrappers
    Themes and Layout Templates

CHAPTER 83

CREATING CONFIGURABLE STYLES FOR PORTLET WRAPPERS

Portlet Decorators customize the style of an application's wrapper. If you inspect the markup of your 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 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.

With Portlet Decorators, you can 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 DXP application when the *Decorate* Portlet Decorator is applied:

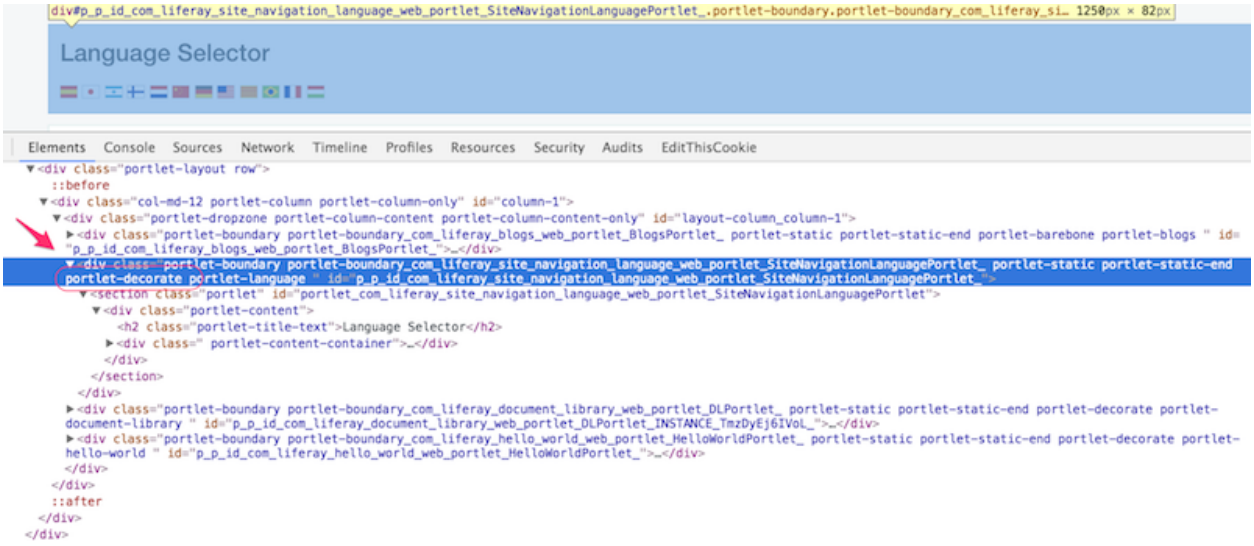


Figure 83.1: Portlet Decorators add the decorator's CSS class to the application's wrapper

Once your Portlet Decorator is complete, apply it to your applications through the Look and

Feel Configuration menu.

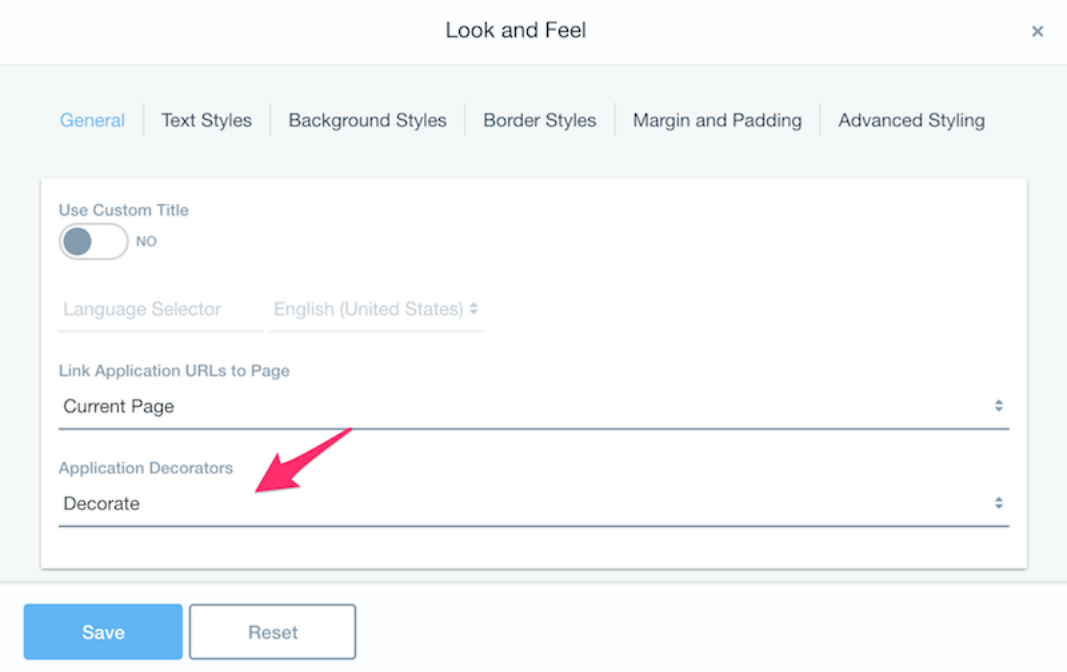


Figure 83.2: Portlet Decorators can be applied through the Look and Feel Configuration menu

The tutorials in this section detail how to customize Portlet Decorators and apply them to your applications.

83.1 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 7.1 Classic theme includes three Portlet Decorators:

- Decorate: this is the default Application Decorator when using the Classic theme. It wraps the application in a white box with a border, and displays the title at the top. theme.
- Borderless: this decorator shows the title at the top, but does not display a wrapping box.
- Barebone: this decorator displays the bare application content, showing neither the wrapping box nor the custom application title.

Note: Upgrading to Liferay DXP assigns the *borderless* decorator automatically to those portlets that had the *Show Borders* option set to false in previous versions of Liferay.

This tutorial demonstrates how to

- Add Portlet Decorators to your theme

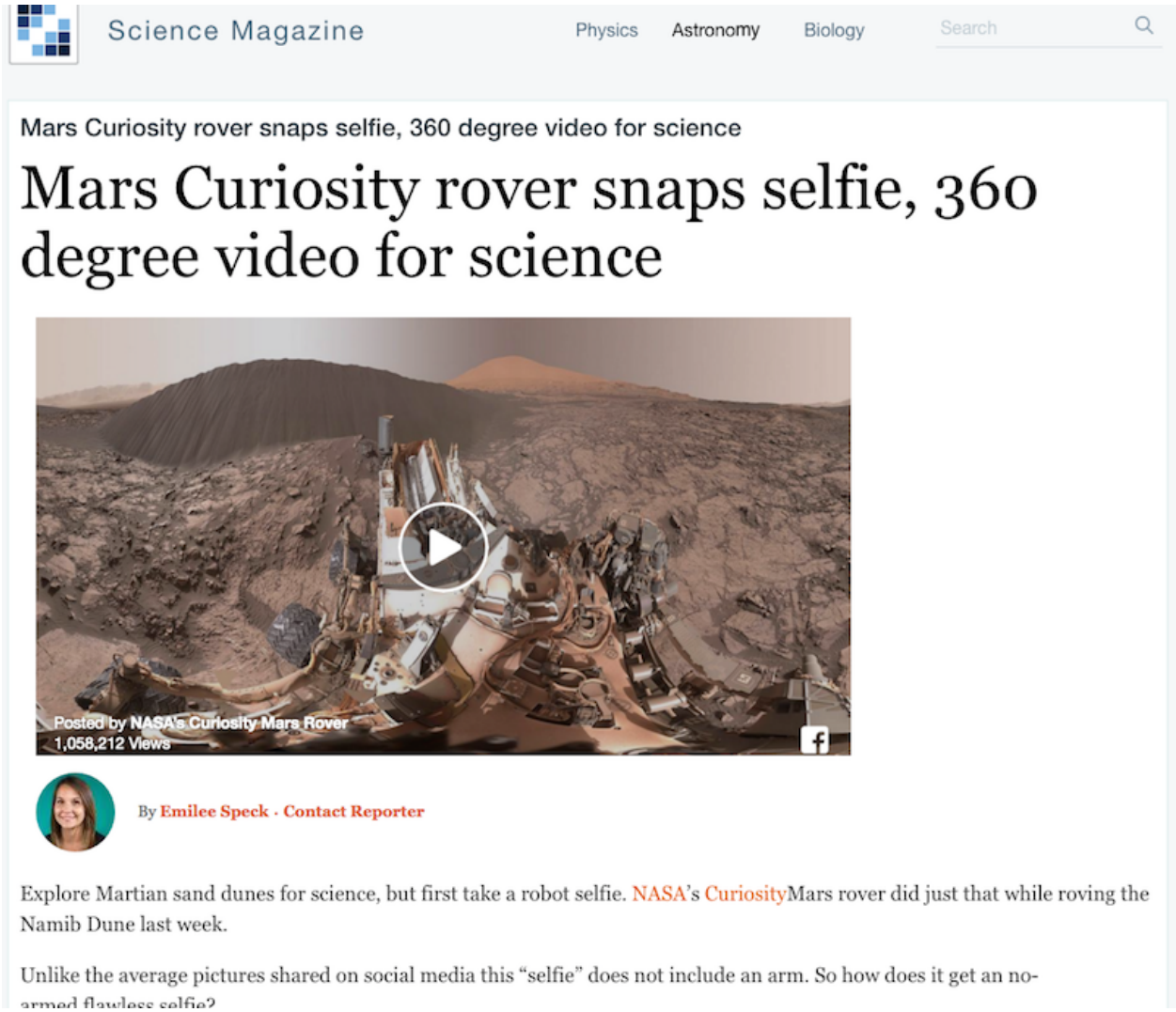


Figure 83.3: The Classic theme's Decorate Application Decorator wraps the portlet in a white box.

- Affect theme markup with Portlet Decorators

Adding Portlet Decorators to a Theme

Adding Portlet Decorators to your theme is similar to adding Color Schemes. 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

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`.

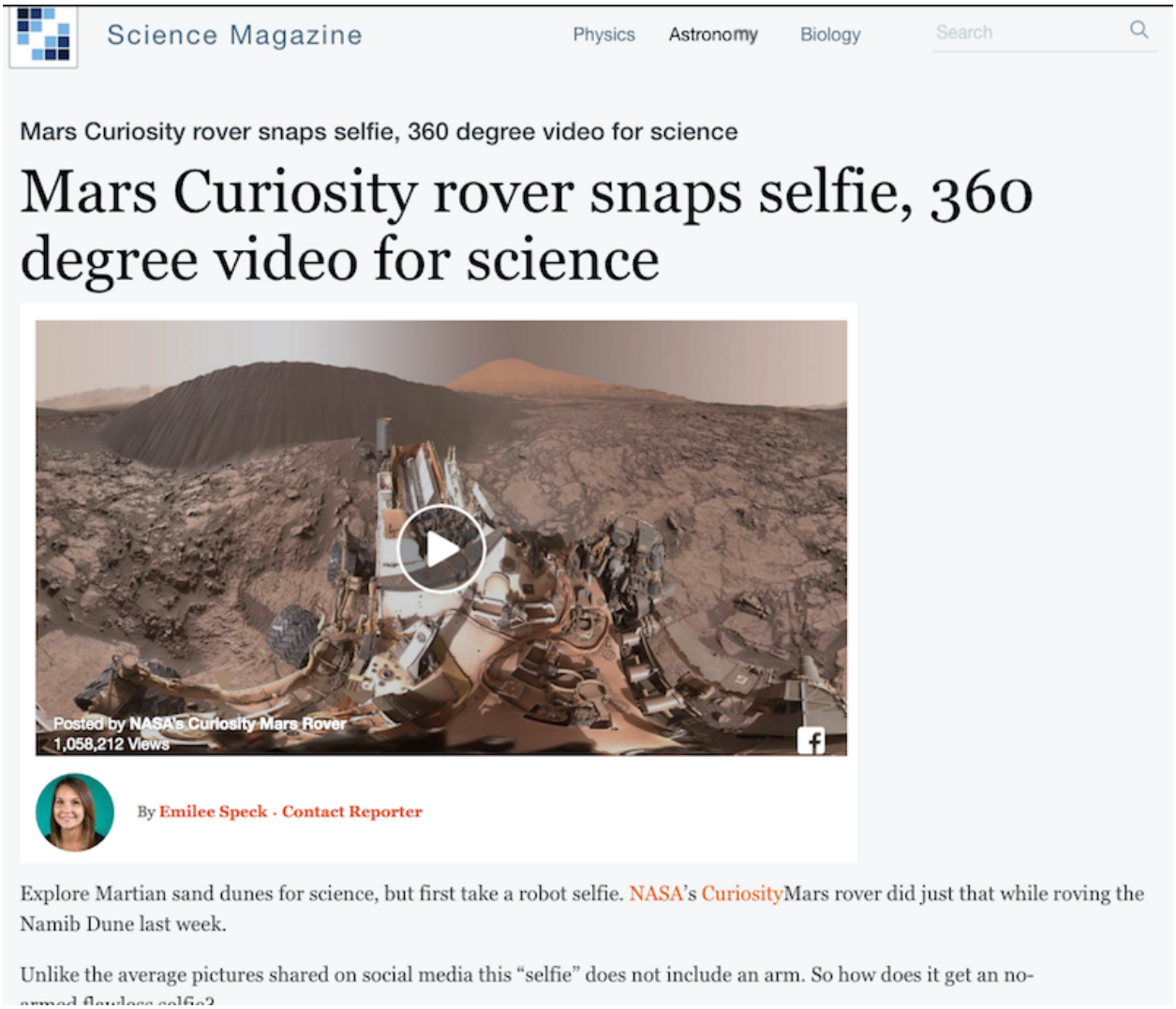


Figure 83.4: The Classic theme's Borderless Application Decorator displays the application's custom title.

The Document Type Definition for the liferay-look-and-feel.xml 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>7.1.0+</version>
    </compatibility>
    ...
        <theme id="classic" name="Classic">
        ...
        <portlet-decorator id="barebone" name="Barebone">
            <portlet-decorator-css-class>portlet-barebone</portlet-decorator-css-class>
        </portlet-decorator>
        <portlet-decorator id="borderless" name="Borderless">
```

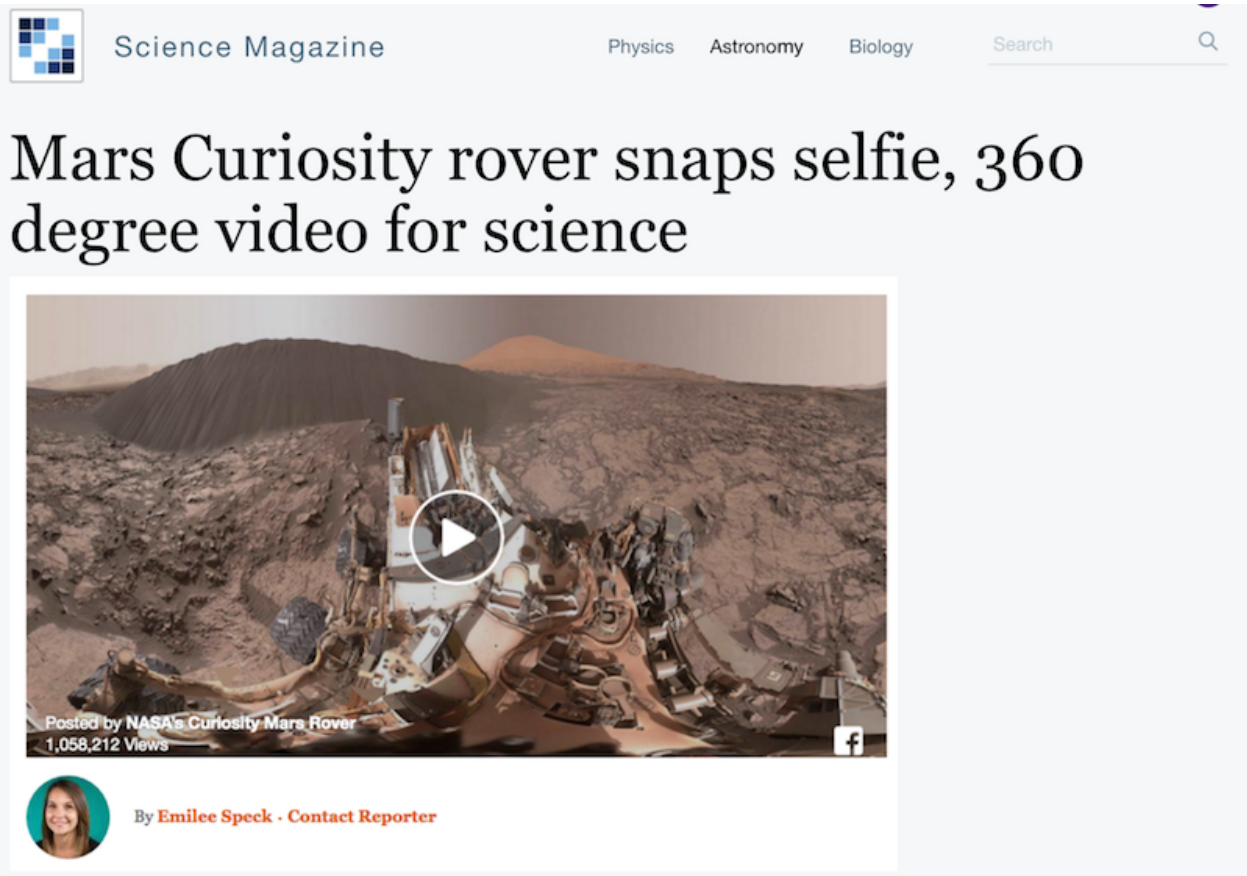


Figure 83.5: The Classic theme's Barebone Application Decorator displays only the application's content.

```
            <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 one asks for some 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, import the CSS file into your `_custom.scss`:

```
@import "portlet_decorator"
```

That's all that's 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 `portletDisplay` 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 !stringUtil.equals(portlet_display.getPortletDecoratorId(), "barebone")>
    <h2 class="portlet-title-text">${portlet_title}</h2>
</#if>
```

Now you know how to add Portlet Decorators to your theme. Let your creativity be your guide.

Related Topics

Themelets

Making Applications Configurable

83.2 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 Search portlet in its `portal_normal.ftl`:

```
<div class="navbar navbar-top navigation-bar-secondary">
  <div class="container user-personal-bar">
    <#assign preferences =
    freeMarkerPortletPreferences.getPreferences(
      "portletSetupPortletDecoratorId", "barebone"
    ) />

    <#if show_header_search>
      <div class="ml-auto mr-4 navbar-form" role="search">
        <@liferay.search default_preferences="${preferences}" />
      </div>
    </#if>

    <@liferay.user_personal_bar />
  </div>
</div>
```

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. The example below assigns the `barebone` decorator to the `preferences` variable:

   ```
   <#assign preferences = freeMarkerPortletPreferences.getPreferences(
     "portletSetupPortletDecoratorId", "barebone"
   ) />
   ```

2. Next, set the `default_preferences` attribute of the portlet's tag to the variable you just defined (`preferences` in the last step):

   ```
   <@liferay.search default_preferences= "${preferences}" />
   ```

Your embedded portlets now have a custom default Application Decorator!

Related Topics

Embedding Portlets in Themes

Theming Portlets

83.3 Embedding Portlets in Themes

You may occasionally want to embed a portlet in a theme, making the portlet visible on all pages where the theme is used. Since there are numerous drawbacks to hard-coding a specific portlet into place, the *Portlet Providers* framework offers an alternative that displays the appropriate portlet based on a given entity type and action.

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. To avoid problems, it's usually best to embed portlets with an entity type and action, but you may encounter circumstances where you'll want to hard code it by portlet name. Both methods are covered here.

Embedding a Portlet by Entity Type and Action

Start by inserting the following declaration where you want the portlet embedded:

```
<@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. It also 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. Retrieve portlet preferences using the `freeMarkerPortletPreferences` object. The example below retrieves the `barebone` portlet decorator:

```
<#assign preferences = freeMarkerPortletPreferences.getPreferences(
  "portletSetupPortletDecoratorId", "barebone"
) />
```

2. Set the `defaultPreferences` attribute of the embedded portlet to the `freeMarkerPortletPreferences` object you just configured:

```
<@liferay_portlet["runtime"]
    defaultPreferences="${preferences}"
    portletName="com_liferay_login_web_portlet_LoginPortlet"
/>
```

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 to use for the application. The default value is `portletInstance`, but it 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

Embedding Portlets in Themes
Portlets
Service Builder

CHAPTER 84

CLAY CSS AND THEMES

Lexicon is a design language that provides a common framework for building consistent UIs. Clay, the web implementation of Lexicon, is an extension of Bootstrap's open source CSS Framework. Bootstrap is by far the most popular CSS framework on the web. Built with Sass, Clay CSS fills the front-end gaps between Bootstrap and the specific needs of Liferay DXP.

These tutorials look briefly at Clay CSS and show you how to use it in your themes.

84.1 Importing Clay CSS into a Theme

Clay 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 Clay 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 Clay's structure.

Clay CSS Structure

Clay CSS is bundled with two sub-themes: Clay Base and Atlas. Clay Base is Liferay DXP's Bootstrap API extension. It adds all the features and components you need and inherits Bootstrap's styles. As a result, Clay Base is fully compatible with third party themes that leverage Bootstrap's Sass variable API.

Atlas is Liferay DXP's custom Bootstrap theme that is used in the Classic Theme. Its purpose is to overwrite and manipulate Bootstrap and Clay Base to create its 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 clay.scss with the code below and save:

```
@import "clay/atlas";
```

2. Add a file named _imports.scss with the code below and save:

```
@import "bourbon";
@import "mixins";
@import "clay/atlas-variables";
```

3. Add a file named font_awesome.scss and add the font-awesome path and import:

```
// Icon paths

$FontAwesomePath: "font-awesome/font";

@import "font-awesome/scss/font-awesome";
```

4. Add a file named _clay_variables.scss. All your Atlas, Bootstrap, and Clay Base variable modifications must be placed in this file.

5. 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 `gulp kickstart` command and following the prompts.

Next you can learn how to customize the Clay Base.

Customizing Clay Base

You can customize Clay Base with just a few imports.

In your custom theme's `/src/css` folder (legacy ant themes: `/_diff/css`) add a file named `font_awesome.scss` with the code below and save:

```
// Icon paths

    $FontAwesomePath: "font-awesome/font";

    @import "font-awesome/scss/font-awesome";
```

Then in that same folder, add a file named `_clay_variables.scss`. All your Atlas, Bootstrap, and Clay Base variable modifications must be placed in this file.

As mentioned earlier, any custom CSS should be placed in `_custom.scss`.

Now you know how to use Clay CSS in your theme!

Related Topics

Applying Clay Styles to Your App

Integrating Third Party Themes with Clay

84.2 Integrating Third Party Themes with Clay

Clay Base provides all the features and components your theme needs and inherits Bootstrap's styles. As a result, Clay Base is fully compatible with third party themes that leverage Bootstrap's Sass variable API.

The Styled Theme uses Clay Base to provide its styles and components. Therefore, as a best practice, you should use the Styled base theme to integrate third party themes.

Note: You can purchase third party themes from the Liferay Marketplace. Third party themes must be built with Sass to be compatible. **Make sure** Sass files are included before making any theme purchase.

Follow these steps to integrate a third party theme with Clay Base:

1. Create a new theme with the Styled Theme as its base. This is the default base theme for newly created themes, so no further action is required. This provides the Clay Base files you need.
2. In the theme's `/src/css` folder (legacy ant themes: `/_diff/css`) add a file named `font_awesome.scss` with the code below and save:

```
// Icon paths

$FontAwesomePath: "font-awesome/font";

@import "font-awesome/scss/font-awesome";
```

3. In that same folder, add a file named `_clay_variables.scss`. All your Atlas, Bootstrap, and Clay Base variable modifications must be placed in this file.
4. 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`)
5. Copy the CSS contents of the theme to the folder you just created.
6. In `_clay_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.
```

---

7. In `_custom.scss`, import the file containing the CSS. For example, `@import "awesome-theme/main.scss";`
8. Deploy your theme with `gulp deploy` (for legacy ant themes, use `ant deploy`)

Now you know how to integrate third party themes with Clay Base!

Related Topics

Applying Clay Styles to Your App
    Importing Clay CSS into a Theme

CHAPTER 85

LIFERAY JAVASCRIPT APIS

The `Liferay` JavaScript object exposes methods, objects, and properties that you can use to access Liferay DXP-specific information. This section contains a comprehensive list of some of the most useful utilities you can find inside the `Liferay` object.

85.1 Accessing ThemeDisplay Information

The `Liferay` global JavaScript Object exposes useful methods, objects, and properties, each containing a wealth of information, one of which is `ThemeDisplay`. If you have experience with Java development in Liferay DXP, you may be familiar with ThemeDisplay. The JavaScript object exposes the same information as the ThemeDisplay Java Class. It gives you access to valuable information that you can use in your applications, such as the Portal instance, the current user, the user's language, whether the user is signed in or being impersonated, the file path to the theme's resources, and much more.

The `Liferay` global object is automatically available in Liferay DXP at runtime. To access the `ThemeDisplay` object, use the following dot notation in your app:

```
`Liferay.ThemeDisplay.method-name`
```

This tutorial describes some of the most commonly used `ThemeDisplay` methods for retrieving IDs, file paths, and login information. An exhaustive list of all of the available methods is displayed in the table at the end of this tutorial.

Retrieving IDs

The methods below retrieve various Portal elements related to the current user:

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.

Next you can learn how to access file paths for various deployed entities.

Retrieving File Paths

The methods below retrieve file paths for various theme resources:

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.

Next you can learn how to retrieve information for the current user.

Retrieving Login Information

The methods below return a boolean value indicating whether the current user is signed in or being impersonated:

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.

The example configuration below alerts users with a standard message if they are a guest or a personal greeting if they are signed in. This is a basic example, and perhaps a bit invasive, but it illustrates how you can create unique experiences for each user with the `ThemeDisplay` APIs:

```
if(Liferay.ThemeDisplay.isSignedIn()){
    alert('Hello ' + Liferay.ThemeDisplay.getUserName() + '. Welcome Back.')
}
else {
    alert('Hello Guest.')
}
```

Liferay ThemeDisplay Methods

A complete list of the available `Liferay.ThemeDisplay` methods is shown in the table below:

Method	Type	Description
getLayoutId	number	
getLayoutRelativeURL	string	Returns the relative URL for the page
getLayoutURL	string	
getParentLayoutId	number	
isControlPanel	boolean	
isPrivateLayout	boolean	
isVirtualLayout	boolean	
getBCP47LanguageId	number	
getCDNBaseURL	string	Returns the content delivery network (CDN) base URL, or the current portal URL if the CDN base URL is null

Method	Type	Description
getCDNDynamicResourcesHost	string	Returns the content delivery network (CDN) dynamic resources host, or the current portal URL if the CDN dynamic resources host is null
getCDNHost	string	
getCompanyGroupId	number	
getCompanyId	number	Returns the portal instance ID
getDefaultLanguageId	number	
getDoAsUserIdEncoded	string	
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	Returns the primary key of the page
getPortalURL	string	Returns the portal instance's base URL
getScopeGroupId	number	Returns the ID of the scoped or sub-scoped active group (e.g. site)
getScopeGroupIdOrLiveGroupId	number	
getSessionId	number	Returns the session ID, or a blank string if the session ID is not available to the application
getSiteGroupId	number	
getURLControlPanel	string	
getURLHome	string	
getUserId	number	Returns the ID of the user for which the current request is being handled
getUserName	string	Returns the user's name
isAddSessionIdToURL	boolean	
isFreeformLayout	boolean	

Method	Type	Description
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

85.2 Working with URLs in JavaScript

The `Liferay` global JavaScript Object exposes methods, objects, and properties that access the portal context. Four of these are helpful when working with URLS: `authToken`, `currentURL`, `currentURLEncoded`, and `PortletURL`. If you have experience with Java development in Liferay DXP, you may have worked with some of these before. The `Liferay` global object is automatically available at runtime, so no additional dependencies are required.

This tutorial covers how to use the `Liferay` global JavaScript object to manipulate URLs. A complete list of the available methods and properties appears in the tables at the end of this tutorial.

Liferay PortletURL

The `Liferay.PortletURL` object provides methods for creating portlet API URLs (`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);
```

See the Portlet URL Methods and Properties section for more information about the methods and properties used in the example above.

Liferay AuthToken

The `Liferay.authToken` property holds the current authentication token value as a String. The `authToken` is used to validate permissions when you make calls to services. To use the `authToken` in a URL, pass `Liferay.authToken` as the URL's `p_auth` parameter, as shown in the example below:

```
portletURL.setParameter('p_auth', Liferay.authToken);
```

Liferay CurrentURL

The `Liferay.currentURL` property 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` property holds the path of the current URL, encoded in ASCII for safe transmission over the Internet, from the server root.

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"
```

Portlet URL Methods and Properties

`Liferay.PortletURL` Methods:

Method	Parameters	Returns
`createURL`	`basePortletURL, params`	new `PortletURL(null, params, basePortletURL);`
`createActionURL`		new `PortletURL(PortletURL.ACTION_PHASE);`
`createRenderURL`		new `PortletURL(PortletURL.RENDER_PHASE);`
`createResourceURL`		new `PortletURL(PortletURL.RESOURCE_PHASE);`

`Liferay.PortletURL` Properties:

Property	Value
`ACTION_PHASE`	"1"
`RENDER_PHASE`	"0"
`RESOURCE_PHASE`	"2"

Once the portlet URL is created, you have access to several methods that you can use to manipulate the URL further:

Method	Description	Parameters	Returns
`setDoAsGroupId`	Sets the ID of the site, organization, or user group for the URL	`doAsGroupId`	The updated Portlet URL Object
`setDoAsUserId`	Sets the ID of the user to impersonate	`doAsUserId`	The updated Portlet URL Object
`setEscapeXML`	Sets whether the URL should be XML escaped	`true` or `false`	The updated Portlet URL Object
`setLifecycle`	Sets the portlet lifecycle of this URL's target portlet	`lifecycle`	The updated Portlet URL Object
`setName`	sets the portlet URL's `javax.portlet.action` name	`name`	The updated Portlet URL Object
`setParameter`	Creates an individual parameter or replaces an existing reserved parameter	key,value	The updated Portlet URL Object
`setParameters`	Creates multiple parameters and/or replaces existing reserved parameters	{key:value,...}	The updated Portlet URL Object
`setPlid`	Sets the portlet layout ID	`plid`	The updated Portlet URL Object
`setPortletId`	Sets the ID of the target portlet	`portletId`	The updated Portlet URL Object
`setPortletMode`	Sets the portlet mode, if the URL triggers a request	`portletMode`	The updated Portlet URL Object
`setResourceId`	Sets the ID of the URL's target resource	`ResourceId`	The updated Portlet URL Object
`setSecure`	Sets whether to make the URL secure (HTTPS).	`true` or `false`	The updated Portlet URL Object
`setWindowState`	Sets the portlet's window state, if the URL triggers a request	`windowState`	The updated Portlet URL Object
`toString`	Returns the URL as a String		The portlet URL as a String
`_isReservedParam`	Returns whether the parameter is reserved	`paramName`	`true` if the parameter is reserved

Now you know how to manipulate URLs using methods within the `Liferay` global JavaScript object.

Related Topics

Liferay DXP JavaScript Utilities
Liferay Theme Display

85.3 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	Return Type	Description
acceptsGzip	boolean	Returns whether the browser accepts gzip file compression
getMajorVersion	number	Returns the major version of the browser
getRevision	number	Returns the revision version of the browser
getVersion	number	Returns the major.minor version of the browser
isAir	boolean	Returns whether the browser is Adobe AIR
isChrome	boolean	Returns whether the browser is Chrome
isFirefox	boolean	Returns whether the browser is Firefox
isGecko	boolean	Returns whether the browser is Gecko
isIe	boolean	Returns whether the browser is Internet Explorer
isIphone	boolean	Returns whether the browser is on an Iphone
isLinux	boolean	Returns whether the browser is being viewed on Linux
isMac	boolean	Returns whether the browser is being viewed on Mac

Method	Return Type	Description
isMobile	boolean	Returns whether the browser is being viewed on a mobile device
isMozilla	boolean	Returns whether the browser is Mozilla
isOpera	boolean	Returns whether the browser is Opera
isRtf	boolean	Returns whether the browser supports RTF
isSafari	boolean	Returns whether the browser is Safari
isSun	boolean	Returns whether the browser is being viewed on Sun OS
isWebKit	boolean	Returns whether the browser is WebKit
isWindows	boolean	Returns whether the browser is being viewed on Windows

Related Topics

Accessing ThemeDisplay Information

85.4 Invoking Liferay Services

Liferay DXP provides many web services out-of-the-box. To see a comprehensive list of the available web services, navigate to `http://localhost:8080/api/jsonws` (assuming your localhost is running on port 8080). If you've deployed your own Service Builder-generated JSON web services, follow these guidelines for invoking them. These services are useful for creating single page applications and can even be used to create custom front-ends in Liferay DXP.

This tutorial explains how to invoke these web services using JavaScript.

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 invokes JSON web services.

The `Liferay.Service` method takes four possible arguments:

service {string|object}: Specify 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}: Specify 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 its first parameter.

exceptionCallback {function}: A function to execute when the response from the server contains a service exception. It receives an exception message as its first parameter.

One of the 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 its `companyId` and `emailAddress`. 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 way to invoke the `Liferay.Service` method is by passing an object with the keys of the service to call and the value of 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 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 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 these concepts:

```
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.

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 DXP JavaScript Utilities

Accessing ThemeDisplay Information

CHAPTER 86

JAVASCRIPT MODULE LOADERS

A JavaScript module encapsulates code into a useful unit that exports its capability/value. This makes it easier to see the broader scope, easier to find what you're looking for, and keeps related code close together. 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 lets an application load dependencies easily by specifying a string that identifies the JavaScript module's name.

These tutorials show you how to load JavaScript modules in Portal.

86.1 Loading AMD Modules in Liferay

Modularized JavaScript code is a specification for the JavaScript language called Asynchronous Module Definition, or AMD. The Liferay AMD Module Loader is the native loader that you can use to load your AMD modules. This tutorial covers how to use the Liferay AMD Module Loader.

Note: While you can manually configure the AMD Loader, we recommend that you use the liferay-npm-bundler instead.

Configuring Your AMD Module for the Loader

Follow these steps to prepare your module:

1. Wrap your AMD module code with the `Liferay.Loader.define()` method, such as the one shown below:

```
Liferay.Loader.define('my-dialog', ['my-node', 'my-plugin-base'],
function(myNode, myPluginBase) {
   return {
       log: function(text) {
           console.log('module my-dialog: ' + text);
       }
   };
});
```

2. You can modify the configuration to load the module when another module is triggered or when a condition is met. The configuration below specifies that this module should be loaded if the developer requests the `my-test` module:

```
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 Liferay AMD Loader uses the definition, along with the listed dependencies, as well as any other configurations specified, to create a `config.json` file. This configuration object tells the loader which modules are available, where they are located, and what dependencies they require. 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"]
    }
}
```

3. Load your module in your scripts. Pass 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);
});
```

Note: By default, the AMD Loader times out in seven seconds. Since Liferay DXP Fix Pack 3 and Liferay Portal 7.1 CE GA 2, you can configure this value through System Settings. Open the Control Panel and navigate to *Configuration → System Settings → PLATFORM → Infrastructure*, and select *JavaScript Loader*. Set the *Module Definition Timeout* configuration to the time you want and click *Save*.

Related Topics

Loading Modules with AUI Script
    Using npm in Your Portlets

86.2 Using External JavaScript Libraries

You can use external JavaScript libraries in your portlets (i.e., anything but Metal.js, jQuery, or Lodash, which are included by default). 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.

This tutorial covers how to adapt external libraries for the JavaScript Loaders.

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 use 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. Follow these steps:

1. Open the Control Panel, navigate to *Configuration → System Settings*.

2. Click *JavaScript Loader* under *Platform → Infrastructure*.

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()`.

Now you know how to adapt external libraries for Liferay's JavaScript Loaders.

Related Topics

Liferay AMD Module Loader
Using ES2015+ Modules in Your Portlet

86.3 Loading Modules with AUI Script

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.

Note: AUI is deprecated and no longer in active development in 7.0, but all the tags will remain fully functional in Liferay DXP 7.1. Eventually, these tags will be replaced with Clay tag counterparts.

Using aui:script

The `aui:script` tag supports the following options:

- `require`: Requires an AMD module to load 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>
```

alternatively, you can specify a variable for your module by adding `as variableName` after the module name, as shown in the example below:

```
<aui:script require="metal-clipboard/src/Clipboard as myModule">
    new myModule.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

Using External JavaScript Libraries
    Loading AMD Modules

CHAPTER 87

USING FRONT-END FRAMEWORKS IN YOUR PORTLETS

You can use the most popular Front-End frameworks in your portlets with just a few minor updates to the standard portlet configuration. To help kick-start your portlet project, you can choose from several Blade portlet project templates. The following templates are covered in this section:

- Angular
- React
- Vue

Alternatively, you can use Liferay DXP's Liferay JS Generator to build JavaScript-based portlets. The tutorials in this section show how to use front-end frameworks in your portlets.

Note: JavaScript Server-Side-Rendering is not supported out-of-the-box. To use JS frameworks for site rendering, you **must** set up your server-side (or search-crawler) rendering generation to support them.

87.1 Using React in Your Portlets

You can use the npm React portlet template to automate much of the required configuration for you or create the module manually. For convenience, all manual steps are listed below. This tutorial shows how to use React in your portlets, whether you're migrating an existing React project or building a fresh project. See the npm React portlet template reference docs for more information on the portlet's anatomy or the react npm portlet sample for a React portlet example that you can test and deploy right now. Get started by creating your OSGi module and configuring its metadata.

Configuring Metadata

Follow these steps to create the module and configure its metadata for React:

1. Create an OSGi module. For example, use the npm React portlet template.

2. Specify the `Web-ContextPath` BND Header in your project's `bnd.bnd` file. Below is the default configuration for the npm React portlet template:

```
Web-ContextPath: /my-npm-react-portlet
```

3. Create a `.babelrc` file and add the following presets to it:

```
{
    "presets": ["env", "react"]
}
```

4. Optionally add a `.npmbundlerrc` file to your project's root folder. This file is not required. You can, however, configure this file to customize the `liferay-npm-bundler` to suit your needs, such as to ignore files.

5. Include the following dependency in your `build.gradle` file:

```
compileOnly group: "com.liferay",
name: "com.liferay.frontend.js.loader.modules.extender.api",
version: "2.0.2"
```

6. Create a `package.json` in your project if it doesn't already exist with the configuration shown below. Update the `"main"` JS path to point to your app's main JS file. Note that the `liferay-npm-bundler` is added last to the build script. List any additional build processes before this that your project requires:

```
{
    "dependencies": {
        "react": "15.6.2",
        "react-dom": "15.6.2"
    },
  "description": "React Portlet",
    "devDependencies": {
        "babel-cli": "^6.26.0",
        "babel-preset-env": "^1.7.0",
        "babel-preset-react": "6.24.1",
        "liferay-npm-bundler": "^2.0.0"
    },
    "main": "js/index.js",
    "name": "my-npm-react-portlet",
    "scripts": {
        "build": "babel --source-maps -d
    build/resources/main/META-INF/resources
    src/main/resources/META-INF/resources && liferay-npm-bundler"
    },
    "version": "1.0.0"
}
```

To use ES2015+ syntax in your portlet, you must transpile it for the browser. Babel, included in your build script, takes care of this for you.

Next You can configure the portlet.

Configuring the Portlet

Follow these steps to configure your portlet:

1. Create a Component class that implements the `Portlet.class` service:

```
@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=" + MyNpmReactPortletKeys.MyNpmReact,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class MyNpmReactPortlet extends MVCPortlet {
  ...
}
```

2. If your React project includes CSS styling as well, add the following additional property to specify the location of the main CSS file:

```
"com.liferay.portlet.header-portlet-css=/css/main.css"
```

   Note that this path is relative to the resources path. If using Sass, drop the `.scss` extension in this property and use `.css` instead. For example, if your main CSS file is located in `src/main/resources/META-INF/resources/css/app.scss`, then you would have the following configuration:

```
"com.liferay.portlet.header-portlet-css=/css/app.css"
```

3. To improve code maintenance, use the NPMResolver APIs to alias your module's package name. The example below exposes the module's name as `bootstrapRequire`:

```
@Override
public void doView(
        RenderRequest renderRequest, RenderResponse renderResponse)
    throws IOException, PortletException {

    JSPackage jsPackage = _npmResolver.getJSPackage();

    renderRequest.setAttribute(
        MyNpmReactWebKeys.BOOTSTRAP_REQUIRE,
        jsPackage.getResolvedId() + " as bootstrapRequire");

    super.doView(renderRequest, renderResponse);
}

@Reference
private NPMResolver _npmResolver;
```

4. Inside your `init.jsp`, add the following Java scriptlet to access the `bootstrapRequire` variable in your portlet's `view.jsp`:

```
<%
String bootstrapRequire = (String)renderRequest.getAttribute(
  MyNpmReactWebKeys.BOOTSTRAP_REQUIRE
);
%>
```

Next you can learn how to render your app's component.

Rendering Your Component

Follow these steps to render your app component:

1. Inside your app's main JS file (`index.js` for example), use the function below to render your component:

```
import React from 'react';
import ReactDOM from 'react-dom';
import AppComponent from './components/App' //parent component

export default function(elementId) {
  ReactDOM.render(<AppComponent />, document.getElementById(elementId));
}
```

2. Open your `view.jsp` and add an element container to house your component. Then add an `<aui:script>` tag and pass your aliased module name as the `require` attribute's value. Finally, call your module's `default` function that you exported in the previous step and pass the container element in as the element ID. Adding the `<portlet:namespace />` to the `<div>`'s `id` ensures that it is unique to the portlet and doesn't clash with any existing elements on the page:

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

<div id="<portlet:namespace />-root"></div>

<aui:script require="<%= bootstrapRequire %>">
    bootstrapRequire.default('<portlet:namespace />-root');
</aui:script>
```

Now you know how to use React in your projects!

Related Topics

Using Angular in Your Portlets
Using Vue in Your Portlets
Using npm in Your Portlets

87.2 Using Vue in Your Portlets

You can create a Vue project manually or use the npm Vue portlet template to automate much of the required configuration for you. For convenience, all manual steps are listed below. This tutorial shows how to use Vue JS in your portlets, whether you're migrating an existing Vue project or building a fresh one. See the npm Vue portlet template reference docs for more information on the portlet's anatomy or the npm Vue portlet sample for a Vue portlet example that you can test and deploy right now. Get started by creating your OSGi module and configuring its metadata.

Configuring Metadata

Follow these steps to create the module and configure its metadata for Vue:

1. Create an OSGi module. For example, use the npm Vue portlet template.

2. Specify the `Web-ContextPath` BND Header in your project's `bnd.bnd` file. Below is the default configuration for the npm Vue portlet template:

```
Web-ContextPath: /my-npm-vue-portlet
```

3. Create a `.babelrc` file and add the following presets to it:

```
{
    "presets": ["env"]
}
```

4. Optionally add a `.npmbundlerrc` file to your project's root folder. This file is not required. You can, however, configure this file to customize the liferay-npm-bundler to suit your needs, such as to ignore files.

5. Include the following dependency to your `build.gradle` file:

```
compileOnly group: "com.liferay",
name: "com.liferay.frontend.js.loader.modules.extender.api",
version: "2.0.2"
```

6. Create a `package.json` in your project if it doesn't already exist and add the configuration shown below to it. Update the `"main"` JS path to point to your app's main JS file. Note that the `liferay-npm-bundler` is added last to the build script. List any additional build processes before this that your project requires:

```
{
    "dependencies": {
        "vue": "2.4.4"
    },
  "description": "Vue.js Portlet",
    "devDependencies": {
        "babel-cli": "^6.26.0",
        "babel-preset-env": "^1.7.0",
        "liferay-npm-bundler": "^2.0.0"
    },
    "main": "js/index.js",
    "name": "my-npm-vuejs-portlet",
    "scripts": {
        "build": "babel --source-maps -d
    build/resources/main/META-INF/resources
    src/main/resources/META-INF/resources && liferay-npm-bundler"
    },
    "version": "1.0.0"
}
```

To use ES2015+ syntax in your portlet, you must transpile it for the browser. Babel, included in your build script, takes care of this for you.

Next You can configure the portlet.

Configuring the Portlet

Follow these steps to configure your portlet:

1. Create a Component class that implements the `Portlet.class` service:

```
@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=" + MyNpmVuejsPortletKeys.MyNpmVuejs,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class MyNpmVuejsPortlet extends MVCPortlet {
  ...
}
```

2. If your Vue project includes CSS styling, add the following additional property to specify the location of the main CSS file:

```
"com.liferay.portlet.header-portlet-css=/css/main.css"
```

   Note that this path is relative to the resources path. If using Sass, drop the `.scss` extension in this property and use `.css` instead. For example, if your main CSS file is located in `src/main/resources/META-INF/resources/css/app.scss`, then you would have the following configuration:

```
"com.liferay.portlet.header-portlet-css=/css/app.css"
```

3. To improve code maintenance, use the NPMResolver APIs to alias your module's package name. The example below exposes the module's name as `bootstrapRequire`:

```
@Override
public void doView(
        RenderRequest renderRequest, RenderResponse renderResponse)
    throws IOException, PortletException {

    JSPackage jsPackage = _npmResolver.getJSPackage();

    renderRequest.setAttribute(
        MyNpmVuejsWebKeys.BOOTSTRAP_REQUIRE,
        jsPackage.getResolvedId() + " as bootstrapRequire");

    super.doView(renderRequest, renderResponse);
}

@Reference
private NPMResolver _npmResolver;
```

4. Inside your `init.jsp`, add the following Java scriptlet to access the `bootstrapRequire` variable in your portlet's `view.jsp`:

```
<%
String bootstrapRequire = (String)renderRequest.getAttribute(
  MyNpmVuejsWebKeys.BOOTSTRAP_REQUIRE
);
%>
```

Next you can learn how to render your app's component.

Rendering Your Component

Follow these steps to render your app component:

1. Inside your app's main JS file (`index.js` for example), use the function below to render your component:

```
import Vue from 'vue/dist/vue.common';

export default function(portletNamespace) {
    // Application 1
    new Vue({
        el: `#${portletNamespace}-root`,
        data: {
            message: 'Hello from Vue.js!',
        },
        methods: {
            reverseMessage: function() {
                this.message = this.message
                    .split('')
                    .reverse()
                    .join('');
            },
        },
    });
}
```

2. Open your `view.jsp` and add an element container to house your component. Then, add an `<aui:script>` and pass your aliased module name as the `require` attribute's value. Finally, call your module's `default` function that you exported in the previous step and pass the portlet namespace. Adding the `<portlet:namespace />` to the `<div>`'s `id` ensures that it is unique to the portlet and doesn't clash with any existing elements on the page:

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

<div id="<portlet:namespace />-root"></div>

<aui:script require="<%= bootstrapRequire %>">
    bootstrapRequire.default('<portlet:namespace />');
</aui:script>
```

Now you know how to use Vue in your projects!

Related Topics

Using Angular in Your Portlets
Using React in Your Portlets
Using npm in Your Portlets

87.3 Using Angular in Your Portlets

You can use the npm Angular portlet template to automate much of the required configuration for you, or create the module manually. For convenience, all manual steps are listed below. This tutorial shows how to use Angular in your portlets, whether you're migrating an existing Angular project or building a fresh one. See the npm Angular portlet template reference docs for more information on the portlet's anatomy. Get started by creating your OSGi module and configuring its metadata.

Configuring Metadata

Follow these steps to create the module and configure its metadata for Angular:

1. Create an OSGi module. For example, use the npm Angular portlet template.

2. Specify the `Web-ContextPath` BND Header in your project's `bnd.bnd` file. Below is the default configuration for the npm Angular portlet template:

```
Web-ContextPath: /my-npm-angular-portlet
```

3. Create a `.tsconfig.json` file and add the following configuration to it:

```
{
    "compilerOptions": {
        "emitDecoratorMetadata": true,
        "experimentalDecorators": true,
        "lib": ["es2015", "dom"],
        "moduleResolution": "node",
        "outDir": "build/resources/main/META-INF/resources/lib",
        "sourceMap": true,
        "suppressImplicitAnyIndexErrors": true,
        "target": "es5",
        "typeRoots": ["./node_modules/@types/"],

        "module": "commonjs",
        "strict": true,
        "noFallthroughCasesInSwitch": true,
        "inlineSources": true,
        "declaration": false,
        "skipLibCheck": true,
        "types": ["jasmine", "node"]
    },
    "include": ["src/main/resources/META-INF/resources/**/*.ts"]
}
```

4. Optionally add a `.npmbundlerrc` file to your project's root folder. This file is not required. You can, however, configure this file to customize the liferay-npm-bundler to suit your needs, such as to ignore files.

5. Include the following dependency to your `build.gradle` file:

```
compileOnly group: "com.liferay",
name: "com.liferay.frontend.js.loader.modules.extender.api",
version: "2.0.2"
```

6. Create a `package.json` in your project if it doesn't already exist and add the configuration shown below to it. Update the `"main"` JS path to point to your app's main JS file. Note that the `liferay-npm-bundler` is added last to the build script. List any additional build processes before this that your project requires, such as the `tsc` (Typescript) process shown below:

```
{
    "dependencies": {
        "@angular/animations": "^5.0.0",
        "@angular/common": "^5.0.0",
        "@angular/compiler": "^5.0.0",
        "@angular/core": "^5.0.0",
        "@angular/forms": "^5.0.0",
        "@angular/http": "^5.0.0",
        "@angular/platform-browser": "^5.0.0",
        "@angular/platform-browser-dynamic": "^5.0.0",
        "@angular/platform-server": "^5.0.0",
        "@angular/router": "^5.0.0",
        "@ngx-translate/core": "^9.1.1",
        "core-js": "^2.5.1",
        "rxjs": "^5.5.2",
        "zone.js": "0.8.12"
    },
    "description": "Angular Portlet",
    "devDependencies": {
        "@angular/cli": "^1.6.7",
        "@angular/compiler-cli": "^5.0.0",
        "@compodoc/compodoc": "1.0.0-beta.10",
        "@types/bootstrap": "^3.3.33",
        "@types/bootstrap-datepicker": "0.0.6",
        "@types/jasmine": "2.5.48",
        "@types/jquery": "^2.0.46",
        "@types/moment": "^2.13.0",
        "@types/node": "~6.0.60",
        "@types/toastr": "^2.1.34",
        "chalk": "1.1.3",
        "codelyzer": "3.1.2",
        "del": "^3.0.0",
        "gulp": "^3.9.1",
        "gulp-flatten": "^0.3.1",
        "gulp-sass": "^3.1.0",
        "inline-ng2-resources": "^1.1.0",
        "jasmine-core": "~2.6.2",
        "jasmine-spec-reporter": "~3.2.0",
        "karma": "1.7.0",
        "karma-chrome-launcher": "~2.1.1",
        "karma-cli": "~1.0.1",
        "karma-coverage-istanbul-reporter": "^1.1.0",
        "karma-jasmine": "~1.1.0",
        "karma-jasmine-html-reporter": "^0.2.2",
        "karma-junit-reporter": "1.2.0",
        "karma-remap-istanbul": "^0.2.1",
        "karma-spec-reporter": "0.0.31",
        "liferay-npm-bundler": "^2.0.0",
        "protractor": "~5.1.0",
        "rollup": "0.41.6",
        "rollup-plugin-commonjs": "^8.0.2",
        "rollup-plugin-node-resolve": "3.0.0",
        "shelljs": "0.7.7",
        "sorcery": "0.10.0",
        "ts-node": "~2.0.0",
        "tslint": "5.4.0",
        "typescript": "2.4.2",
        "webpack": "2.6.1",
        "yargs": "8.0.1"
    },
    "main": "js/angular-loader.js",
```

```
    "name": "my-npm-angular-portlet",
    "scripts": {
        "build": "tsc && babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-
INF/resources && liferay-npm-bundler"
    },
    "version": "1.0.0"
}
```

Next You can configure the portlet.

Configuring the Portlet

Follow these steps to configure your portlet:

1. Create a Component class that implements the `Portlet.class` service:

```
@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=" + MyNpmAngularPortletKeys.MyNpmAngular,
        "javax.portlet.resource-bundle=content.Language",
        "javax.portlet.security-role-ref=power-user,user"
    },
    service = Portlet.class
)
public class MyNpmAngularPortlet extends MVCPortlet {
  ...
}
```

2. If your Angular project includes CSS styling, add the following additional property to specify the location of the main CSS file:

```
"com.liferay.portlet.header-portlet-css=/css/main.css"
```

   Note that this path is relative to the resources path. If using Sass, drop the `.scss` extension in this property and use `.css` instead. For example, if your main CSS file is located in `src/main/resources/META-INF/resources/css/app.scss`, then you would have the following configuration:

```
"com.liferay.portlet.header-portlet-css=/css/app.css"
```

3. To improve code maintenance, use the NPMResolver APIs to alias your module's package name. The example below exposes the module's name as `bootstrapRequire`:

```
@Override
public void doView(
        RenderRequest renderRequest, RenderResponse renderResponse)
    throws IOException, PortletException {

    JSPackage jsPackage = _npmResolver.getJSPackage();

    renderRequest.setAttribute(
        MyNpmAngularWebKeys.BOOTSTRAP_REQUIRE,
```

```
        jsPackage.getResolvedId() + " as bootstrapRequire");

    super.doView(renderRequest, renderResponse);
}

@Reference
private NPMResolver _npmResolver;
```

4. Inside your `init.jsp`, add the following Java scriptlet to access the `bootstrapRequire` variable in your portlet's `view.jsp`:

```
<%
String bootstrapRequire = (String)renderRequest.getAttribute(
  MyNpmAngularWebKeys.BOOTSTRAP_REQUIRE
);
%>
```

Next you can learn how to render your app's component.

Rendering Your Component

Follow these steps to render your app component:

1. Inside your app's main TS file (`main.ts` for example), use the function below to render your component:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppComponent } from './app/app.component';
import { AppModule } from './app/app.module';
import { DynamicLoader } from './app/dynamic.loader';

export default function(rootId: any) {
    platformBrowserDynamic()
        .bootstrapModule(AppModule)
        .then((injector: any) => {

            // Load the bootstrap component dynamically so that we can attach it
            // to the portlet's DOM, which is different for each portlet
            // instance and, thus, cannot be determined until the page is
            // rendered (during runtime).

            // The rootId argument is passed from view.jsp where we can obtain
            // the portlet's namespace by using JSP tags.

            const dynamicLoader = new DynamicLoader(injector);

            dynamicLoader.loadComponent(AppComponent, rootId);
        });
}
```

2. In a separate file, such as `angular-loader.ts`, you can add the following configuration:

```
// Import needed polyfills before application is launched

import 'reflect-metadata';
import 'zone.js';

// Declare Liferay AMD loader
```

```
declare var Liferay: any;

// Launch application

export default function(rootId: any) {
    Liferay.Loader.require('my-npm-angular-portlet@1.0.0/js/main',
  (main: any) => {
        main.default(rootId);
    });
}
```

3. Open your `view.jsp` and add an element container to house your component. Then, add an `<aui:script>` and pass your aliased module name as the `require` attribute's value. Finally, call your module's `default` function that you exported in the previous step, and pass the container element in as the root ID. Adding the `<portlet:namespace />` to the `<div>`'s `id` ensures that it is unique to the portlet and doesn't clash with any existing elements on the page:

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

<div id="<portlet:namespace />-root"></div>

<aui:script require="<%= bootstrapRequire %>">
    bootstrapRequire.default('#<portlet:namespace />-root');
</aui:script>
```

Now you know how to use Angular in your projects!

Related Topics

Using React in Your Portlets
Using Vue in Your Portlets
Using npm in Your Portlets

87.4 Creating and Bundling JavaScript Widgets with JavaScript Tooling

The Liferay JS Generator generates JavaScript widgets for Liferay DXP. It is just one of Liferay JS Bundle Toolkit's tools.

Important: To use the Liferay JS Generator, you must have the Liferay JS Portlet Extender installed in your Liferay DXP instance. The JS Portlet Extender is a Labs application available from Liferay Marketplace for Liferay Digital Enterprise 7.1 and Liferay Portal CE 7.1. Apps designated as Labs are experimental and not supported by Liferay. They're released to accelerate the availability of useful and cutting-edge features. This status may change without notice. Please download and use Labs apps at your own discretion.

Portlets are a Java standard, so you must have a knowledge and understanding of how Java works to write one. This can be quite the hurdle for front-end developers who want to use JavaScript frameworks in their widgets. Thanks to the JS Portlet Extender and liferay-npm-bundler, developers can easily create and develop JavaScript widgets in Liferay DXP using pure JavaScript tooling.

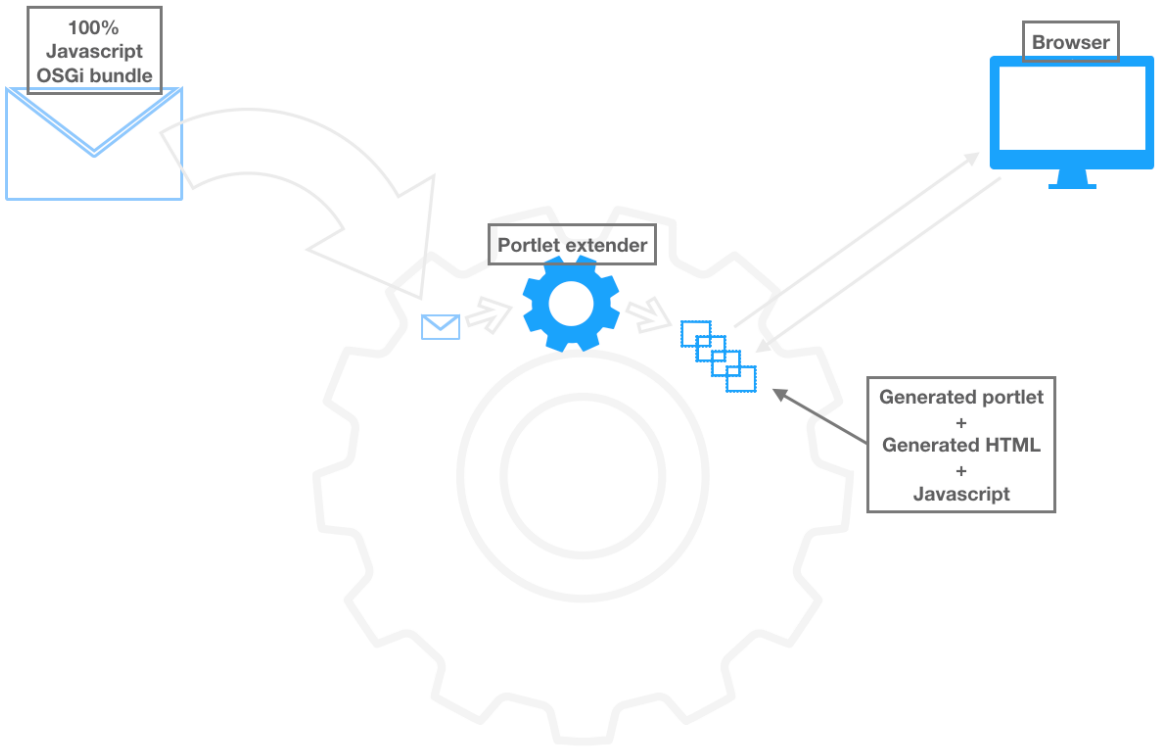


Figure 87.1: The JS Portlet Extender lets you use pure JavaScript tooling to write widgets.

Note: JavaScript Server-Side-Rendering is not supported out-of-the-box. To use JS frameworks for site rendering, you **must** set up your server-side (or search-crawler) rendering generation to support them.

This section explains how to configure these options for generated JS widgets:

- Installing the Liferay JS Generator and generating a bundle
- Configuring system and instance settings
- Localization
- Setting portlet properties
- Using translation features

87.5 Installing the Bundle Generator and Generating a Bundle

This tutorial shows how to install the Liferay JS Generator and how to use it to create JavaScript widgets.

Note: The Liferay Bundle Generator is deprecated as of v2.7.1 of the Liferay JS Toolkit. It has been renamed the Liferay JS Generator. If you're still running the Liferay Bundle Generator, we

recommend that you install the Liferay JS Generator instead at your earliest convenience, as the Liferay Bundle Generator will be removed in future versions.

Important: To use the Liferay JS Generator, you must have the Liferay JS Portlet Extender installed in your Liferay DXP instance. The JS Portlet Extender is a Labs application available from Liferay Marketplace for Liferay Digital Enterprise 7.1 and Liferay Portal CE 7.1. Apps designated as Labs are experimental and not supported by Liferay. They're released to accelerate the availability of useful and cutting-edge features. This status may change without notice. Please download and use Labs apps at your own discretion.

Follow these steps to create your JavaScript widget:

1. Install Node.js. Note that Node Package Manager (npm) is installed with this as well. You'll use npm to install the remaining dependencies and generator, and configure your npm environment.

2. Install Yeoman for the generator:

```
npm install -g yeoman
```

3. Install the Liferay JS Generator:

```
npm install -g generator-liferay-js
```

4. Run the generator with the command below, select the JavaScript widget you want to create, and answer the prompts that follow.

```
yo liferay-js
```

5. If you specified your app server information when your widget was generated, you can deploy your widget by running the command below. You can verify this by checking the value of the `liferayDir` entry in the widget's `.npmbuildrc`.

```
npm run deploy
```

Great! Now you know how to install and run the Liferay JS Generator.

Related Topics

liferay-npm-bundler

Using npm in Your Portlets

Applying Clay Styles to Your App

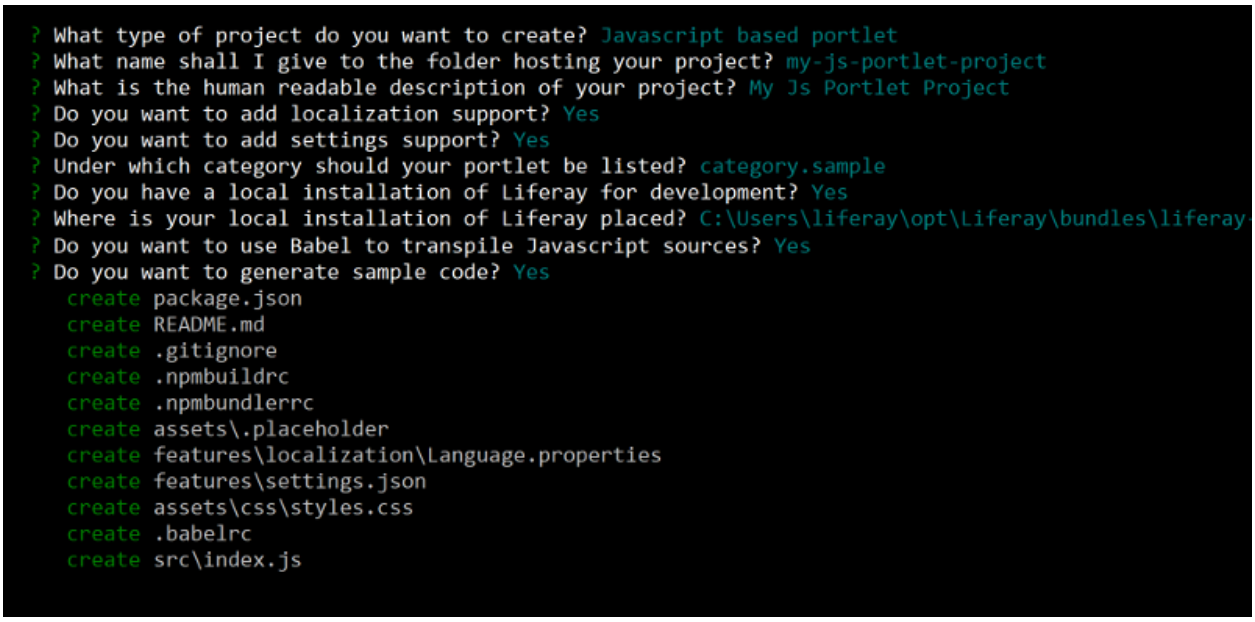


Figure 87.2: The liferay-bundle generator prompts you for widget options.

87.6 Configuring System Settings and Instance Settings for Your JavaScript Widget

As of v1.1.0 of the JS Portlet Extender, you can define configuration options for your widget. These options are passed to the widget's JavaScript entry point as the `configuration` parameter. See the main entry point's reference for more information on the entry point. Follow these steps to set system and/or portlet instance settings for your widget:

1. Add a `/features` folder in your project's root folder if it doesn't already exist.

```
**Note:** This location can be overridden with the
`create-jar.features.configuration` option in your project's `.npmbundlerrc`
file. See
[OSGi bundle configuration options](/docs/7-1/reference/-/knowledge_base/r/configuring-liferay-npm-bundler#osgi-bundle-
creation-options)
for all the available options for the bundle.
```

---

2. Create a `configuration.json` file in the `/features` folder and follow the pattern below. See the Configuration JSON reference for an explanation of each of the available options:

```
{
  "system": {
    "category": "{category identifier}",
    "name": "{name of configuration}",
    "fields": {
      "{field id 1}": {
        "type": "{field type}",
        "name": "{field name}",
```

```
        "description": "{field description}",
        "default": "{default value}",
        "options": {
          "{option id 1}": "{option name 1}",
          "{option id 2}": "{option name 2}",

          "{option id n}": "{option name n}"
        }
      },
      "{field id 2}": {},

      "{field id n}": {}
    }
  },
  "portletInstance": {
    "name": "{name of configuration}",
    "fields": {
      "{field id 1}": {
        "type": "{field type}",
        "name": "{field name}",
        "description": "{field description}",
        "default": "{default value}",
        "options": {
          "{option id 1}": "{option name 1}",
          "{option id 2}": "{option name 2}",

          "{option id n}": "{option name n}"
        }
      },
      "{field id 2}": {},

      "{field id n}": {}
    }
  }
}
```

3. Access a system setting's value or a portlet instance setting's value with the syntax `configuration.system` or `configuration.portletInstance` respectively. For instance, to retrieve the `{field id 1}` system setting's value, you would use `configuration.system.{field id 1}`. Note that all fields are passed as strings no matter what type they declare in their descriptor.

Awesome! Now you know how to configure system settings and portlet instance settings for your widget.

Related Topics

- Localizing Your Widget
- Using Translation Features in Your JavaScript Widget
- Configuring Portlet Properties for Your JavaScript Widget

87.7 Localizing Your Widget

Follow the steps below to learn how to localize your widget:

1. If you didn't choose to use localization when you generated the bundle, follow this step to enable it in your bundle now, otherwise you can skip this step. Create a `/features/localization` folder in your project and add a `Language.properties` file to it.

Add a `create-jar.features.localization` key to your `.npmbuildrc` file that points to the `Language.properties` file. An example configuration is shown below:

{ “create-jar”: { “output-dir”: “dist”, “features”: { “js-extender”: true, “web-context”: “/my-test-js-widget”, “localization”: “features/localization/Language”, “settings”: “features/settings.json” } }, ... }

```
**Note:** The default file path is shown above. You can update this value,
if you want to place your `Language.properties` file in a different
location.
```

---

2. Configure the `Language.properties` file and provide the localized property files (e.g. `Language_[locale].properti` with the language keys for each available translation. The *JavaScript based widget* configuration is shown below:

   javax.portlet.title.my_js_portlet_project=My JS widget Project porlet-namespace=Porlet Namespace context-path=Context Path portlet-element-id=Portlet Element Id configuration=Configuration fruit=Favourite fruit fruit-help=Choose the fruit you like the most an-orange=An orange a-pear=A pear an-apple=An apple

3. Retrieve a language key’s localized value in JavaScript with the `Liferay.Language.get('key')` method.

Great! Now you know how to localize your widget!

Related Topics

- Configuring System Settings and Instance Settings for Your JavaScript Widget
- Using Translation Features in Your JavaScript Widget
- Configuring Portlet Properties for Your JavaScript Widget

87.8 Configuring Portlet Properties for Your JS Widget

Follow these steps to configure your portlet’s properties:

1. Open your generated JavaScript widget’s `package.json` file.
2. Set the properties under the `portlet` entry. Note that these are the same properties you would define in the Java `@Component` annotation of a portlet, as defined in the liferay-portlet-app_7_2_0.dtd. An example configuration is shown below:

```
"portlet": {
    "com.liferay.portlet.display-category": "category.sample",
    "com.liferay.portlet.header-portlet-css": "/css/styles.css",
    "com.liferay.portlet.instanceable": true,
    "javax.portlet.name": "my_js_portlet_project",
    "javax.portlet.security-role-ref": "power-user,user",
    "javax.portlet.resource-bundle": "content.Language"
},
```

3. Deploy your bundle to apply the changes.

Great! Now you know how to configure your JavaScript portlet's properties.

Related Topics

- Configuring System Settings and Instance Settings for Your JavaScript Widget
- Localizing Your Widget
- Using Translation Features in Your JavaScript Widget

87.9 Using Translation Features in Your Widget

By default, the Liferay JS Generator creates an empty configuration for translation. The translate script instructs the user how to add new supported locales or configure the credentials when it is run. The translate target reads the supported locales you have defined in the `supportedLocales` key of your `.npmbuildrc` file and checks your `*language.properties` files to make sure they match.

Note: To use the translation features, you must have a Microsoft Translator key. Provide your credentials through either the `translatorTextKey` variable in your `.npmbuildrc` file, or provide them in the `TRANSLATOR_TEXT_KEY` environment variable.

Follow these steps to add a new supported locale and automatically create a language properties file for it with translations:

1. Add the locale to the `supportedLocales` array in your `.npmbuildrc` file.
2. Run the translate target with the command below:

   ```
   npm run translate
   ```

3. The translate target automatically creates a language properties file for each new **supported** locale with translations for your language keys. It also warns about locales that are not supported, but have a `*language.properties` file.

Great! Now you know how to use the Liferay JS Generator's translation features in your app.

Related Topics

- Configuring System Settings and Instance Settings for Your JavaScript Widget
- Localizing Your Widget
- Configuring Portlet Properties for Your JavaScript Widget

CHAPTER 88

FRONT-END TAGLIBS

You have access to a powerful set of taglibs for creating commonly used UI components in your apps, themes, and web content. The following taglibs are covered in this section of tutorials:

- AUI: lets you create common UI components such as forms, buttons, and more.
- Chart: visualizes data. You can create bar charts, line charts, scatter charts, spline charts, and much more.
- Clay: lets you add Clay components, such as alerts, buttons, drop-down menus, form elements, and more to your apps.
- Frontend: Lets you create UI components commonly used throughout Portal's apps, such as add menus, cards, management bars, and more.
- Liferay UI: lets you create common UI components such as icons, tabs, and more.

Note: Each taglib is available as a FreeMarker macro, except for the Chart taglib. The Chart taglib is **not** available as a FreeMarker macro. The tutorials in this section provide the proper syntax to use for each macro. See the FreeMarker Taglib Mappings reference for a complete list of the available FreeMarker taglib macros.

In this section of tutorials, you'll learn how to use taglibs to build awesome user interfaces for your apps!

CHAPTER 89

USING THE LIFERAY UI TAGLIB

The Liferay UI tag library provides tags that implement commonly used UI components. These tags make your markup consistent, responsive, and accessible.

You can find a list of the available Liferay UI taglibs in the Liferay UI taglibdocs. 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. You'll find the full markup generated by the tags in their JSPs in their Liferay Github Repo folders.

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 />
```

This section of tutorials covers how to create UI components with the Liferay UI taglibs. Each tutorial contains code examples along with a screenshot of the resulting UI.

89.1 Liferay UI Icons

The Liferay UI taglibs provide several icons you can include in your apps. To add an icon to your app, use the `liferay-ui:icon` tag and specify the icon with either the `icon`, `iconCssClass`, or `image` attribute. An example of each use case is shown below.

The `image` attribute specifies Liferay UI icons to use (as defined in the Unstyled theme's `images/common` folder). Here's an example configuration for a JSP:

```
<div class="col-md-3">
    <liferay-ui:icon image="subscribe" />

    <span class="ml-2">Subscribe</span>
</div>
```

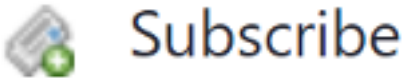


Figure 89.1: Use the image attribute to use a theme icon.

The Liferay UI taglib also exposes language flag icons. To use a language flag icon, provide the `../language/` relative path before the icon's name. Below is an example snippet from the Web Content Search portlet that displays the current language's flag along with a localized message:

```
<liferay-ui:icon
    image='<%= "../language/" + languageId %>'
    message='<%= LanguageUtil.format(
        request,
        "this-result-comes-from-the-x-version-of-this-content",
        snippetLocale.getDisplayLanguage(locale),
        false
    ) %>'
/>
```

You can achieve the same result in FreeMarker with the following code that uses the available `init.ftl` variables and Liferay DXP macros:

```
<#assign flag_message>
    <@liferay.language_format
        arguments=language
        key="this-result-comes-from-the-x-version-of-this-content"
    />
</#assign>

<@liferay_ui["icon"]
    image="../language/${language_id}"
    message=flag_message
/>
```

The full list of available icons is shown in the figures below:
The `icon` attribute specifies Font Awesome icons to use:

```
<liferay-ui:icon icon="angle-down" />
```

The `iconCssClass` attribute specifies a glyphicon to use:

```
<liferay-ui:icon
    iconCssClass="icon-remove-sign"
    label="<%= true %>"
    message="unsubscribe"
    url="<%= unsubscribeURL %>"
/>
```

The examples above use some of the icon's available attributes. See the Icon taglibdocs for the full list.

Related Topics

Clay Icons
Liferay UI Icon Lists
Liferay UI Icon Menus

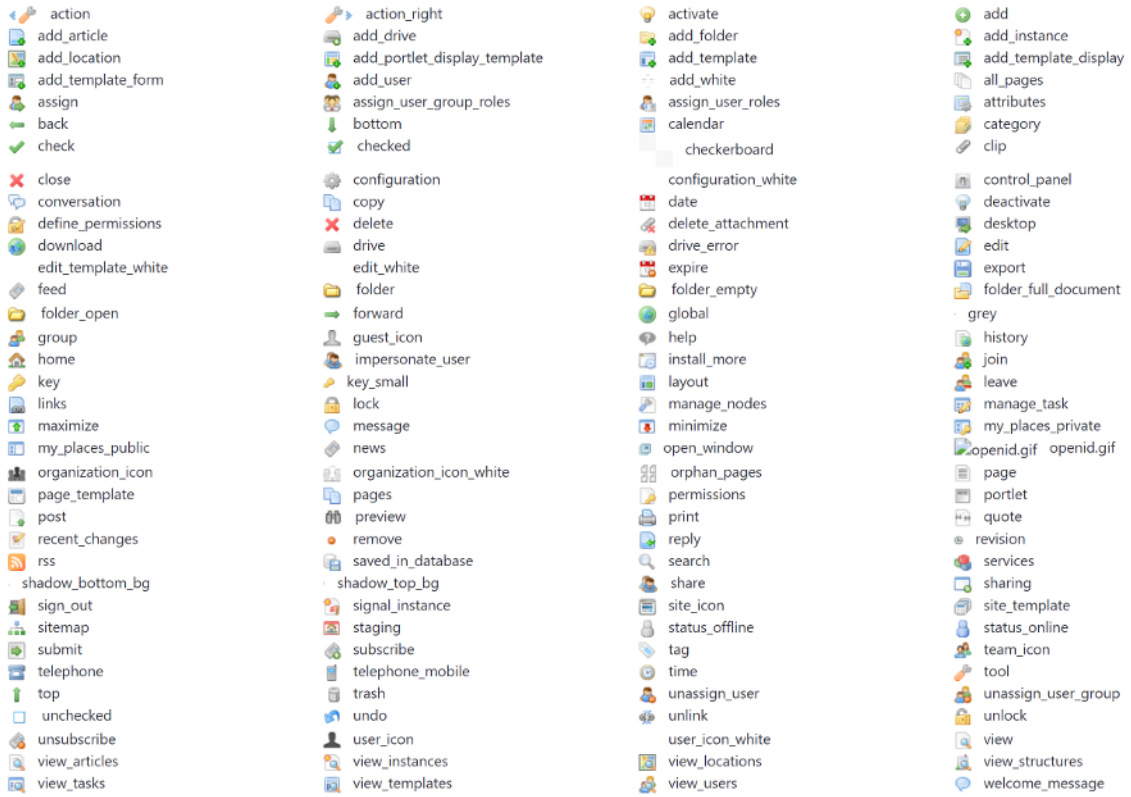


Figure 89.2: The Liferay UI taglib offers multiple icons for use in your app.

Figure 89.3: Liferay UI icons can be configured based on language.

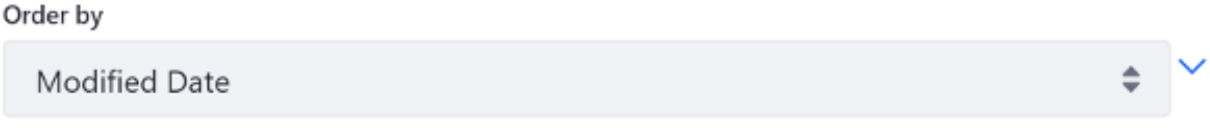


Figure 89.4: You can use the icon attribute to include Font Awesome icons in your app.

Figure 89.5: You can use Font Awesome icons in your app.

89.2 Liferay UI Icon Lists

An icon list displays icons in a horizontal list, instead of in a pop-up navigation menu like an icon menu. You can see an example of an icon list menu in a message board thread. The thread's actions are visible at all times for administrators:

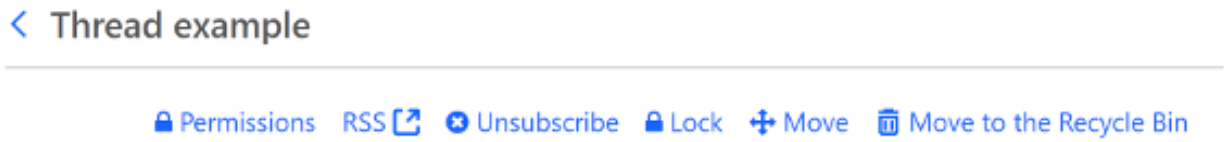


Figure 89.6: Icon lists display an app's actions at all times.

Create the list menu with the `liferay-ui:icon-list` tag and nest icons for each list item, as shown below:

```
<div class="thread-actions">
    <liferay-ui:icon-list>

        <liferay-ui:icon
        iconCssClass="icon-lock"
        message="permissions"
        method="get"
        url="<%= permissionsURL %>"
        useDialog="<%= true %>"
        />

        <liferay-rss:rss
        delta="<%= rssDelta %>"
        displayStyle="<%= rssDisplayStyle %>"
        feedType="<%= rssFeedType %>"
        url="<%= MBRSSUtil.getRSSURL(plid, 0, message.getThreadId(), 0, themeDisplay) %>"
        />

        <liferay-ui:icon
        iconCssClass="icon-remove-sign"
        message="unsubscribe"
        url="<%= unsubscribeURL %>"
        />

        <liferay-ui:icon
        iconCssClass="icon-lock"
        message="lock"
        url="<%= lockThreadURL %>"
        />
```

```
        <liferay-ui:icon
        iconCssClass="icon-move"
        message="move"
        url="<%= editThreadURL %>"
        />

        <liferay-ui:icon-delete
        showIcon="<%= true %>"
        trash="<%= trashHelper.isTrashEnabled(themeDisplay.getScopeGroupId()) %>"
        url="<%= deleteURL %>"
        />
    </liferay-ui:icon-list>
</div>
```

See the Icon List taglibdocs for the full list of available attributes.

Related Topics

Clay Icons
    Liferay UI Icon Menus
    Liferay UI Icons

89.3 Liferay UI Icon Menus

You can add a pop-up navigation menu to your app with the `liferay-ui:icon-menu` tag. Icon menus display menu options when needed, storing them away in a collapsed menu when they're not. This keeps the UI clean and uncluttered. Just as with an icon list, you nest icons for each navigation item. You can see an example of a icon menu in a site's actions menu in the My Sites portlet:

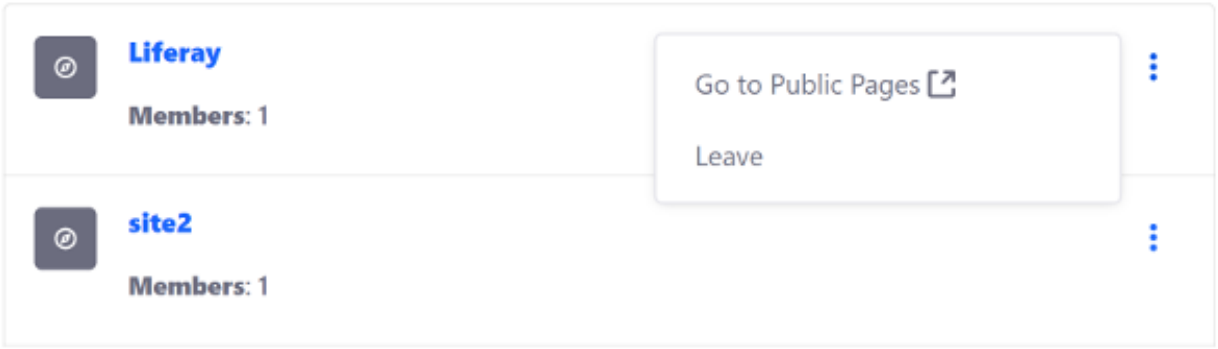


Figure 89.7: Setting up an icon menu is a piece of cake.

Example JSP configuration:

```
<liferay-ui:icon-menu
    direction="left-side"
    icon="<%= StringPool.BLANK %>"
    markupView="lexicon"
    message="<%= StringPool.BLANK %>"
    showWhenSingleIcon="<%= true %>"
>
```

```
            <liferay-ui:icon
                message="go-to-public-pages"
                target="_blank"
                url="<%= group.getDisplayURL(themeDisplay, false) %>"
            />

            <liferay-ui:icon
                message="leave"
                url="<%= leaveURL %>"
            />

</liferay-ui:icon-menu>
```

Note that the `url` attribute is required for icons to render properly. See the Icon Menu taglibdocs for the full list of attributes.

Related Topics

Clay Icons
    Liferay UI Icon Lists
    Liferay UI Icons

89.4 Liferay UI Tabs

Tabs create dividers that organize content into individual sections. Content can be embedded or included from another JSP.

To add tabs to your app, use the `<liferay-ui:tabs>` tag and specify each tab's name as a comma-separated list for the `names` attribute. For example, three tabs named `tab1`, `tab2`, and `tab3`, look like this in the JSP:

```
<liferay-ui:tabs names="tab1,tab2,tab3">

</liferay-ui:tabs>
```

Each tab requires a corresponding section to display content. Nest `liferay-ui:section` tags for each of the tabs. Within each section, you can add HTML content or add content indirectly by including content from another JSP (via the `<%@ includefile="filepath"%>` directive). The example snippet below is from the Calendar portlet's `configuration.jsp`:

```
<liferay-ui:tabs
    names='<%= "user-settings,display-settings,rss" %>'
    param="tabs2"
    refresh="<%= false %>"
    type="tabs nav-tabs-default"
>
    <liferay-ui:section>
        <%@ include file="/configuration/user_settings.jspf" %>
    </liferay-ui:section>

    <liferay-ui:section>
        <%@ include file="/configuration/display_settings.jspf" %>
    </liferay-ui:section>

    <liferay-ui:section>
        <%@ include file="/configuration/rss.jspf" %>
    </liferay-ui:section>
</liferay-ui:tabs>
```

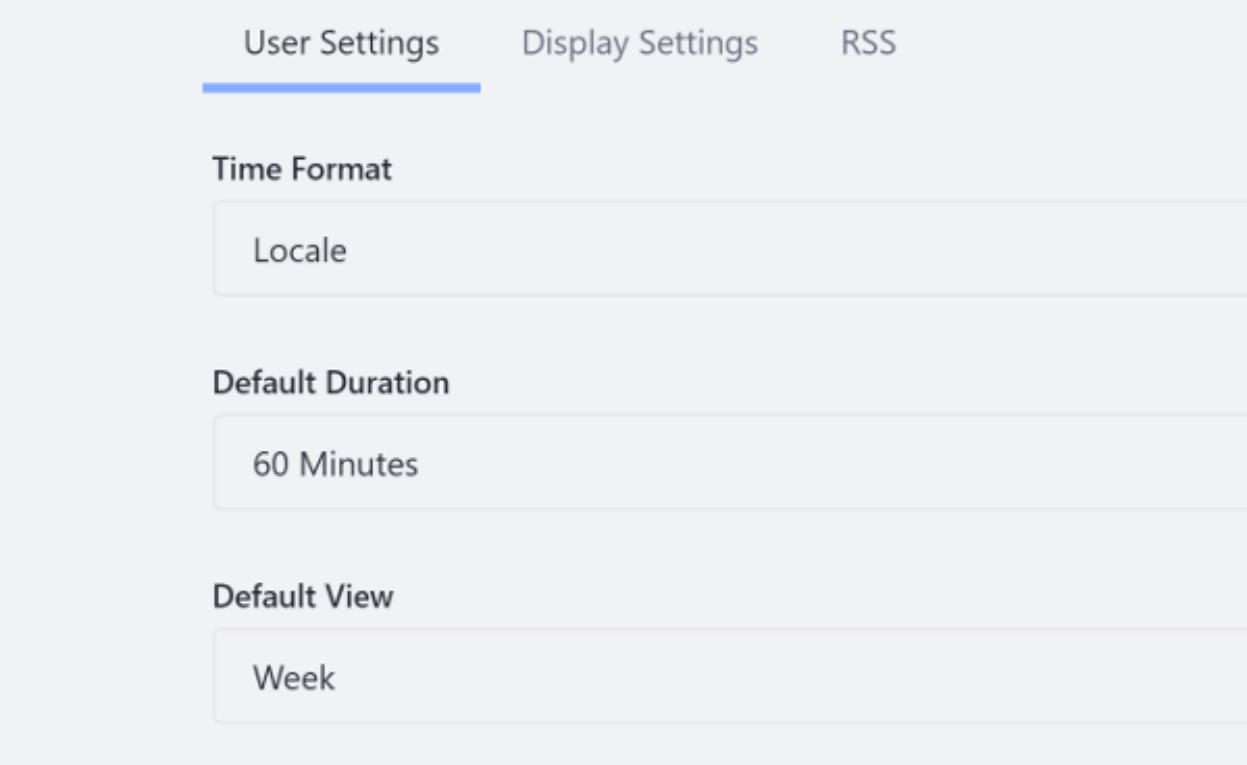


Figure 89.8: Tabs are a useful way to organize configuration options into individual sections within the same UI.

The example above uses some of the tab's available attributes. See the Tabs taglibdocs for the full list of attributes.

Related Topics

Clay Navigation Bars
    Clay Dropdown Menus and Action Menus
    Liferay UI Icon Help

89.5 Liferay UI Icon Help

The icon help tag lets you communicate additional information to your users in an unobtrusive way. It renders as an iconic question mark that provides more information through a pop-up tooltip on mouse over. You can see an example of this in the Control Panel:

Note: If you have installed a custom theme you may also need to add the following imports to your `view.jsp` to make `liferay-ui:icon-help` tag work:

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

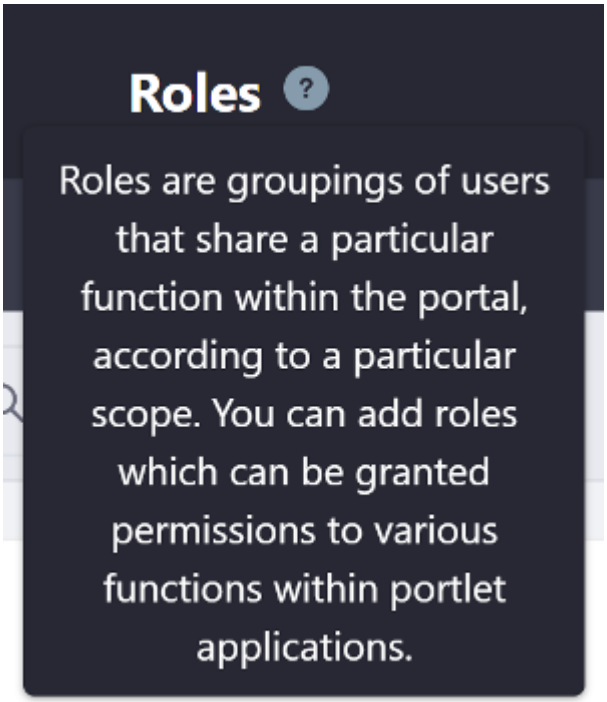


Figure 89.9: Here's an example of the icon help tag.

Add the `<liferay-ui:icon-help/>` tag next to the UI that needs tooltip information. Define the informational text with the required `message` attribute. Below is an example snippet for one of the Server Administration's clean up actions:

```
<h5>
    <liferay-ui:message key="clean-up-permissions" />
    <liferay-ui:icon-help message="clean-up-permissions-help" />
</h5>
```

Note that the message is supplied via a language key. While you can use a string for the tooltip's message for testing purposes, a language key is considered best practice and should be used in production.

Related Topics

Clay Badges
    Clay Stickers
    Liferay UI Icon Menus

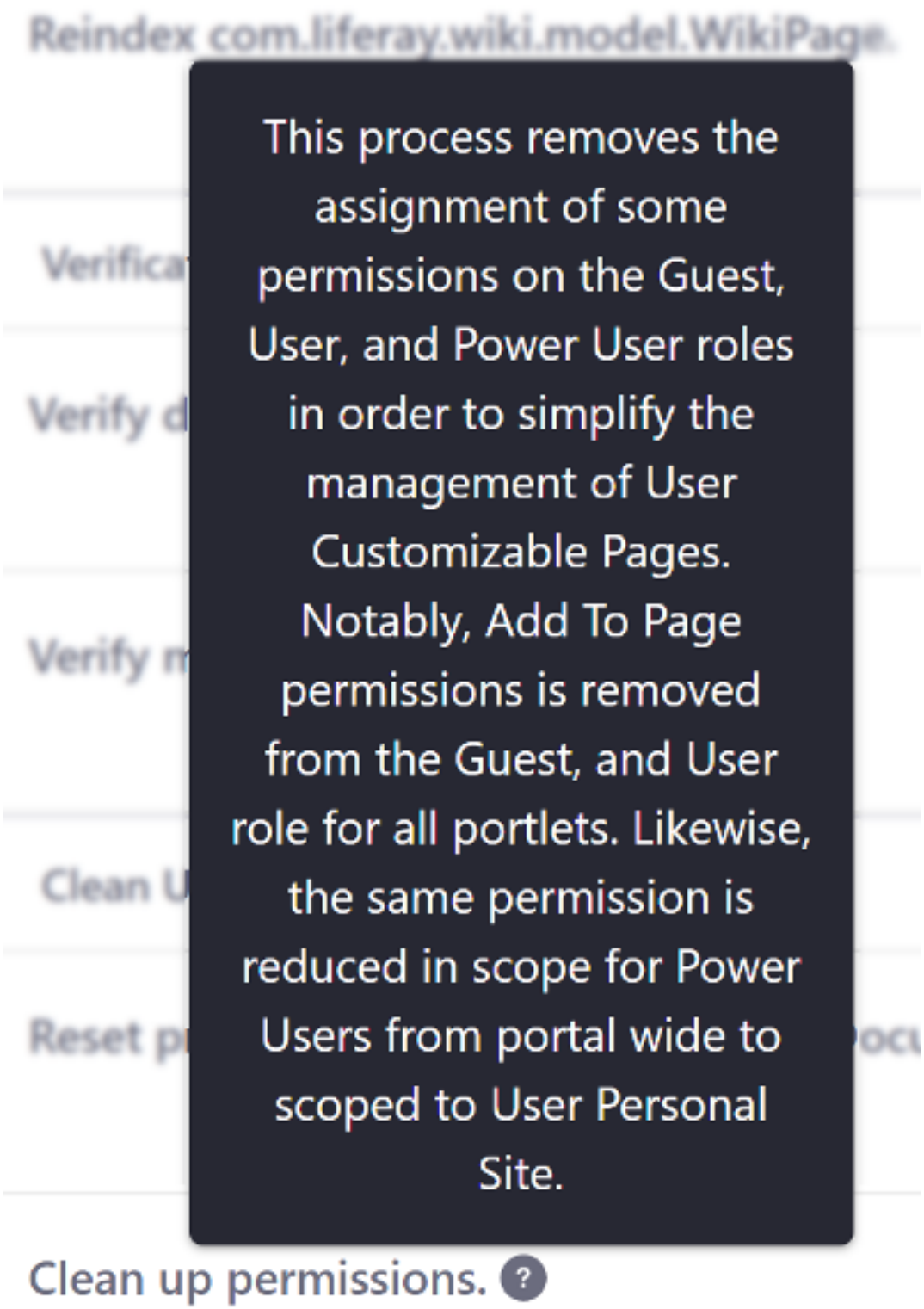


Figure 89.10: help icons are used throughout the Control Panel.

CHAPTER 90

USING LIFERAY FRONT-END TAGLIBS IN YOUR PORTLET

The Liferay Front-end tag library provides a set of tags for creating common front-end UI components in your app.

To use the Front-end taglib in you apps, add the following declaration to your JSP:

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

The Liferay Front-end taglib is also available via a macro for your FreeMarker theme templates and web content templates. Follow this syntax:

```
<@liferay_frontend["tag-name"] attribute="string value" attribute=10 />
```

The following Front-end UI components are covered in this section of tutorials:

- Add Menu
- Cards
- Info Bar
- Management Bar

The tutorials in this section cover how to create these components with the Front-end taglibs. Each tutorial contains a set of examples along with a screenshot of the resulting UI.

90.1 Liferay Front-end Add Menu

The add menu tag creates an add menu button for one or multiple items. It's used for actions that add entities (e.g. a new blog entry), and is part of the Management Bar. Use the `<liferay-frontend:add-menu>` tag to create the add menu and nest a `<liferay-frontend:add-menu-item>` tag for each item.

Note: This pattern is deprecated as of 7.0. We recommend that you use the Clay Management Toolbar's creation menu pattern instead.

When the menu has one item, the button triggers the item's action as shown in the example below for the Blogs Admin App:

```
<liferay-frontend:management-bar-buttons>
    ...
    <liferay-frontend:add-menu
      inline="<%= true %>"
    >
      <liferay-frontend:add-menu-item
        title='<%= LanguageUtil.get(request, "add-blog-entry") %>'
        url="<%= addEntryURL %>"
      />
    </liferay-frontend:add-menu>

</liferay-frontend:management-bar-buttons>
```

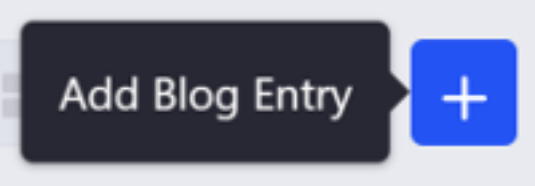


Figure 90.1: The add button pattern consists of an `add-menu` tag and at least one `add-menu-item` tag.

When the menu has multiple items, they display in a pop-up menu. For example, the Message Boards Admin application has the configuration below:

```
<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>
```

The examples above use some of the available attributes. See the add menu and add menu item taglibdocs for the full list of available attributes for the tags.

Related Topics

Liferay Frontend Cards
    Liferay Frontend Info Bar
    Liferay Frontend Management Bar

90.2 Liferay Front-end Cards

If you have data you want to compare that's heavy on image usage, cards are the component for the job. Cards visually represent data in a minimal and compact format. Use them for images, document libraries, user profiles, and more. There are four main types of Cards covered in this tutorial:

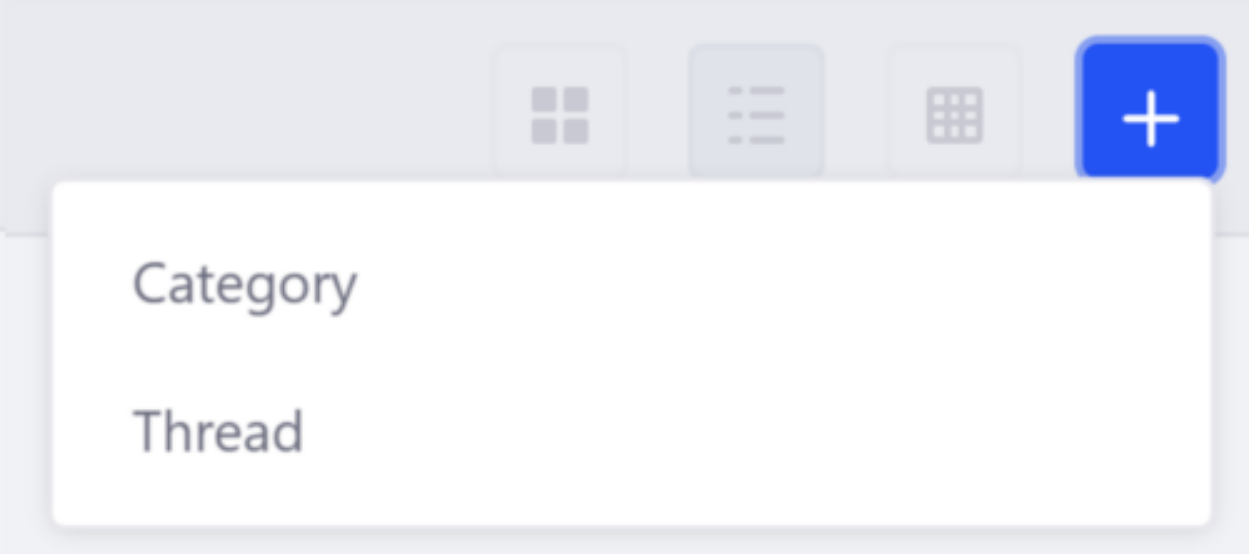


Figure 90.2: The add button pattern consists of an `add-menu` tag and at least one `add-menu-item` tag.

- Horizontal Cards
- Icon Cards
- Vertical Cards
- User Cards

Examples of each card are shown below.

Horizontal Card

Horizontal cards are used primarily to display documents, such as files and folders. An example configuration is shown below:

```
<liferay-frontend:horizontal-card
    text="Documents"
  url="/docs/7-1/tutorials/-/knowledge_base/t/clay-icons"
>
    <liferay-frontend:horizontal-card-col>
                <liferay-frontend:horizontal-card-icon
                    icon="folder"
                />
    </liferay-frontend:horizontal-card-col>
</liferay-frontend:horizontal-card>
```

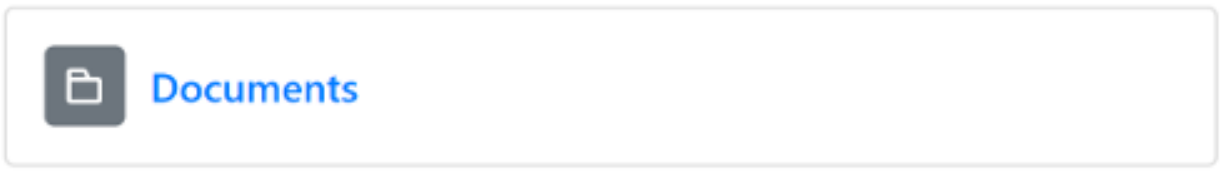


Figure 90.3: Horizontal cards are perfect to display files and documents.

The `<liferay-frontend:horizontal-card-icon>` tag uses Clay Icons for its `icon` attribute.

Icon Vertical Card

Icon vertical cards, as the name suggests, are cards that display information in a vertical format that emphasizes an icon. These cards show content that doesn't have an associated image. Instead, an icon representing the type of content is displayed. The example snippet below displays information for a web content article:

```
<liferay-frontend:icon-vertical-card
  cssClass="article-preview-content"
  icon="web-content"
  title="<%= title %>"
>
  <liferay-frontend:vertical-card-sticker-bottom>
    <liferay-ui:user-portrait
      cssClass="sticker sticker-bottom"
      userId="<%= assetRenderer.getUserId() %>"
    />
  </liferay-frontend:vertical-card-sticker-bottom>

  <liferay-frontend:vertical-card-footer>
    <aui:workflow-status
      markupView="lexicon"
      showIcon="<%= false %>"
      showLabel="<%= false %>"
      status="<%= article.getStatus() %>"
    />
  </liferay-frontend:vertical-card-footer>
</liferay-frontend:icon-vertical-card>
```

Vertical Card

Vertical cards display information in a vertical card format, as opposed to a horizontal format. If the content has an associated image (like a blog header image) you can use a vertical card to display the image. If there is no associated image, you can use an icon vertical card to represent the content's type instead (e.g. a PDF file). The example below displays a vertical card for a web content article when an image preview is available:

```
<liferay-frontend:vertical-card
  cssClass="article-preview-content"
  imageUrl="<%= articleImageURL %>"
  title="<%= title %>"
>
  <liferay-frontend:vertical-card-sticker-bottom>
    <liferay-ui:user-portrait
      cssClass="sticker sticker-bottom"
      userId="<%= assetRenderer.getUserId() %>"
    />
  </liferay-frontend:vertical-card-sticker-bottom>

  <liferay-frontend:vertical-card-footer>
    <aui:workflow-status
      markupView="lexicon"
      showIcon="<%= false %>"
      showLabel="<%= false %>"
      status="<%= article.getStatus() %>"
    />
  </liferay-frontend:vertical-card-footer>
</liferay-frontend:vertical-card>
```

Figure 90.4: Vertical icon cards are perfect to display an entity selection, such as a web content article.

HTML Vertical Card

The HTML Vertical card lets you display custom HTML in the header of the vertical card. The example below embeds a video:

```
<liferay-util:buffer var = "customThumbnailHtml">
    <div class="embed-responsive embed-responsive-16by9">
      <iframe class="embed-responsive-item"
    src="https://www.youtube.com/embed/8Bg9jPJpGOM?rel=0"
    allowfullscreen></iframe>
    </div>
</liferay-util:buffer>

<div class="container">
  <div class="row">
    <div class="col-md-4">
      <liferay-frontend:html-vertical-card
        html="<%= customThumbnailHtml %>"
```

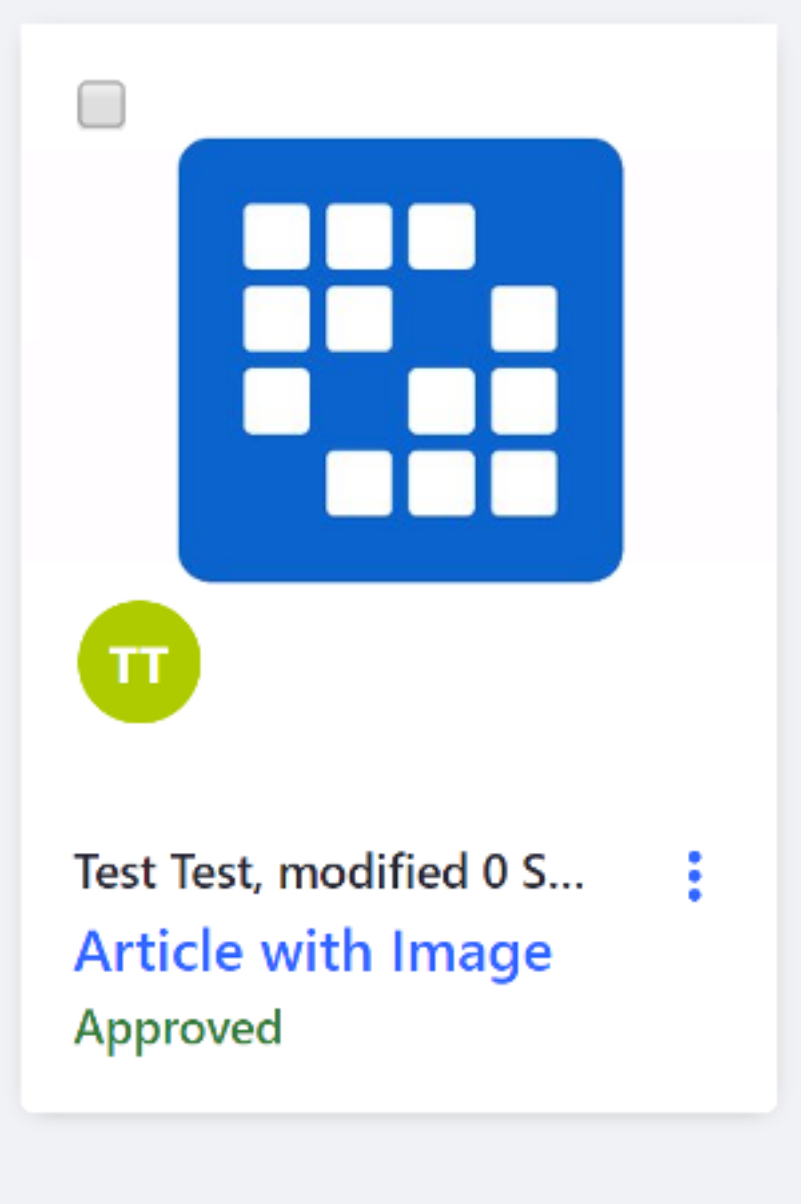


Figure 90.5: Vertical cards are perfect to display files and documents.

```
        title="My Video"
      >
      </liferay-frontend:html-vertical-card>
    </div>
  </div>
</div>
```

Figure 90.6: Html vertical cards let you display custom HTML in the card's header.

User Vertical Card

The User Vertical card displays user profile selections in the icon view of the Management Bar. Below is an example snippet from the User Admin portlet:

```
<liferay-frontend:user-vertical-card
  actionJsp="/membership_request_action.jsp"
  actionJspServletContext="<%= application %>"
  resultRow="<%= row %>"
  subtitle="<%= membershipRequestUser.getEmailAddress() %>"
  title="<%= HtmlUtil.escape(membershipRequestUser.getFullName()) %>"
  userId="<%= membershipRequest.getUserId() %>"
>
  <liferay-frontend:vertical-card-header>
    <liferay-ui:message
      arguments="<%= LanguageUtil.getTimeDescription(
      request,
      System.currentTimeMillis() - membershipRequest.getCreateDate().getTime(),
      true) %>"
      key="x-ago"
      translateArguments="<%= false %>"
    />
  </liferay-frontend:vertical-card-header>
</liferay-frontend:user-vertical-card>
```

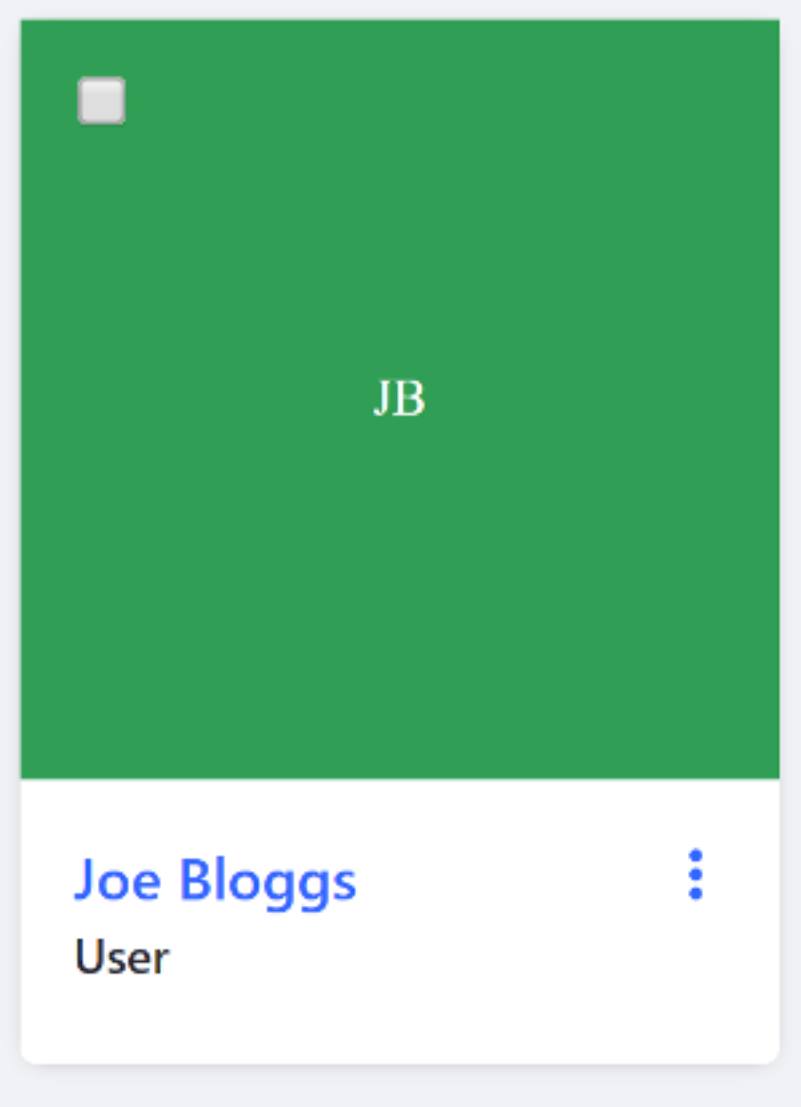


Figure 90.7: User vertical cards are perfect to display files and documents.

Related Topics

Liferay Front-end Add Menu
Liferay Front-end Info Bar
Liferay Front-end Management Bar

90.3 Liferay Front-end Info Bar

An info bar provides a button that toggles the visibility of additional sidebar information. This is perfect for providing more detailed metadata for a search result, such as the file size, type, URL, etc.

The configuration has two key parts: the info bar—and buttons—and the sidebar panel.

Info bar:

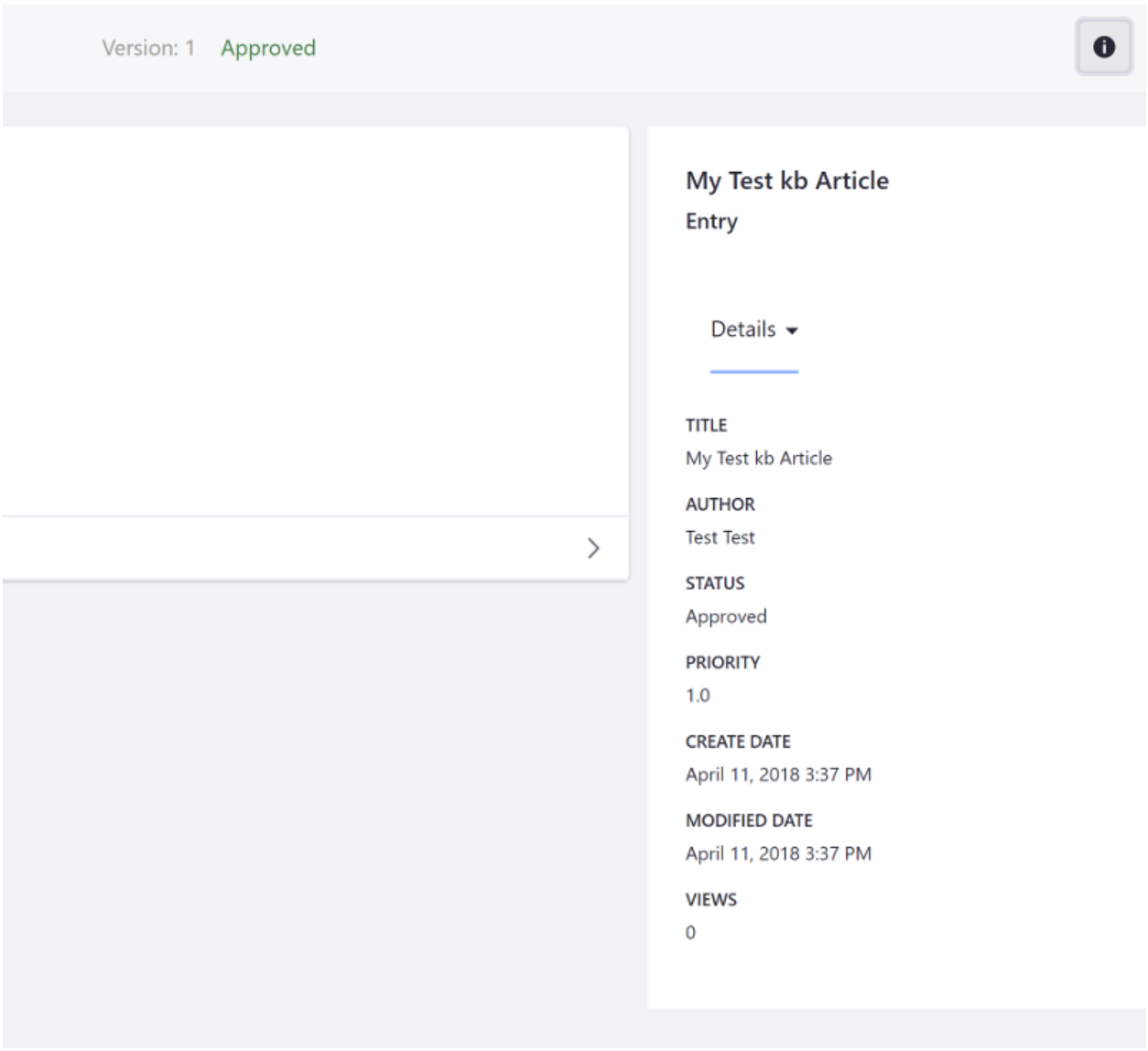


Figure 90.8: The info bar tags create a sidebar panel toggler that reveals additional info.

```
<liferay-frontend:info-bar>
  <liferay-frontend:info-bar-buttons>
    <liferay-frontend:info-bar-sidenav-toggler-button
      icon="info-circle"
      label="my info"
    />
  </liferay-frontend:info-bar-buttons>
</liferay-frontend:info-bar>
```

The `<liferay-frontend:info-bar-sidenav-toggler-button>` tag uses Clay Icons for the `icon` attribute.

Sidebar panel:

```
<div class="closed container-fluid-1280 sidenav-container sidenav-right" id="<portlet:namespace />infoPanelId">
    <liferay-frontend:sidebar-panel>
      <div>
      <h2>sidebar content</h2>
      <p>Here is some content</p>
      </div>
    </liferay-frontend:sidebar-panel>
</div>
```

Note that the sidebar panel's wrapper `<div>` has the classes `closed` and `sidenav-right`. The info button toggles the classes `open` and `closed`, showing and hiding the sidebar panel. The `sidenav-right` class specifies that the panel should open on the right.

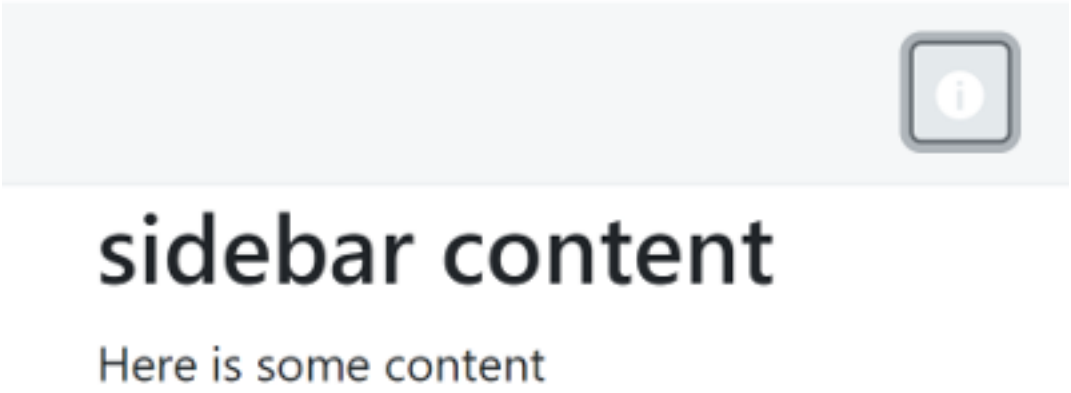


Figure 90.9: The info bar tags create a sidebar panel toggler that reveals additional info.

The examples above use some of the available attributes. See the info bar, info bar buttons, info bar sidenav toggler button, and sidebar panel taglibdocs for the full list of available attributes for the tags.

Related Topics

Liferay Front-end Add Menu
Liferay Front-end Cards
Liferay Front-end Management Bar

CHAPTER 91

LIFERAY FRONT-END MANAGEMENT BAR

The Management Bar gives administrators control over search container results. It lets you filter, sort, and choose a display style for search results, so you can quickly identify the document, web content, asset entry, or whatever you're looking for in your app. The Management Bar is fully customizable, so you can implement all the controls, or just the ones your app requires.

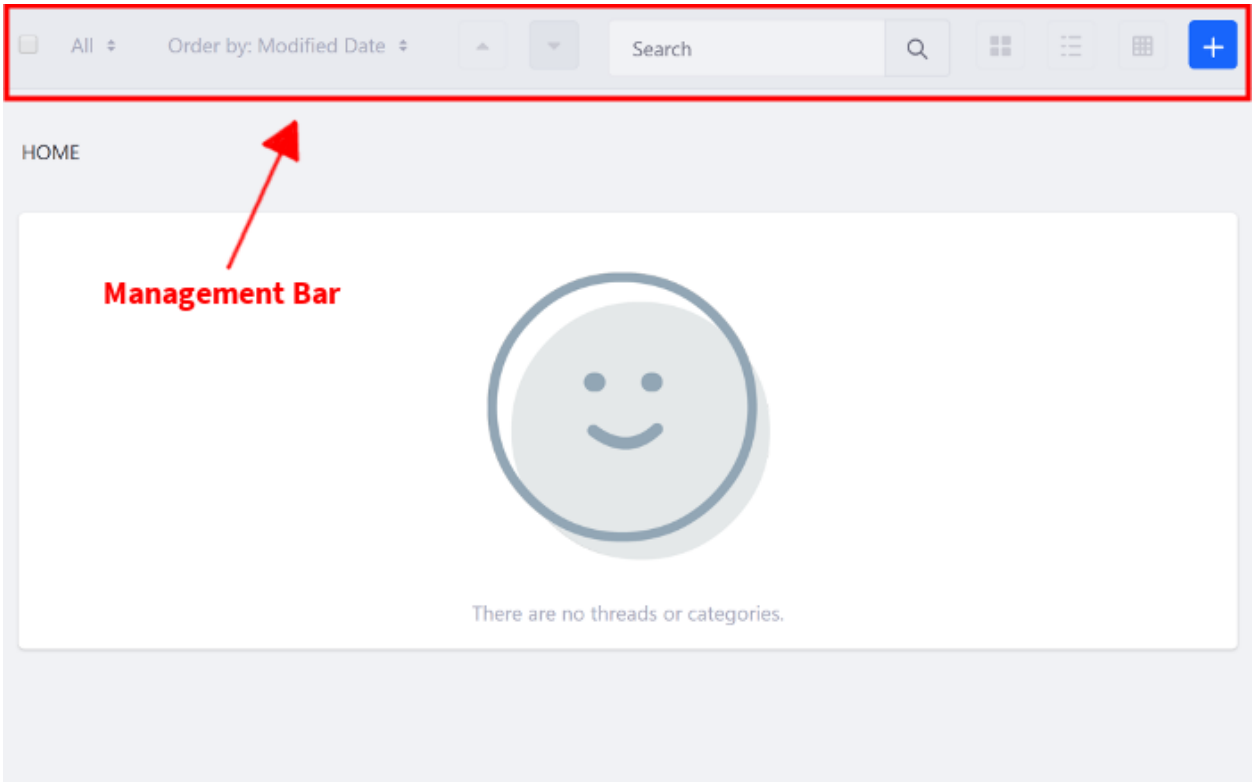


Figure 91.1: The Management Bar lets the user customize how the app displays content.

Note: The Liferay Front-end Management Bar is deprecated as of 7.0. We recommend that you

use the Clay Management Toolbar instead.

The Management Bar has 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:

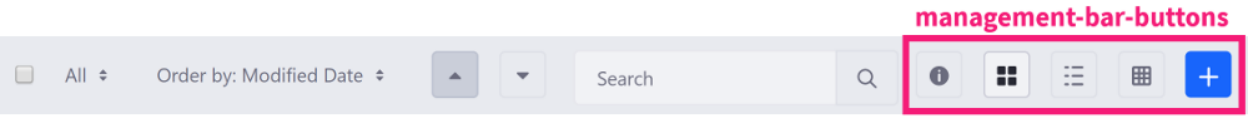


Figure 91.2: The `management-bar-buttons` tag contains the Management Bar's main buttons.

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.

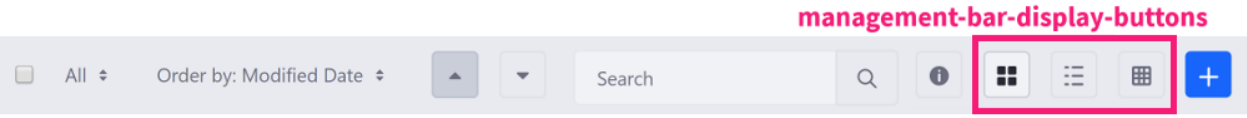


Figure 91.3: The `management-bar-display-buttons` tag contains the content's display options.

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.

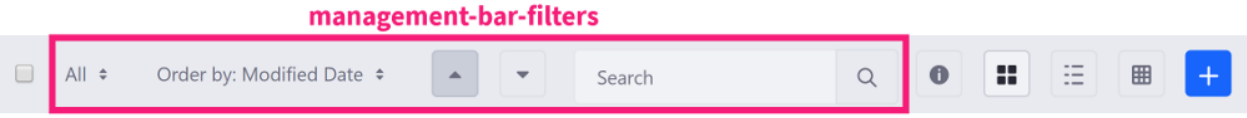


Figure 91.4: The `management-bar-filters` tag contains the content filtering options.

Finally, the `<liferay-frontend:management-bar-action-buttons>` tag wraps the actions that you can execute over selected items. You can select multiple items between pages. The management bar keeps track of the number of selected items for you.

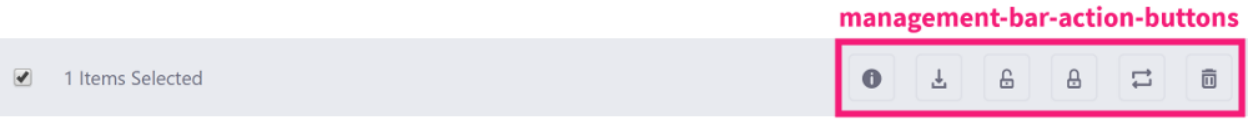


Figure 91.5: The management bar keeps track of the items selected and displays the actions to execute on them.

For example, here's the Management Bar configuration in the Trash app:

```
<liferay-frontend:management-bar
   includeCheckBox="<%= true %>"
   searchContainerId="trash"
>
   <liferay-frontend:management-bar-buttons>
       <liferay-frontend:management-bar-sidenav-toggler-button />

       <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() %>"
           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>
</liferay-frontend:management-bar>
```

91.1 Including Actions in the Management Bar

While an actions menu is typically included with each search container result, you can also include these actions in the management bar. This keeps everything organized within the same UI. This update adds a checkbox next to each search container result, as well as adds one in the management bar itself to select all results. The actions are displayed when a checkbox is checked—individual or select all—and hidden from view otherwise.

Follow these steps to include actions in your management bar:

1. Update the `<liferay-frontend:management-bar>` tag to include the checkbox and provide the search container's ID:

```
<liferay-frontend:management-bar
    includeCheckBox="<%= true %>"
    searchContainerId="mySearchContainerId"
>
```

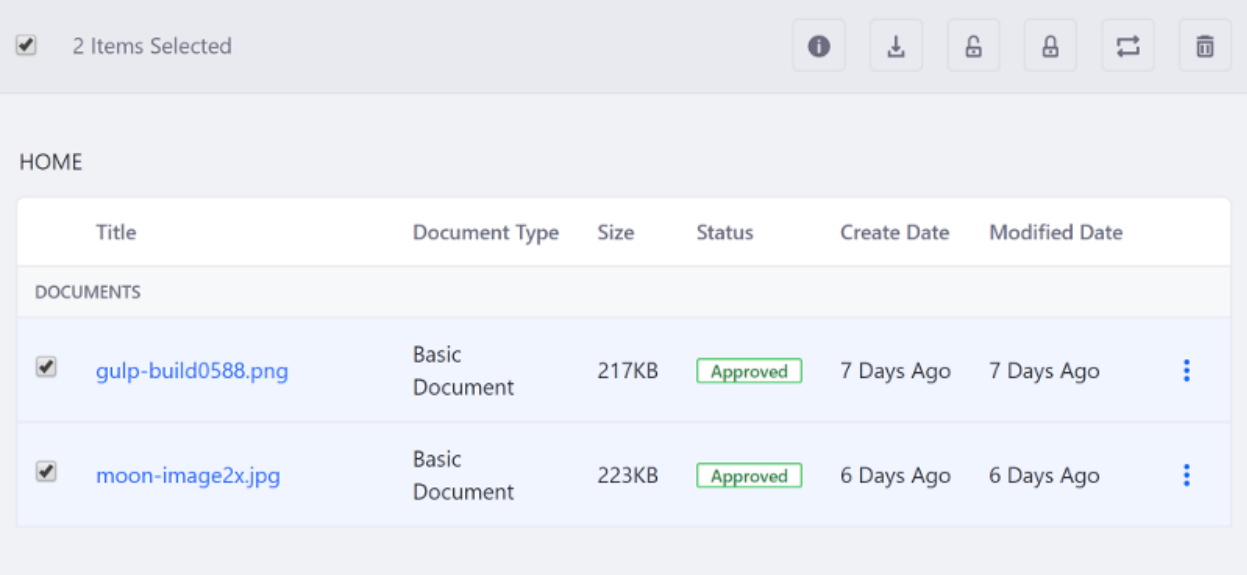


Figure 91.6: You can select individual results or all results at once.

2. After the closing `</liferay-frontend:management-bar-filters>` tag, add the `<liferay-frontend:management-bar-action-buttons>` tags:

```
<liferay-frontend:management-bar-action-buttons>

</liferay-frontend:management-bar-action-buttons>
```

3. Use the available management bar button taglibs (e.g. `management-bar-button`) to build the action buttons for your app's management bar. A code snippet from the Site admin portlet is shown below:

```
<liferay-frontend:management-bar-action-buttons>
    <liferay-frontend:management-bar-sidenav-toggler-button
        icon="info-circle"
        label="info"
    />

    <liferay-frontend:management-bar-button
        href="javascript:deleteEntries();"
        icon="trash"
        id="deleteSites"
        label="delete"
    />
</liferay-frontend:management-bar-action-buttons>
```

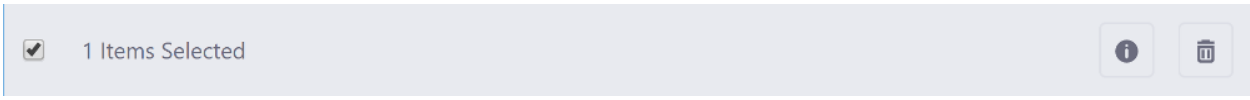


Figure 91.7: You can have as many actions as your app requires.

Related Topics

Disabling All or Portions of the Management Bar
    Clay Management Toolbar

91.2 Disabling All or Portions of the Management Bar

When there are no search results to display, you should disable all the Management Bar's buttons, except the sidenav toggler 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 components by adding the `disabled` attribute to the corresponding tag. The example below disables the display buttons when the search container displays 0 results, since changing the display style has no effect when there aren't any results to view:

```
<liferay-frontend:management-bar-display-buttons
        disabled="<%= total == 0 %>"
        displayViews='<%= new String[] {"descriptive", "icon", "list"} %>'
        portletURL="<%= changeDisplayStyleURL %>"
        selectedDisplayStyle="<%= displayStyle %>"
/>
```

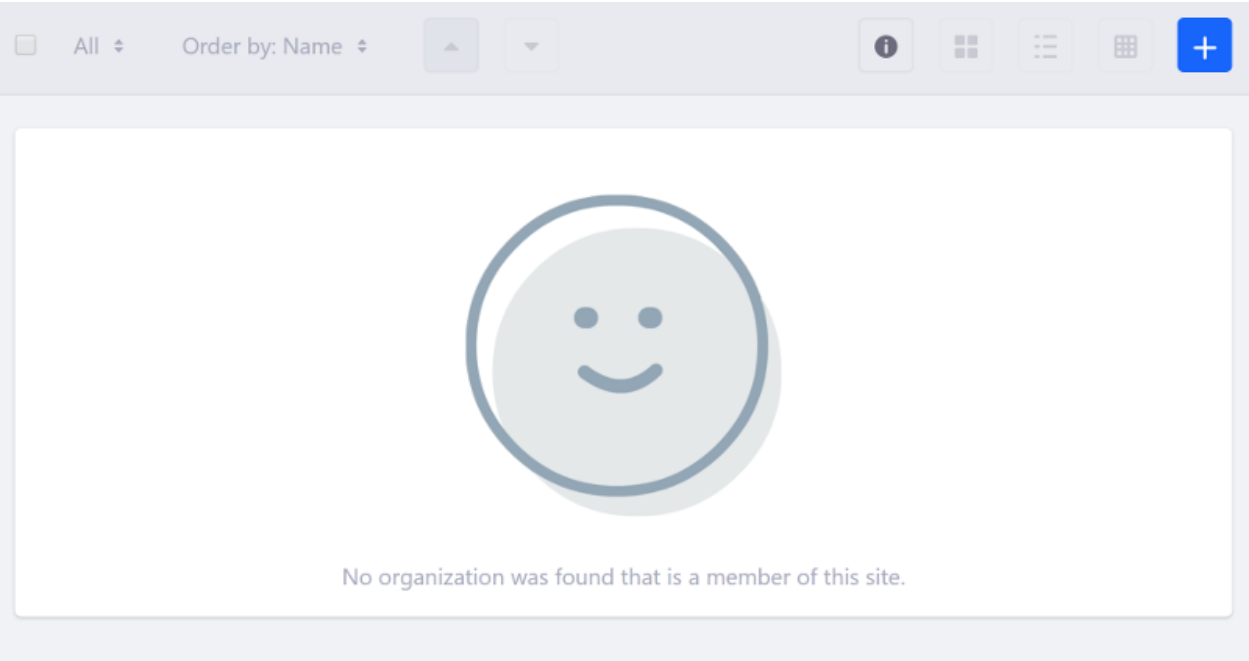


Figure 91.8: You can disable all or portions of the Management Bar.

Related Topics

Including Actions in the Management Bar
    Clay Management Toolbar

CHAPTER 92

USING THE LIFERAY UTIL TAGLIB

The Liferay Util taglib is used to pull other resources into a portlet or theme. You can use it to specify which resources to insert at the bottom or top of the page's HTML.

To use the Liferay-Util taglib, 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 templates and web content templates. Follow this syntax:

```
<@liferay_util["tag-name"] attribute="string value" attribute=10 />
```

This section of tutorials covers the available Liferay Util tags you can use in your app to inject content into portlets and themes.

92.1 Using Liferay Util Body Bottom

The body bottom tag is not a self-closing tag. It lets you add additional HTML or scripts to the bottom of the `body` tag. content placed between the opening and closing of this tag is passed to the body_bottom.jsp and outputs in this JSP.

This tag also has an optional `outputKey` attribute. If several portlets on the page include the same resource with this tag, you can specify the same `outputKey` value for each tag so the resource is only loaded once.

The example configuration below uses the `<liferay-util:body-bottom>` tag to include JavaScript provided by the portlet's bundle:

```
<liferay-util:body-bottom outputKey="bodybottom" >
    <script
  src="/o/my-liferay-util-portlet/js/my_custom_javascript_body_bottom.js"
  type="text/javascript"></script>
</liferay-util:body-bottom>
```

Now you know how to use the `<liferay-util:body-bottom>` tag to include additional resources in the bottom of the page's body.

Related Topics

Using the Liferay Util HTML Body Top Tag
    Using the Liferay Util HTML Top Tag
    Using the Liferay UI Taglib

92.2 Using Liferay Util Body Top

The 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.

This tag also has an optional `outputKey` attribute. If several portlets on the page include the same resource with this tag, you can specify the same `outputKey` value for each tag so the resource is only loaded once.

The example configuration below uses the `<liferay-util:body-top>` tag to include JavaScript provided by the portlet's bundle:

```
<liferay-util:body-top outputKey="bodytop" >
    <script
  src="/o/my-liferay-util-portlet/js/my_custom_javascript_body_top.js"
  type="text/javascript"></script>
</liferay-util:body-top>
```

Now you know how to use the `<liferay-util:body-top>` tag to include additional resources in the top of the page's body.

Related Topics

Using the Liferay Util HTML Body Bottom Tag
    Using the Liferay Util HTML Bottom Tag
    Using the Clay Taglib

92.3 Using Liferay Util Buffer

The buffer tag is not a self-closing tag. The content placed between the opening and closing of this tag is saved to a buffer and its output is assigned to the Java variable declared with the tag's `var` attribute. The output is returned as a String, letting you post-process it. For example, you can use this to override a JSP's existing contents.

The example below saves the link's generated markup to a buffer and then uses the returned string as the argument for a `liferay-ui:message` key:

```
<liferay-util:buffer
        var="linkContent"
>
        <aui:a
            href="https://portal.liferay.dev/"
            target="_blank">the Liferay Portal project
        </aui:a>
</liferay-util:buffer>

<liferay-ui:message
```

```
        arguments="<%= linkContent %>"
        key="see-x-for-more-information"
        translateArguments="<%= false %>"
/>
```

Now you know how to use the `<liferay-util:buffer>` tag to save content to a buffer.

Figure 92.1: You can use the Liferay Util Buffer tag to save pieces of markup to reuse in your JSP.

Related Topics

JSP Overrides Using OSGi Fragments
Using the Liferay Util Param Tag
Using the Liferay Front-End Taglibs

92.4 Using Liferay Util Dynamic Include

The dynamic include tag lets you specify a point or points in a JSP or theme where a developer can inject additional HTML, resources, or functionality, using the `DynamicIncludeRegistry`. You can read more about the OSGi Service Registry here. The `key` attribute identifies the extension point. See the Dynamic Include tutorials section for example configurations that use dynamic include extension points to inject additional functionality.

The example configuration below uses the `<liferay-util:dynamic-include>` tag to include an extension point before the primary code and an extension point after the primary code:

```
<%@ 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" />
```

Now you know how to use the `<liferay-util:dynamic-include>` tag to add extension points to your app.

Related Topics

Dynamic Includes
Using the Liferay Util Body Top Tag
Using the Chart Taglib

92.5 Using Liferay Util Get URL

The 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.

A basic configuration for the `<liferay-util:get-url>` tag is shown below:

```
<liferay-util:get-url url="https://portal.liferay.dev/" />
```

Here is an example that uses the var attribute:

```
<liferay-util:get-url url="https://portal.liferay.dev/" var="ldn" />

<div>
        <h2>We stole <a href="https://portal.liferay.dev/">Liferay Portal</a>, here it is.</h2>

        <div class="ldn">
                <%= ldn %>
        </div>
</div>
```

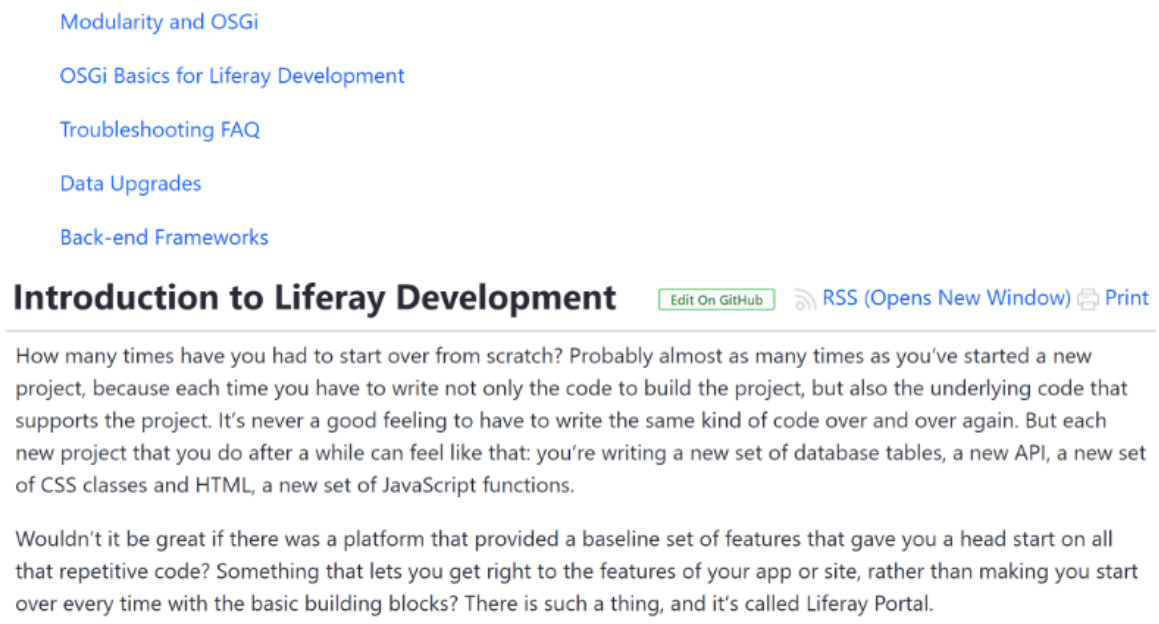


Figure 92.2: You can use the Liferay Util Get URL tag to scrape URLs.

Now you know how to use the `<liferay-util:get-url>` tag to scrape URLs.

Related Topics

Using the Liferay Util Param Tag
Using the Liferay Util Include Tag
Using the AUI Taglib

92.6 Using Liferay Util HTML Bottom

The HTML bottom tag is not a self-closing tag. Content placed between the opening and closing of this tag is 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.

This tag also has an optional `outputKey` attribute. If several portlets on the page include the same resource with this tag, you can specify the same `outputKey` value for each tag so the resource is only loaded once.

The example configuration below uses the `<liferay-util:html-bottom>` tag to include JavaScript (a common use case) provided by the portlet's bundle:

```
<liferay-util:html-bottom outputKey="htmlbottom">

    <script src="/o/my-liferay-util-portlet/js/my_custom_javascript.js"
    type="text/javascript"></script>

</liferay-util:html-bottom>
```

Now you know how to use the `<liferay-util:html-bottom>` tag to include additional resources in the bottom of the page's HTML tag.

Related Topics

Using the Liferay Util HTML Body Bottom Tag
Using the Liferay Util HTML Top Tag
Using the Liferay UI Taglib

92.7 Using Liferay Util HTML Top

The 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.

This tag also has an optional `outputKey` attribute. If several portlets on the page include the same resource with this tag, you can specify the same `outputKey` value for each tag so the resource is only loaded once.

The example configuration below uses the `<liferay-util:html-top>` tag to include additional CSS styles provided by the portlet's bundle:

```
<liferay-util:html-top outputKey="htmltop">
                <link data-senna-track="permanent"
        href="/o/my-liferay-util-portlet/css/my-custom-styles.css"
        rel="stylesheet" type="text/css" />
</liferay-util:html-top>
```

Now you know how to use the `<liferay-util:html-top>` tag to include additional resources in the top of the page's HTML tag.

Related Topics

Using the Liferay Util HTML Bottom Tag
    Using the Liferay Util Body Top Tag
    Using the Clay Taglib

92.8 Using Liferay Util Include

The include tag lets you include other JSP files in your portlet's JSP, theme, or web content. This can increase readability as well as provide separation of concerns for JSP files.

The `page` attribute is required and specifies the path to the JSP or JSPF to include. The `servletContext` refers to the request context that the included JSP should use. Passing `<%= application %>` to this attribute lets the included JSP use the same `request` object as other objects that might be set in the prior JSP.

Below is an example configuration for the `<liferay-util:include>` tag:

```
<liferay-util:include
  page="/relative/path/to/file.jsp"
  servletContext="<%= application %>"
/>
```

Now you know how to use the `<liferay-util:include>` tag to include other JSPs in your portlets, themes, and web content.

Related Topics

Using the Liferay Util Param Tag
    Using the Liferay Util Dynamic Include Tag
    Using the Liferay Front-End Taglibs

92.9 Using Liferay Util Param

The param tag lets you set a parameter for an included JSP page. This configuration requires two JSPs. JSP A, the main view of the app, includes JSP B and sets its parameter value. This lets you dynamically set content when you include the JSP.

For example, say you have your main functionality in `my-app.jsp`, and you have additional functionality provided by `more-content.jsp`. You could have the example configuration shown below:

`more-content.jsp`:

```
<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>

<%
String answer = ParamUtil.getString(request, "answer");
%>

<div>
  <p>The answer to life, the universe and everything is <%= answer %>.</p>
</div>
```

Then in `my-app.jsp`, you can include `more-content.jsp` and set the value of the `answer` parameter:

```
<liferay-util:include page="/path/to/more-content.jsp" servletContext="<%= application %>">
  <liferay-util:param name="answer" value="42" />
</liferay-util:include>
```

This results in the following output in `my-app.jsp`:

```
The answer to life, the universe and everything is 42.
```

Now you know how to use the `<liferay-util:param>` tag to set parameters for included JSPs. You can use this approach to include common reusable pieces of code in your apps.

Related Topics

Using the Liferay Util Include Tag
Using the Liferay Util Body Top Tag
Using the Chart Taglib

92.10 Using Liferay Util Whitespace Remover

The whitespace remover tag removes line breaks and tabs from code blocks included between the opening and closing of the tag. Below is an example configuration for the `<liferay-util:whitespace-remover>` tag:
with remover:

```
<liferay-util:whitespace-remover>
    <p>Here is some text with       tabs.</p>
</liferay-util:whitespace-remover>
```

result:

```
Here is some text withtabs.
```

Now you know how to use the `<liferay-util:whitespace-remover>` tag to ensure that your code formatting is consistent.

Related Topics

Using the Liferay Util Param Tag
Using the Liferay Util Buffer Tag
Using the AUI Taglib

CHAPTER 93

USING THE CLAY TAGLIB IN YOUR PORTLETS

The Liferay Clay tag library provides a set of tags for creating Clay UI components in your app.

Note: AUI taglibs are deprecated as of 7.0. We recommend that you use Clay taglibs to avoid future compatibility issues.

To use the Clay taglib in your apps, add the following declaration to your JSP:

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

The Liferay Clay taglib is also available via a macro for your FreeMarker theme templates and web content templates. Follow this syntax:

```
<@clay["tag-name"] attribute="string value" attribute=10 />
```

Clay taglibs provide the following UI components for your apps:

- Alerts
- Badges
- Buttons
- Cards
- Dropdown Menus and Action Menus
- Form Elements
- Icons
- Labels and links
- Management Toolbar
- Navigation Bars
- Progress Bars
- Stickers

The tutorials in this section cover how to create these components with the Clay taglibs. Each tutorial contains a set of clay component examples along with a screenshot of the resulting UI.

93.1 Clay Alerts

Clay alerts come in two types: embedded and stripe. This tutorial covers both types and provides several examples of each.

Embedded Alerts

Embedded alerts are usually used inside forms. The element that contains it determines an embedded alert's width. The close action is not required for embedded alerts. The following embedded alerts can be created with Clay taglibs:

Danger alert (embedded):

```
<clay:alert
    message="This is an error message."
    style="danger"
    title="Error"
/>
```

Figure 93.1: The danger alert notifies the user of an error or issue.

Success alert (embedded):

```
<clay:alert
    message="This is a success message."
    style="success"
    title="Success"
/>
```

Figure 93.2: The success alert notifies the user when an action is successful.

Info alert (embedded):

```
<clay:alert
    message="This is an info message."
    title="Info"
/>
```

Figure 93.3: The info alert displays general information to the user.

Warning alert (embedded):

```
<clay:alert
    message="This is a warning message."
    style="warning"
    title="Warning"
/>
```

Figure 93.4: The warning alert displays a warning message to the user.

Stripe Alerts

Stripe alerts are placed below the last navigation element (either the header or the navigation bar), and they usually appear on *Save* action, communicating the status of the action once received from the server. Unlike embedded alerts, stripe alerts require the close action. A stripe alert is always the full width of the container and pushes all the content below it. The following stripe alerts can be created with Clay taglibs:

Danger alert (stripe):

```
<clay:stripe
    message="This is an error message."
    style="danger"
    title="Error"
/>
```

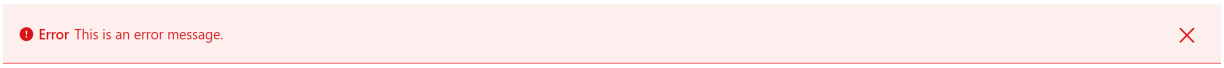


Figure 93.5: The danger striped alert notifies the user that an action has failed.

Success alert (stripe):

```
<clay:stripe
    message="This is a success message."
    style="success"
    title="Success"
/>
```

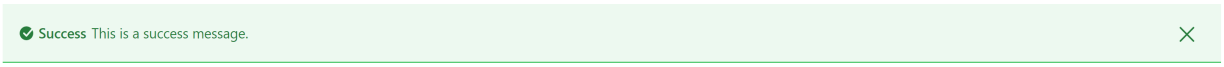


Figure 93.6: The success striped alert notifies the user that an action has completed successfully.

Info alert (stripe):

```
<clay:stripe
    message="This is an info message."
    title="Info"
/>
```

Figure 93.7: The info striped alert displays general information about an action to the user.

Warning alert (stripe):

```
<clay:stripe
    message="This is a warning message."
    style="warning"
    title="Warning"
/>
```

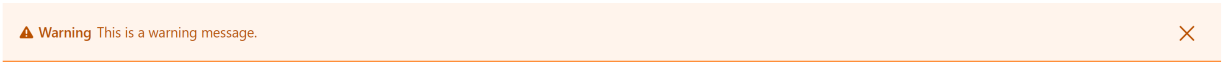


Figure 93.8: The warning striped alert warns the user about an action.

Now you know how to alert users!

Related Topics

Clay Buttons
Clay Form Elements
Clay Labels and Links

93.2 Clay Badges

Badges help highlight important information such as notifications or new and unread messages. Badges have circular borders and are only used to specify a number. This tutorial covers the different types of Clay badges you can add to your app.

Badge Types

The following badge styles are available:
Primary badge:

```
<div class="col-md-1">
    <clay:badge label="8" />

    <div>Primary</div>
</div>
```

Secondary badge:

```
<div class="col-md-1">
    <clay:badge label="87" style="secondary" />

    <div>Secondary</div>
</div>
```

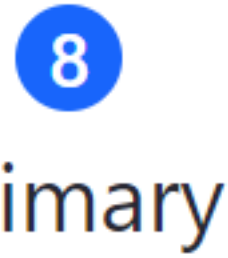


Figure 93.9: A primary badge is bright blue, commanding attention like the primary button of a form.

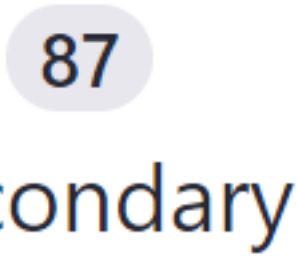


Figure 93.10: A secondary badge is light-grey and draws less focus than a primary button.

Info badge:

```
<div class="col-md-1">
    <clay:badge label="91" style="info" />

    <div>Info</div>
</div>
```

Figure 93.11: A info badge is dark blue and meant for numbers related to general information.

Error badge:

```
<div class="col-md-1">
    <clay:badge label="130" style="danger" />
```

```
    <div>Error</div>
</div>
```

Figure 93.12: An error badge displays numbers related to an error.

Success badge:

```
<div class="col-md-1">
    <clay:badge label="1111" style="success" />

    <div>Success</div>
</div>
```

Figure 93.13: A success badge displays numbers related to a successful action.

Warning badge:

```
<div class="col-md-1">
    <clay:badge label="21" style="warning" />

    <div>Warning</div>
</div>
```

Now you know how to use badges to keep track of values in your app.

Related Topics

Clay Labels and Links
Clay Cards
Clay Stickers

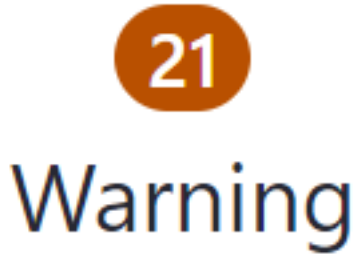


Figure 93.14: A warning badge displays numbers related to warnings that should be addressed.

93.3 Clay Buttons

Buttons come in several types and variations. This tutorial covers the different styles and variations of buttons you can create with the Clay taglibs.

Types

Primary button: Used for the most important actions. Two primary buttons should not be together or near each other.

Primary button with label:

```
<clay:button label="Primary" />
```

Figure 93.15: A primary button is bright blue, grabbing the user's attention.

Secondary button: Used for secondary actions. There can be multiple secondary buttons together or near each other.

```
<div class="col">
    <clay:button label="Secondary" style="secondary" />
</div>
<div class="col">
    <clay:button ariaLabel="Wiki" icon="wiki" style="secondary" />
</div>
```

Borderless button: Used in cases such as toolbars where the secondary button would be too heavy for the design. This keeps the design clean.

```
<div class="col">
    <clay:button label="Borderless" style="borderless" />
</div>
<div class="col">
    <clay:button ariaLabel="Page Template" icon="page-template" style="borderless" />
</div>
```

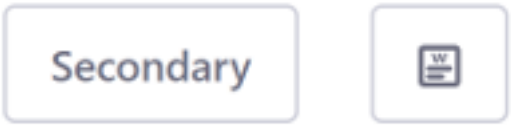


Figure 93.16: A secondary button draws less attention than a primary button and is meant for secondary actions.

Figure 93.17: Borderless buttons remove the dark outline from the button.

Link button: Used for Cancel actions.

```
<div class="col">
    <clay:button label="Link" style="link" />
</div>
<div class="col">
    <clay:button ariaLabel="Add Role" icon="add-role" style="link" />
</div>
```

Figure 93.18: You can also turn buttons into links.

You can use labels or icons for your buttons. Below is an example of a Primary button with an icon:

```
<clay:button ariaLabel="Workflow" icon="workflow" />
```

Figure 93.19: Buttons can also display icons.

You can disable a button by adding the `disabled` attribute:

```
<div class="col">
    <clay:button disabled="<%= true %>" label="Primary" />
</div>
<div class="col">
    <clay:button ariaLabel="Workflow" disabled="<%= true %>" icon="workflow" />
</div>
```

Figure 93.20: Buttons can be disabled if you don't want the user to interact with them.

Variations

Button with icon and text:

```
<clay:button icon="share" label="Share" />
```

Figure 93.21: Buttons can display both icons and text.

Button with monospaced text:

```
<clay:button icon="indent-less" monospaced="<%= true %>" style="secondary" />
```

Figure 93.22: Buttons can display monospaced text.

Block level button:

```
<clay:button block="<%= true %>" label="Button" />
```

Figure 93.23: Block level buttons span the entire width of the container.

Plus button:

```
<clay:button icon="plus" monospaced="<%= true %>" style="secondary" />
```

Action button:

```
<clay:button icon="ellipsis-v" monospaced="<%= true %>" style="borderless" />
```

Figure 93.24: : A plus button is used for add actions in an app.

Figure 93.25: : An action button is used to display actions menus.

Related Topics

Clay Alerts
    Clay Buttons
    Clay Labels and Links

93.4 Clay Cards

Cards visually represent data. Use them for images, document libraries, user profiles and more. There are four main types of Cards:

- Image Cards
- File Cards
- User Cards
- Horizontal Cards

Each of these types is covered in this tutorial.

Image Cards

Image Cards are used for image/document galleries.
    Image Card:

```
<clay:image-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    href="#1"
    imageAlt="thumbnail"
    imageSrc="https://images.unsplash.com/photo-1506976773555-b3da30a63b57"
    subtitle="Author Action"
    title="Madrid"
/>
```

Image Card with icon:

```
<clay:image-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    icon="camera"
    subtitle="Author Action"
    title="<%= SVG_FILE_TITLE %>"
/>
```

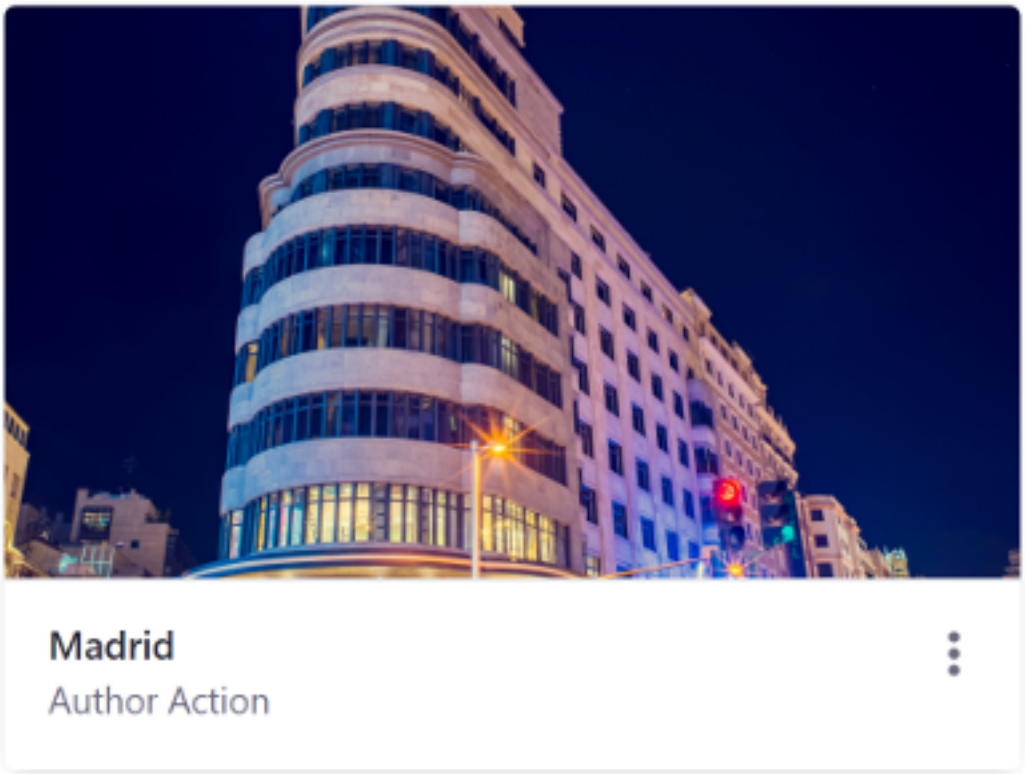


Figure 93.26: Image Cards display images and documents.

Image Card empty:

```
<clay:image-card
  actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    subtitle="Author Action"
    title="<%= SVG_FILE_TITLE %>"
/>
```

Cards can also contain file types. Specify the file type with the `filetype` attribute:

```
<clay:image-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    fileType="JPG"
    fileTypeStyle="danger"
    href="#1"
    imageAlt="thumbnail"
    imageSrc="https://images.unsplash.com/photo-1499310226026-b9d598980b90"
    subtitle="Author Action"
    title="California"
/>
```

Include the `labels` attribute to add a label to a Card:

```
<clay:image-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    fileType="JPG"
    fileTypeStyle="danger"
    href="#1"
    imageAlt="thumbnail"
    imageSrc="https://images.unsplash.com/photo-1503703294279-c83bdf7b4bf4"
```

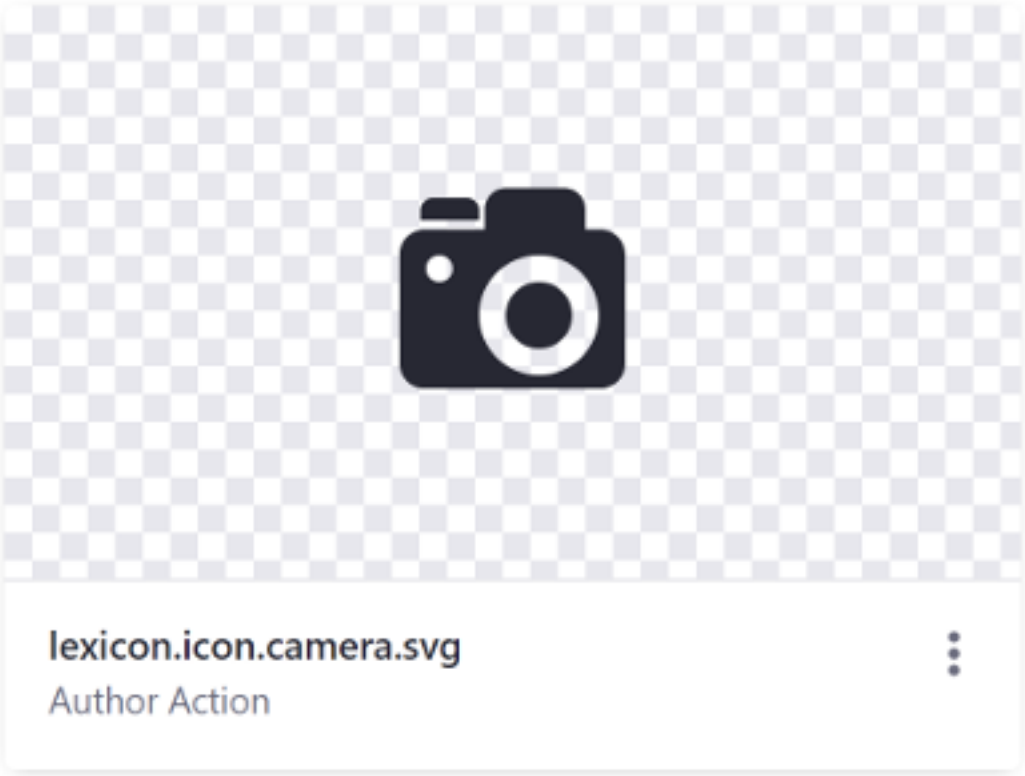


Figure 93.27: Image Cards can also display icons instead of images.

```
    labels="<%= cardsDisplayContext.getLabels() %>"
    subtitle="Author Action"
    title="Beetle"
/>
```

Include the `selectable` attribute to make cards selectable (include a checkbox):

```
<clay:image-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    fileType="JPG"
    fileTypeStyle="danger"
    href="#1"
    imageAlt="thumbnail"
    imageSrc="https://images.unsplash.com/photo-1506020647804-b04ee956dc04"
    labels="<%= cardsDisplayContext.getLabels() %>"
    selectable="<%= true %>"
    selected="<%= true %>"
    subtitle="Author Action"
    title="Beetle"
/>
```

File Cards

File Cards display an icon of the file's type. They represent file types other than image files (i.e. PDF, MP3, DOC, etc.).

```
<clay:file-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
```

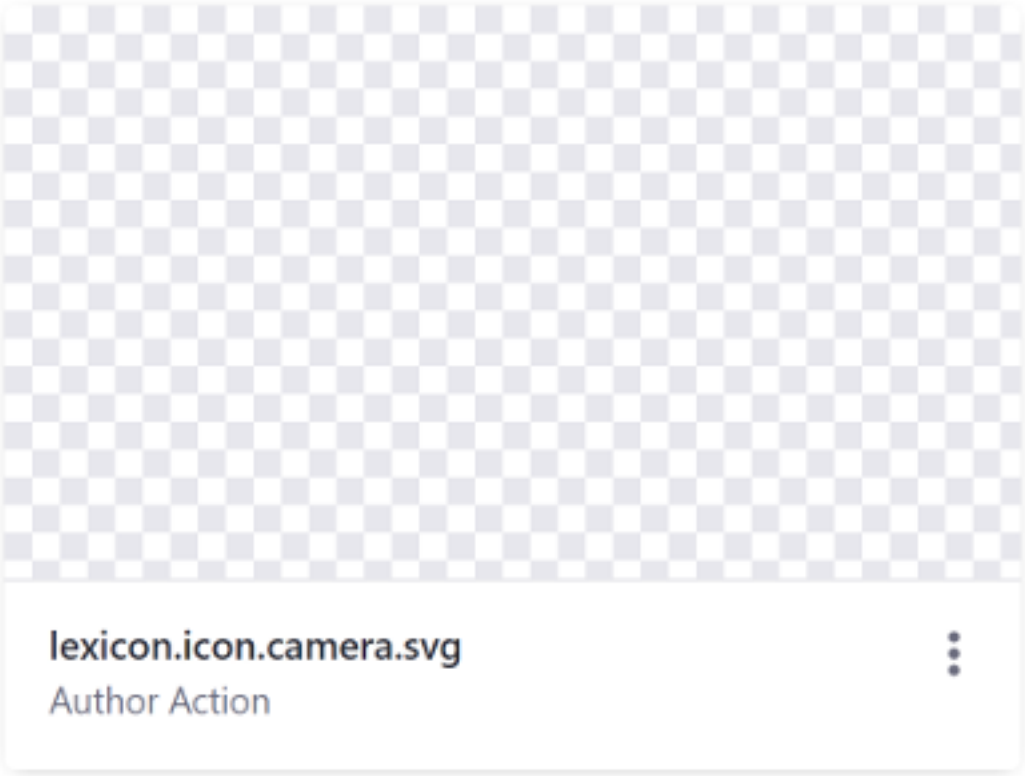


Figure 93.28: Cards can also display nothing.

```
    fileType="MP3"
    fileTypeStyle="warning"
    labels="<%= cardsDisplayContext.getLabels() %>"
    labelStylesMap="<%= cardsDisplayContext.getLabelStylesMap() %>"
    selectable="<%= true %>"
    selected="<%= true %>"
    subtitle="Jimi Hendrix"
    title="<%= MP3_FILE_TITLE %>"
/>
```

You can optionally use the `labelStylesMap` attribute to pass a `HashMap` of multiple labels, as shown above.

The example below specifies a list `icon` instead of the default file icon:

```
<clay:file-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    fileType="DOC"
    fileTypeStyle="info"
    icon="list"
    labels="<%= cardsDisplayContext.getLabels() %>"
    selectable="<%= true %>"
    selected="<%= true %>"
    subtitle="Paco de Lucia"
    title="<%= DOC_FILE_TITLE %>"
/>
```

Note: The full list of available Liferay icons can be found on the Clay CSS website.

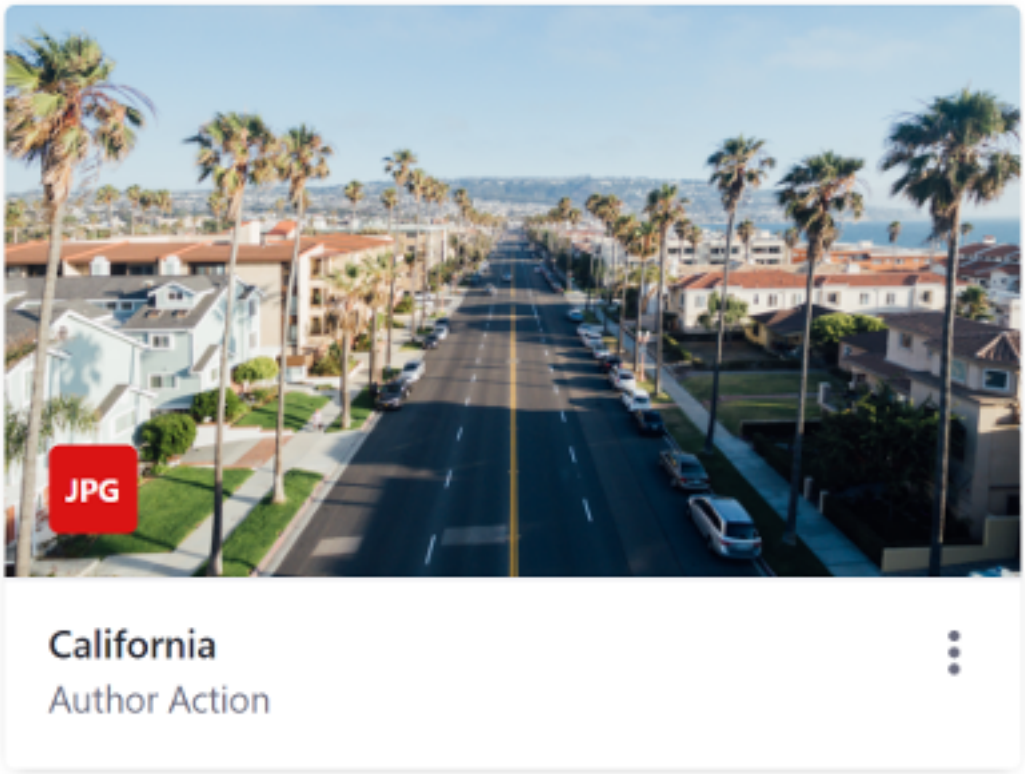


Figure 93.29: Cards can also contain file types.

User Cards

User Cards display user profile images or the initials of the user's name or name+surname.

User Card with initials:

```
<clay:user-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    initials="HS"
    name="User Name"
    subtitle="Latest Action"
    userColor="danger"
/>
```

User Card with profile image:

```
<clay:user-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    disabled="<%= true %>"
    imageAlt="thumbnail"
    imageSrc="https://images.unsplash.com/photo-1502290822284-9538ef1f1291"
    name="User name"
    selectable="<%= true %>"
    selected="<%= true %>"
    subtitle="Latest Action"
/>
```

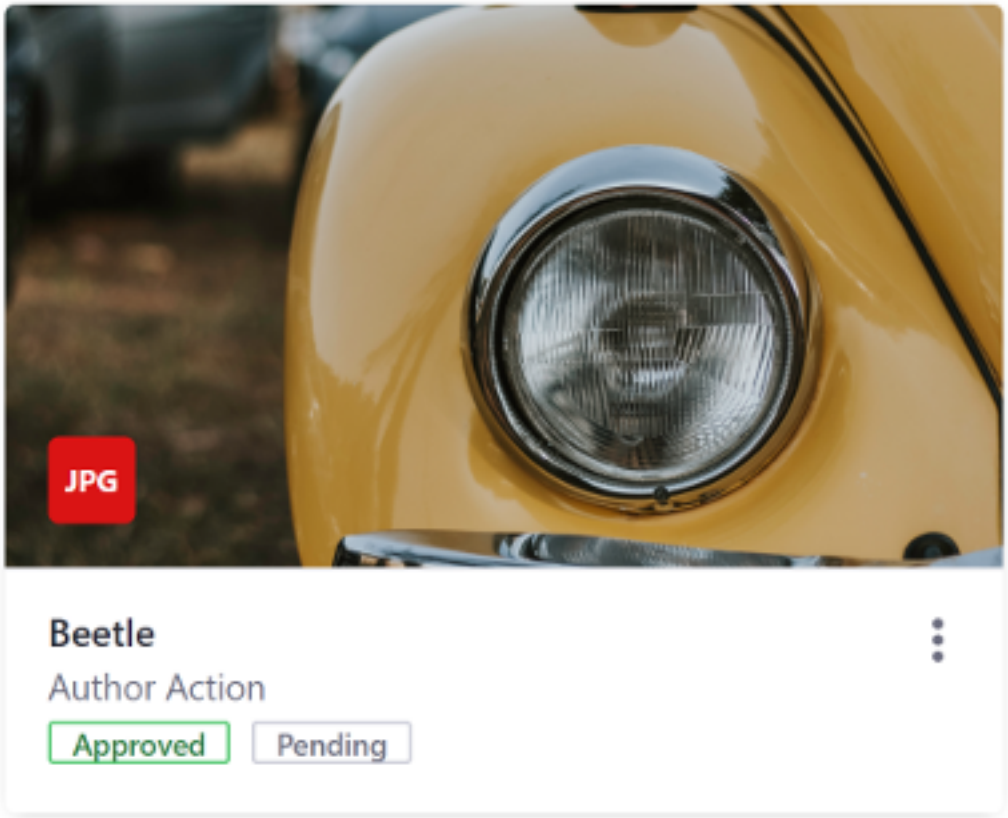


Figure 93.30: You can include labels in Cards.

Horizontal Cards

Horizontal Cards represent folders and can have the same amount of information as other Cards. The key difference is that horizontal Cards let you remove the image portion of the Card, since only the folder icon is required.

```
<clay:horizontal-card
    actionItems="<%= cardsDisplayContext.getDefaultActionItems() %>"
    selectable="<%= true %>"
    selected="<%= true %>"
    title="ReallySuperInsanelyJustIncrediblyLongAndTotallyNotPossibleWordButWeAreReallyTryingToCoverAllOurBasesHereJustInCaseSomeoneIsNutsAsPerUsual"
/>
```

Now you know how to use Cards in your UI to display information in your apps.

Related Topics

Clay Badges
    Clay Labels and Links
    Clay Stickers

93.5 Clay Dropdown Menus and Action Menus

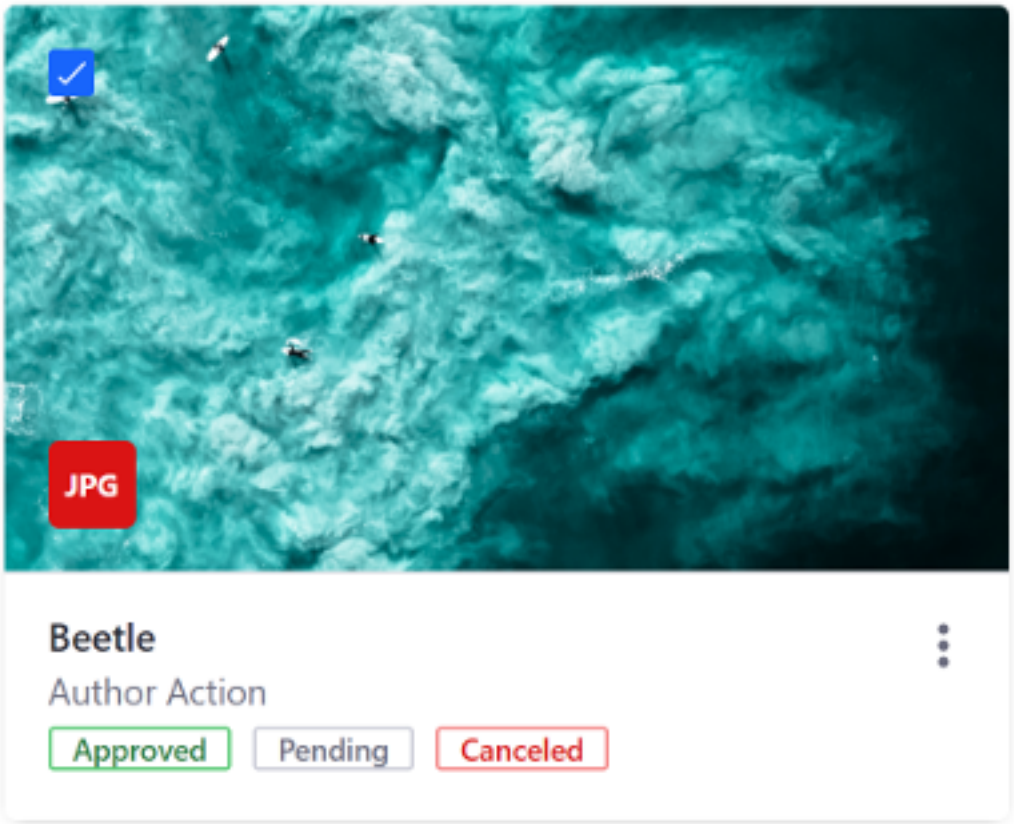


Figure 93.31: Cards can be selectable.

You can add dropdown menus to your app via the `clay:dropdown-menu` and `clay:actions-menu` taglibs. The Clay taglibs provide several menu variations for you to choose. Both taglibs with several examples are shown below.

Dropdown Menus

Basic dropdown menu:

```
<clay:dropdown-menu
    items="<%= dropdownsDisplayContext.getDefaultDropdownItems() %>"
    label="Default"
/>
```

The dropdown menu's items are defined in its Java class–`dropdownDisplayContext` in this case. Menu items are `NavigationItem` objects. You can disable menu items with the `setDisabled(true)` method and make a menu item active with the `setActive(true)` method. The `href` attribute is set with the `setHref()` method, and labels are defined with the `setLabel()` method. Here's an example implementation of the `dropdownDisplayContext` class:

```
    if (_defaultDropdownItems != null) {
      return _defaultDropdownItems;
    }
```

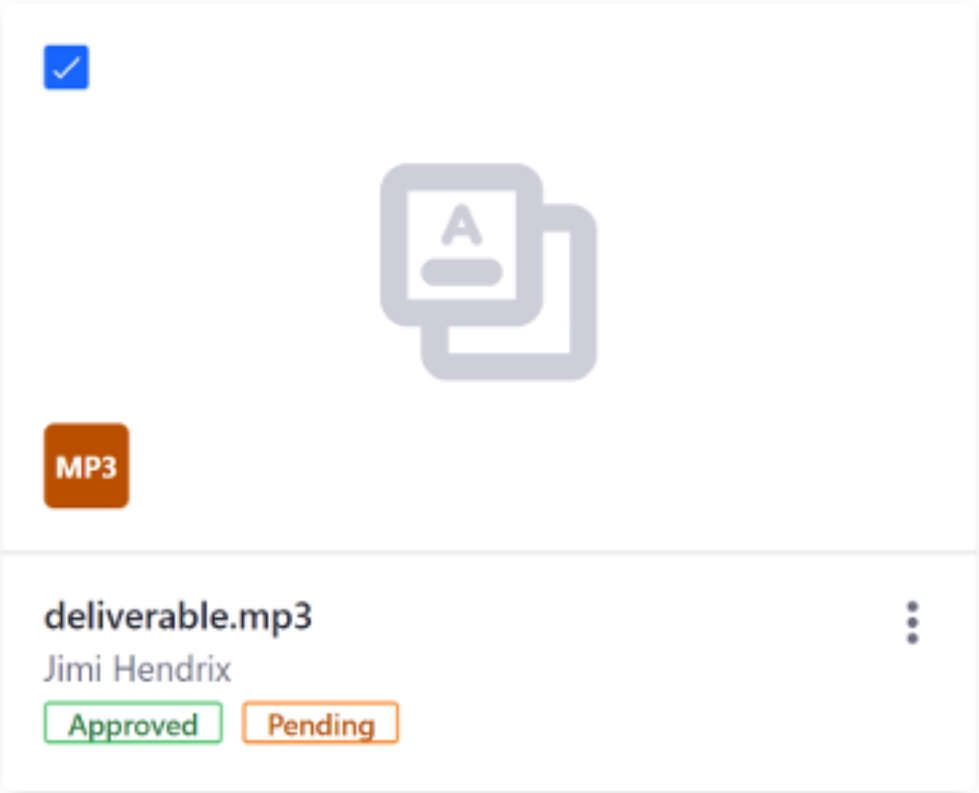


Figure 93.32: File Cards display file type icons.

```
_defaultDropdownItems = new ArrayList<>();

for (int i = 0; i < 4; i++) {
  NavigationItem navigationItem = new NavigationItem();

  if (i == 1) {
    navigationItem.setDisabled(true);
  }
  else if (i == 2) {
    navigationItem.setActive(true);
  }

  navigationItem.setHref("#" + i);
  navigationItem.setLabel("Option " + i);

  _defaultDropdownItems.add(navigationItem);
}

return _defaultDropdownItems;
}
```

You can organize menu items into groups by setting the `NavigationItem`'s type to `TYPE_GROUP` and nesting the items in separate `ArrayLists`. You can add a horizontal separator to separate the groups visually with the `setSeparator(true)` method. Below is a code snippet from the `dropdownsDisplayContext` class:

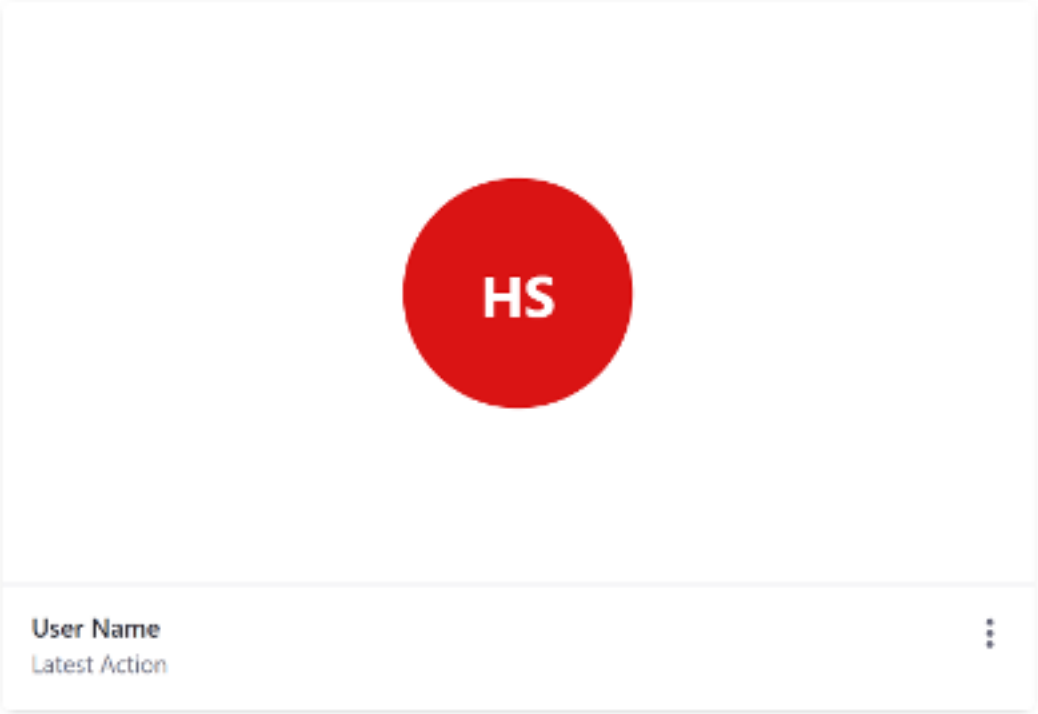


Figure 93.33: User Cards can display a user's initials.

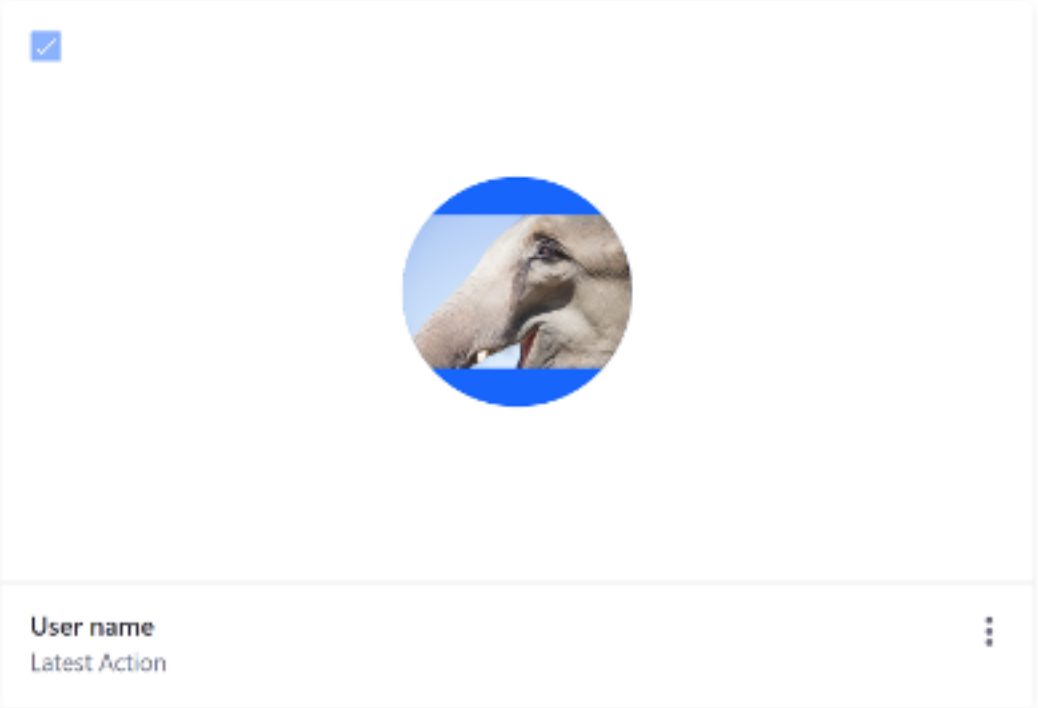


Figure 93.34: A User Card can also display a profile image.

Figure 93.35: : Horizontal Cards are good for displaying folders.

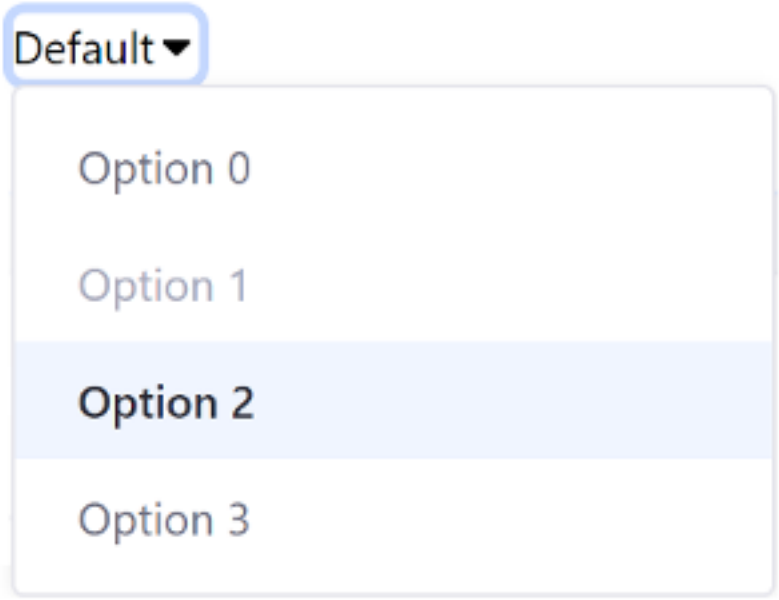


Figure 93.36: Clay taglibs provide everything you need to add dropdown menus to your app.

```
group1NavigationItem.setSeparator(true);
group1NavigationItem.setType(NavigationItem.TYPE_GROUP);
```

Corresponding taglib:

```
<clay:dropdown-menu
    items="<%= dropdownsDisplayContext.getGroupDropdownItems() %>"
    label="Dividers"
/>
```

You can also add inputs to dropdown menus. To add an input to a dropdown menu, set the input's type with the `setType()` method (e.g. `NavigationItem.TYPE_CHECKBOX`), its name with the `setInputName()` method, and its value with the `setInputValue()` method. Here's an example implementation:

```
navigationItem.setInputName("checkbox" + i);
navigationItem.setInputValue("checkboxValue" + i);
navigationItem.setLabel("Group 1 - Option " + i);
navigationItem.setType(NavigationItem.TYPE_CHECKBOX);
```

Corresponding taglib:

```
<clay:dropdown-menu
    buttonLabel="Done"
    items="<%= dropdownsDisplayContext.getInputDropdownItems() %>"
```

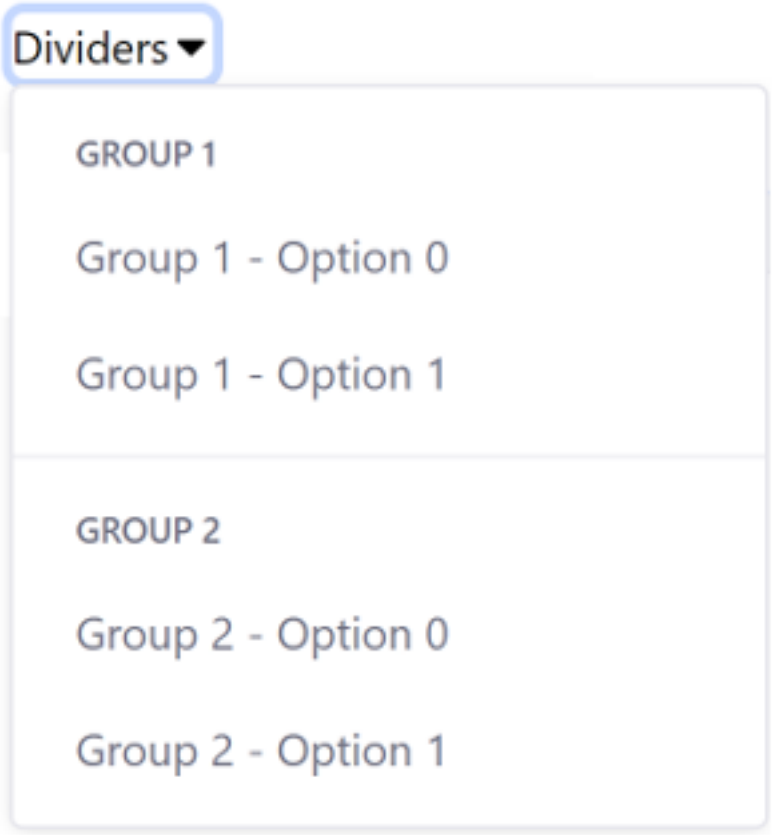


Figure 93.37: You can organize dropdown menu items into groups.

```
    label="Inputs"
    searchable="<%= true %>"
/>
```

Menu items can also contain icons. To add an icon to a menu item, use the `setIcon()` method. Below is an example:

```
navigationItem.setIcon("check-circle-full")
```

Corresponding taglib:

```
<clay:dropdown-menu
    items="<%= dropdownsDisplayContext.getIconDropdownItems() %>"
    itemsIconAlignment="left"
    label="Icons"
/>
```

Actions Menus

Basic actions menu:

```
<clay:dropdown-actions
    items="<%= dropdownsDisplayContext.getDefaultDropdownItems() %>"
/>
```

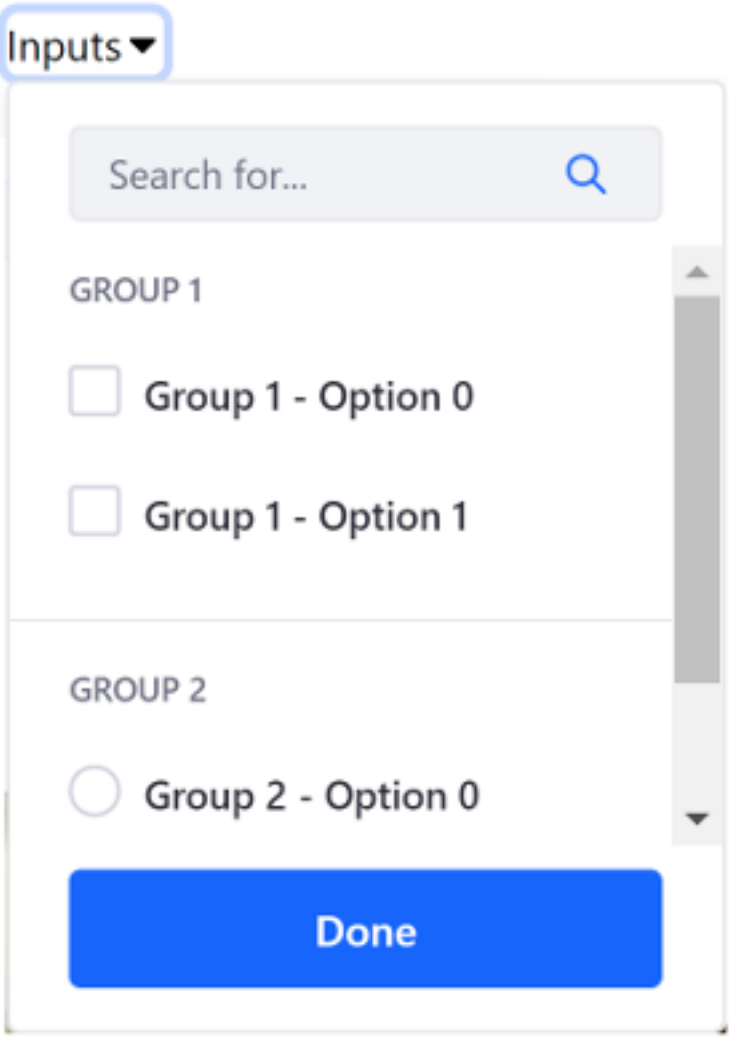


Figure 93.38: Inputs can be included in dropdown menus.

An actions menu can also display help text to the user:

```
<clay:dropdown-actions
    buttonLabel="More"
    buttonStyle="secondary"
    caption="Showing 4 of 32 Options"
    helpText="You can customize this menu or see all you have by pressing \"more\"."
    items="<%= dropdownsDisplayContext.getDefaultDropdownItems() %>"
/>
```

Clay taglibs make it easy to add dropdown menus and action menus to your apps.

Related Topics

Clay Form Elements
Clay Navigation Bars

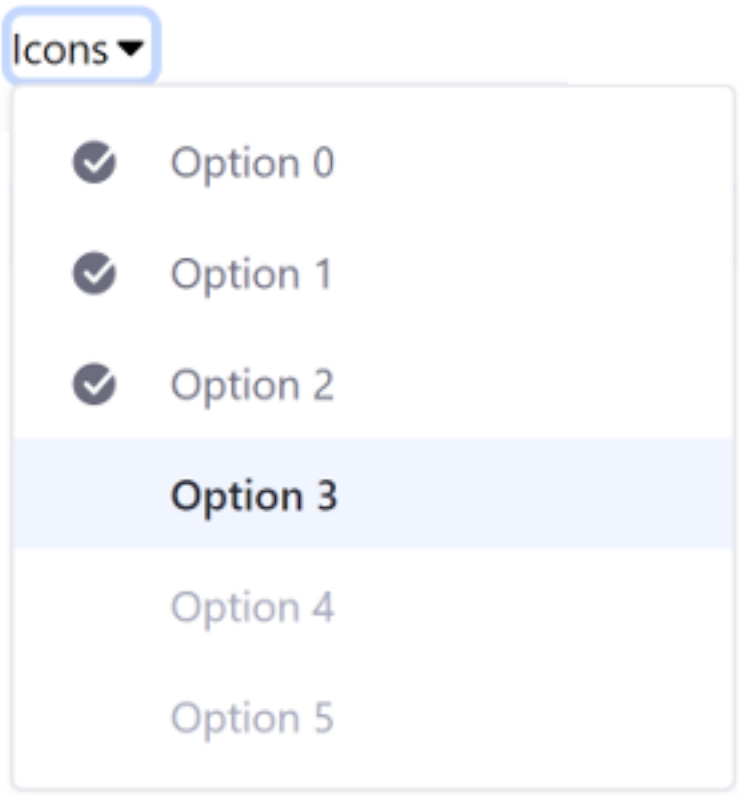


Figure 93.39: Icons can be included in dropdown menus.

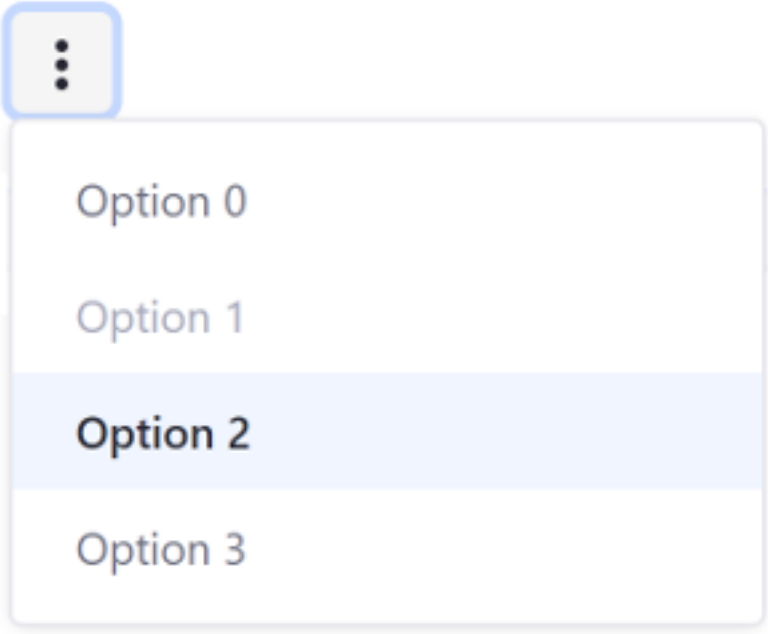


Figure 93.40: You can also create Actions menus with Clay taglibs.

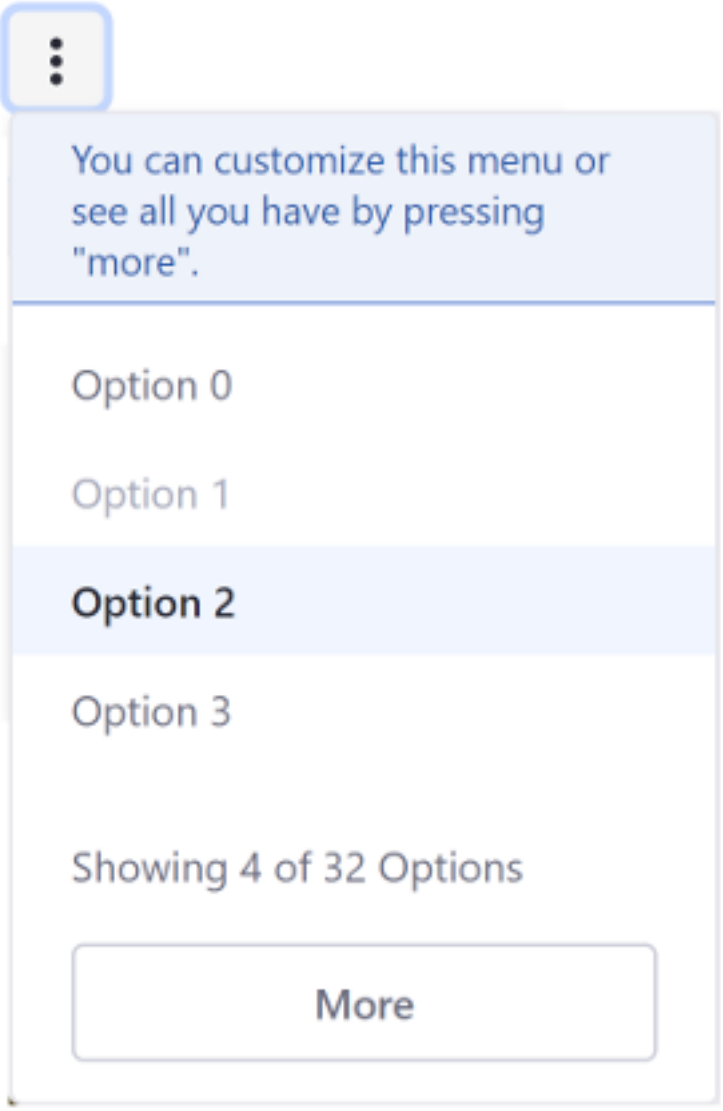


Figure 93.41: You can provide help text in Actions menus.

Clay Progress Bars

93.6 Clay Form Elements

The Liferay Clay tag library provides several tags for creating form elements. An example of each tag is shown below.

Checkbox

Checkboxes give the user a true or false input.

```
<clay:checkbox
        checked="<%= true %>"
        hideLabel="<%= true %>"
        label="My Input"
        name="name"
/>
```

Attributes:
checked: Whether the checkbox is checked
disabled: Whether the checkbox is enabled
hideLabel: Whether to display the checkbox label
indeterminate: Checkbox variable for multiple selection
label: The checkbox's label
name: The checkbox's name

Figure 93.42: Clay taglibs provide checkboxes.

Radio

A radio button lets the user select one choice from a set of options in a form.

```
<clay:radio
        checked="<%= true %>"
        hideLabel="<%= true %>"
        label="My Input"
        name="name"
/>
```

Attributes:
checked: Whether the radio button is checked
hideLabel: Whether to display the radio button label
disabled: Whether the radio button is enabled
label: The radio button's label
name: The radio button's name

Figure 93.43: Clay taglibs provide radio buttons.

Selector

A selector gives the user a select box with a set of options to choose from.

The Java scriplet below creates eight dummy options for the selector:

```
<%
List<Map<String, Object>> options = new ArrayList<>();

for (int i = 0; i < 8; i++) {
    Map<String, Object> option = new HashMap<>();

    option.put("label", "Sample " + i);
    option.put("value", i);

    options.add(option);
}
%>

<clay:select
            label="Regular Select Element"
            name="name"
            options="<%= options %>"
    />
```

If you want let users select multiple options at once, set the `multiple` attribute to `true`:

```
<clay:select
            label="Multiple Select Element"
            multiple="<%= true %>"
            name="name"
            options="<%= options %>"
    />
```

Attributes:

disabled: Whether the selector is enabled **label:** The selector's label **multiple:** Whether multiple options can be selected **name:** The selector's name

Now you know how to use Clay taglibs to add common form elements to your app!

Related Topics

Clay Buttons

Clay Icons

Clay Labels and Links

93.7 Clay Icons

The Liferay Clay taglibs provide several icons that you can use in your apps. Use the `clay:icon` tag and specify the icon with the `symbol` attribute:

```
<clay:icon symbol="folder" />
```

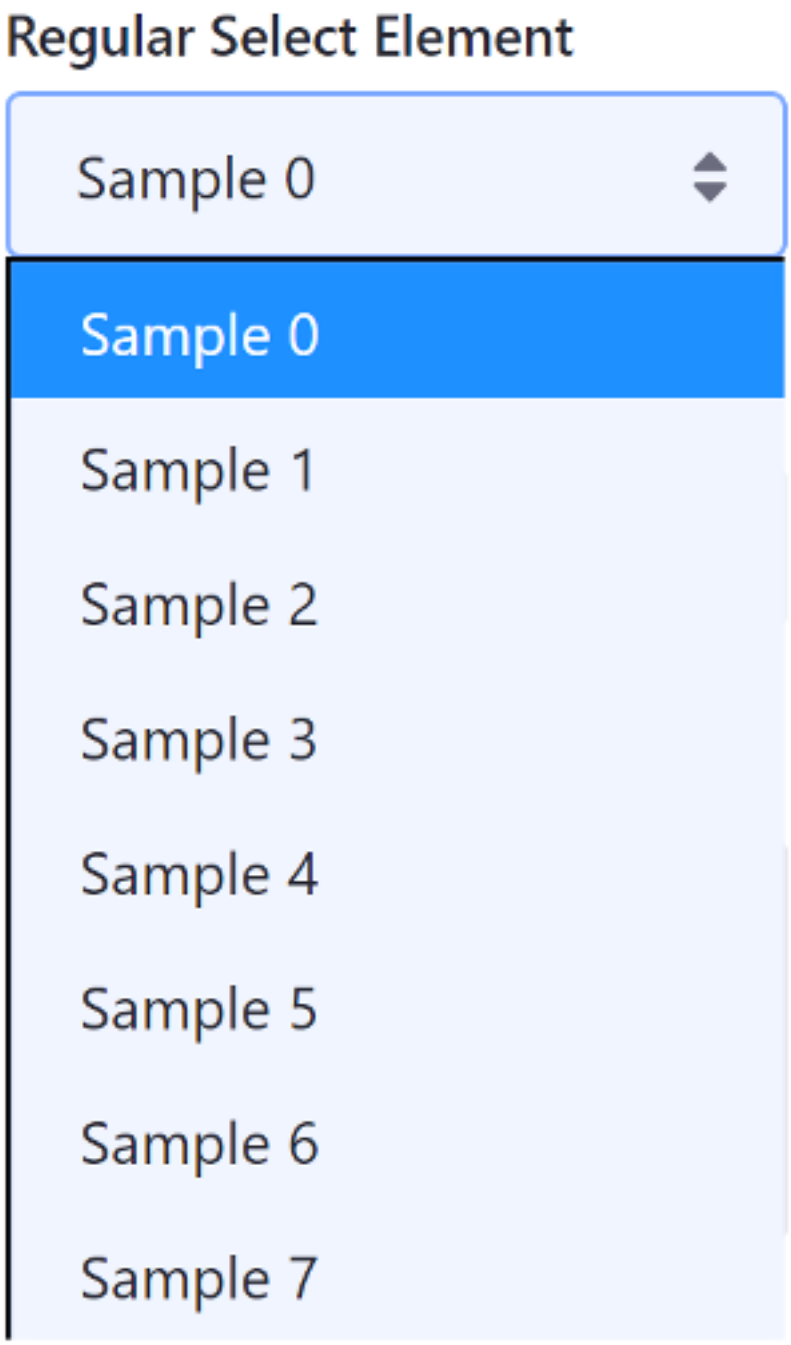


Figure 93.44: Clay taglibs provide select boxes.

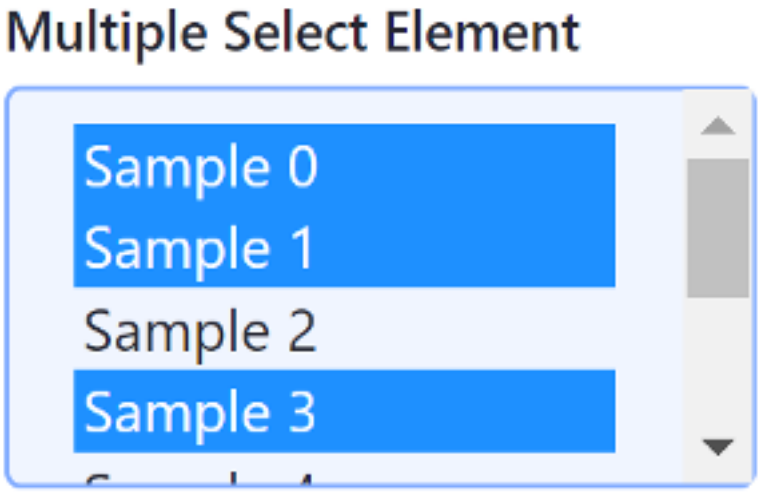


Figure 93.45: You can let users select multiple options from the select menu.

Figure 93.46: You can include icons in your app with the Clay taglib.

The full list of icons is shown below:

The Liferay Clay taglibs also provide a set of language flag icons that you can use in your app. The full list of language flags is shown below:

Related Topics

Clay Badges

Clay Stickers

Clay Icon Component

93.8 Clay Labels and Links

Liferay Clay taglibs provide tags for creating labels and links in your app. This tutorial shows how to add both of these UI elements to your apps.

Labels

The Liferay Clay taglibs provide a few different labels for your app. Use the `clay:label` tag to add a label to your app. You can create color-coded labels, removable labels, and labels that contain links. The sections below demonstrate all of these options.

Figure 93.47: The Clay taglib gives you access to several Liferay DXP icons.

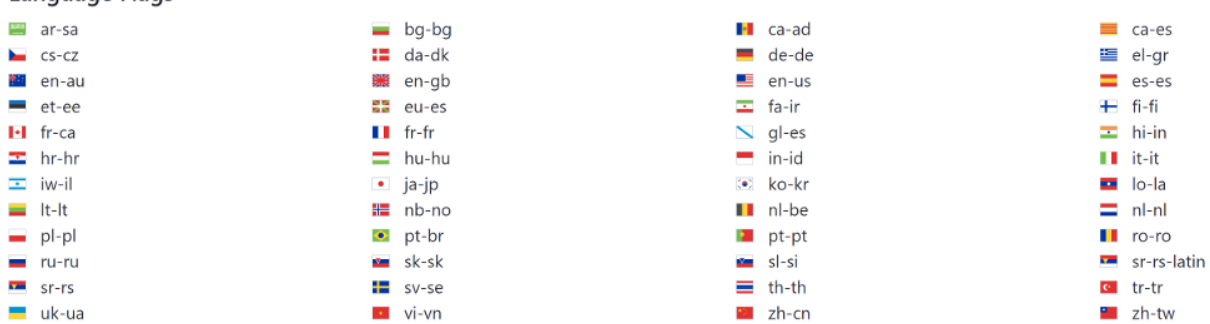


Figure 93.48: You can include language flags in your apps.

Color-coded Labels

The Liferay Clay labels come in four different colors: dark-blue for info, light-gray for status, orange for pending, red for rejected, and green for approved.

Info labels are dark-blue, and since they stand out a bit more than status labels, they are best for conveying general information. To use an info label, set the `style` attribute to `info`:

```
<clay:label label="Label text" style="info" />
```

Figure 93.49: Info labels convey general information.

Status labels are light-gray, and due to their neutral color, they are best for conveying basic information. Status labels are the default label and therefore require no `style` attribute:

```
<clay:label label="Status" />
```

Figure 93.50: Status labels are the least flashy and best for displaying basic information.

Warning labels are orange, and due to their color, they are best for conveying a warning message. To use a warning label, set the `style` attribute to `warning`:

```
<clay:label label="Pending" style="warning" />
```

Figure 93.51: Warning labels notify the user of issues, but nothing app breaking.

Danger labels are red and indicate that something is wrong or has failed. To use a danger label, set the `style` attribute to `danger`:

```
<clay:label label="Rejected" style="danger" />
```

Success labels are green and indicate that something has completed successfully. To use a success label, set the `style` attribute to `success`:

```
<clay:label label="Approved" style="success" />
```

Labels can also be bigger. Set the `size` attribute to `lg` to display large labels:

```
<clay:label label="Approved" size="lg" style="success" />
```

Figure 93.52: Danger labels convey a sense of urgency that must be addressed.

Figure 93.53: Success labels indicate a successful action.

Removable Labels

If you want to let a user close a label (e.g. a temporary notification), you can make the label removable by setting the `closeable` attribute to `true`.

```
<clay:label closeable="<%= true %>" label="Normal Label" />
```

Figure 93.54: Labels can be removable.

Labels with Links

You can make a label a link by adding the `href` attribute to it just as you would an anchor tag:

```
<clay:label href="#" label="Label Text" />
```

Figure 93.55: Labels can also be links.

Links

You can add traditional hyperlinks to your app with the `<clay:link>` tag:

```
<clay:link href="#" label="link text" />
```

Now you know how to add links and labels to your apps!

Figure 93.56: Clay taglibs also provide link elements.

Related Topics

Clay Badges
    Clay Cards
    Clay Form Elements

93.9 Clay Management Toolbar

The Management Toolbar gives administrators control over search container results in their apps. It lets you filter, sort, and choose a view type for search results, so you can quickly identify the document, web content, asset entry, or whatever you're looking for. The Management Toolbar is fully customizable, so you can implement all the controls or just the ones your app requires.

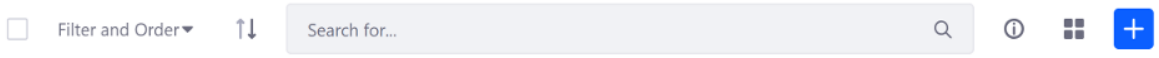


Figure 93.57: The Management ToolBar lets the user customize how the app displays content.

To create a management toolbar, use the `clay:management-toolbar` taglib. The toolbar contains a few key sections. Each section is grouped and configured using different attributes. These attributes are described in more detail below.

Using a Display Context to Configure the Management Toolbar

If you're using a Display Context—a separate class to configure your display options for your management toolbar—to define all or some of the configuration options for the toolbar, you can specify the Display Context with the `displayContext` attribute. An example is shown below:

```
<clay:management-toolbar
    displayContext="<%= viewUADEntitiesManagementToolbarDisplayContext %>"
/>
```

You can see an example use case of a Display Context in the Filtering and Sorting Items with the Management Toolbar tutorial. A Display Context is not required for a management toolbar's configuration. You can provide as much or as little of the configuration options for your management toolbar through the Display Context as you like.

Checkbox and Actions

The `actionItems`, `searchContainerId`, `selectable`, and `totalItems` attributes let you include a checkbox in the toolbar to select all search container results and run bulk actions on them. Actions and total items display when an individual result is checked, or when the master checkbox is checked in the toolbar.

`actionItems`: The list of dropdown items to display when a result is checked or the master checkbox in the Management Toolbar is checked. You can select multiple results between pages. The Management Toolbar keeps track of the number of selected results for you.

`searchContainerId`: The ID of the search container connected to the Management Toolbar

`selectable`: Whether to include a checkbox in the Management Toolbar

`totalItems`: The total number of items across pagination. This number displays when one or multiple items are selected.

An example configuration is shown below:

```
actionItems="<%=
    new JSPDropdownItemList(pageContext) {
        {
          add(
            dropdownItem -> {
              dropdownItem.setHref("#edit");
              dropdownItem.setLabel("Edit");
            });

          add(
            dropdownItem -> {
              dropdownItem.setHref("#download");
              dropdownItem.setIcon("download");
              dropdownItem.setLabel("Download");
              dropdownItem.setQuickAction(true);
            });

          add(
            dropdownItem -> {
              dropdownItem.setHref("#delete");
              dropdownItem.setLabel("Delete");
              dropdownItem.setIcon("trash");
              dropdownItem.setQuickAction(true);
            });
        }
    }
%>"
```

Action items are listed in the Actions menu, along with the number of items selected across pagination.

Figure 93.58: Actions are also listed in the Management Toolbar's dropdown menu when an item, multiple items, or the master checkbox is checked.

If an action has an icon specified, such as the Delete and Download actions in the example above, the icon is displayed next to the action menu as well.

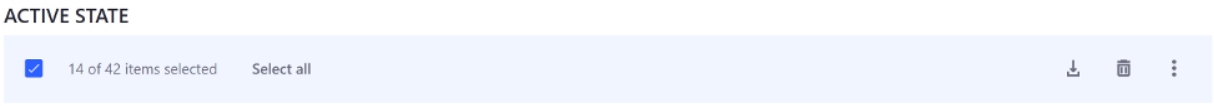


Figure 93.59: The Management Toolbar keeps track of the results selected and displays the actions to execute on them.

Filtering and Sorting Search Results

The `filterItems`, `sortingOrder`, and `sortingURL` attributes let you filter and sort search container results. Filtering and sorting are grouped together in one convenient dropdown menu.

`filterItems`: Sets the search container's filtering options. This filter should be included in all control panel applications. Filtering options can include sort criteria, sort ordering, and more.

`sortingOrder`: The current sorting order: ascending or descending.

`sortingURL`: The URL to change the sorting order

The example below adds two filter options and two sorting options:

```
filterItems="<%=
    new DropdownItemList(_request) {
        {
            addGroup(
                dropdownGroupItem -> {
                    dropdownGroupItem.setDropdownItemList(
                        new DropdownItemList(_request) {
                            {
                                add(
                                    dropdownItem -> {
                                        dropdownItem.setHref("#1");
                                        dropdownItem.setLabel("Filter 1");
                                    });

                                add(
                                    dropdownItem -> {
                                        dropdownItem.setHref("#2");
                                        dropdownItem.setLabel("Filter 2");
                                    });
                            }
                        }
                    );
                    dropdownGroupItem.setLabel("Filter By");
                });

            addGroup(
                dropdownGroupItem -> {
                    dropdownGroupItem.setDropdownItemList(
                        new DropdownItemList(_request) {
                            {
                                add(
                                    dropdownItem -> {
                                        dropdownItem.setHref("#3");
                                        dropdownItem.setLabel("Order 1");
                                    });

                                add(
                                    dropdownItem -> {
                                        dropdownItem.setHref("#4");
                                        dropdownItem.setLabel("Order 2");
                                    });
                            }
                        }
```

```
                );
                dropdownGroupItem.setLabel("Order By");
            });
        }
    }
%>"
```

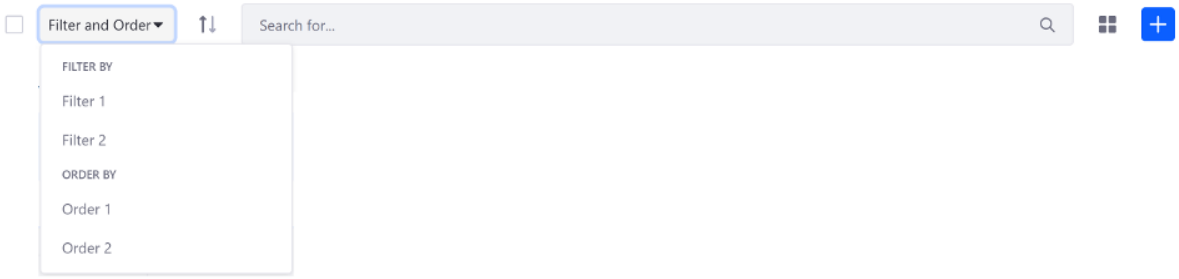


Figure 93.60: You can also sort and filter search container results.

Search Form

The `clearResultsURL`, `searchActionURL`, `searchFormName`, `searchInputName`, and `searchValue` attributes let you configure the search form. The main portion of the Management Toolbar is reserved for the search form.

`clearResultsURL`: The URL to reset the search

`searchActionURL`: The action URL to send the search form

`searchFormName`: The search form's name

`searchInputName`: The search input's name

`searchValue`: The search input's value

An example configuration is shown below:

```
<clay:management-toolbar
    clearResultsURL="<%= searchURL %>"
    disabled="<%= isDisabled %>"
    namespace="<%= renderResponse.getNamespace() %>"
    searchActionURL="<%= searchURL %>"
    searchFormName="fm"
    searchInputName="<%= DisplayTerms.KEYWORDS %>"
    searchValue="<%= ParamUtil.getString(request, searchInputName) %>"
    selectable="<%= false %>"
    totalItems="<%= totalItems %>"
/>
```

Info Panel

The `infoPanelId` and `showInfoButton` attributes let you add a retractable sidebar panel that displays additional information related to a search container result.

`infoPanelId`: The ID of the info panel to toggle

`showInfoButton`: Whether to show the info button

In the example configuration below, the `showInfoButton` attribute is provided in the Display Context—specified with the `displayContext` attribute—and the `infoPanelId` is explicitly set in the JSP:

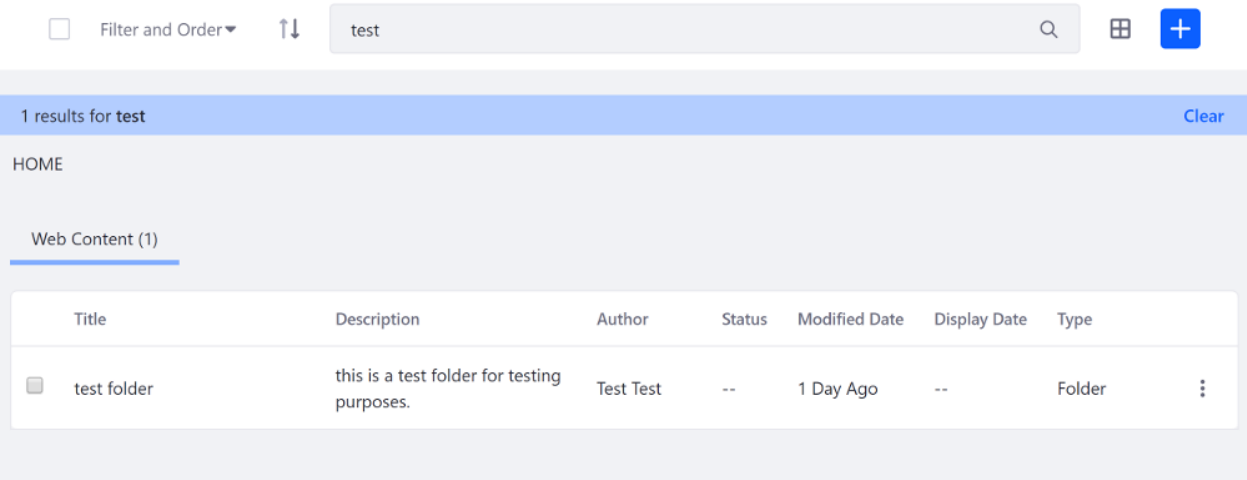


Figure 93.61: The search form comprises most of the Management Toolbar, letting users search through the search container results.

```
<clay:management-toolbar
    displayContext="<%= journalDisplayContext %>"
    infoPanelId="infoPanelId"
    namespace="<%= renderResponse.getNamespace() %>"
    searchContainerId="≤% searchContainerId %>"
/>
```

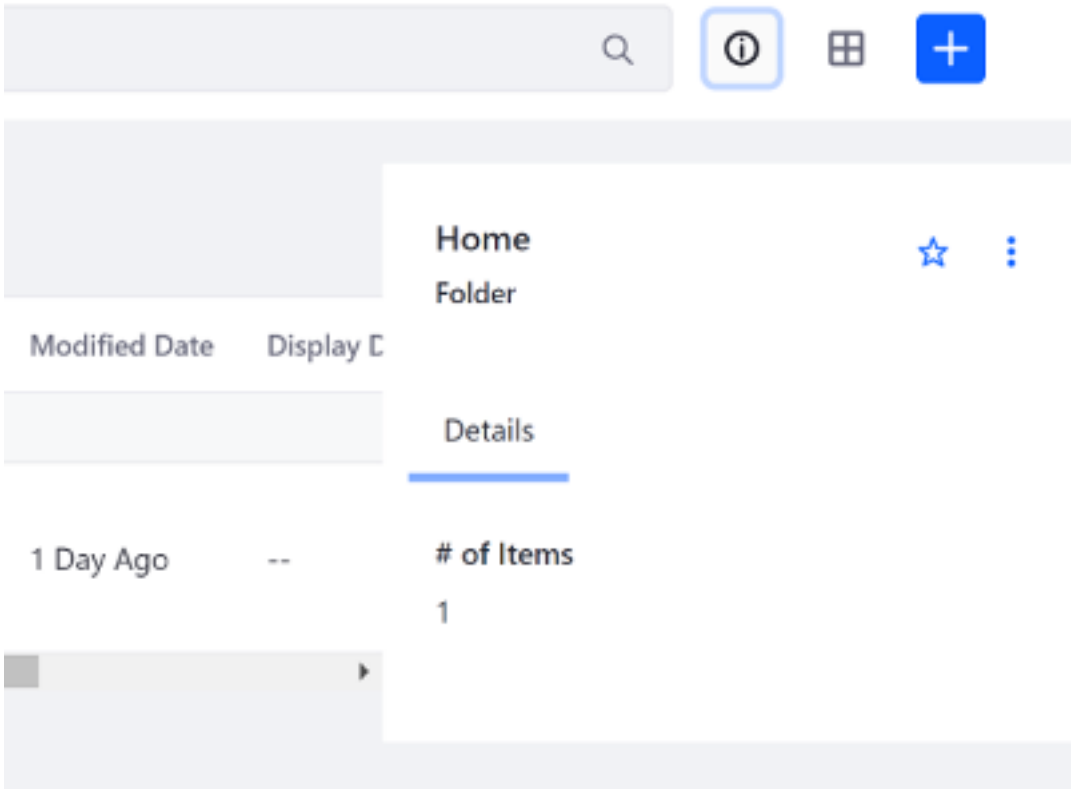


Figure 93.62: The info panel keeps your UI clutter-free.

View Types

The `viewTypes` attribute specifies the display options for the search container results. There are three display options to choose from:

Cards: Displays search result columns on a horizontal or vertical card.

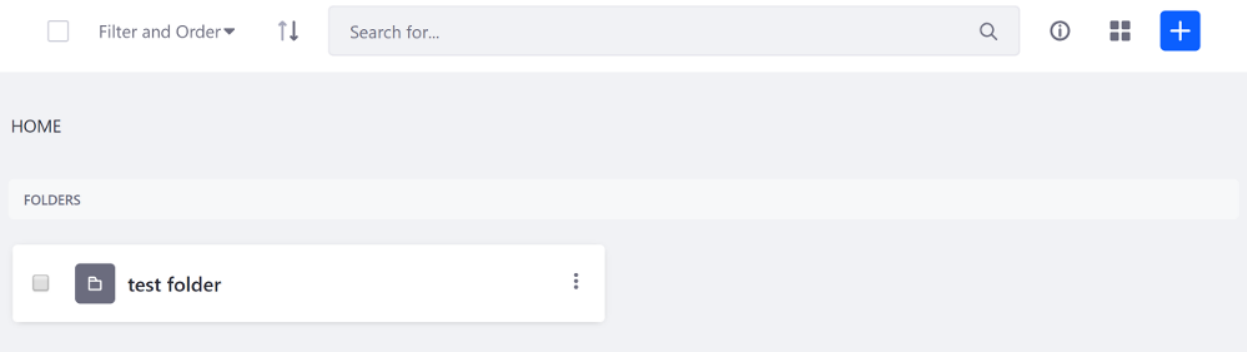


Figure 93.63: The Management Toolbar's icon display view gives a quick summary of the content's description and status.

List: Displays a detailed description along with summarized details for the search result columns.

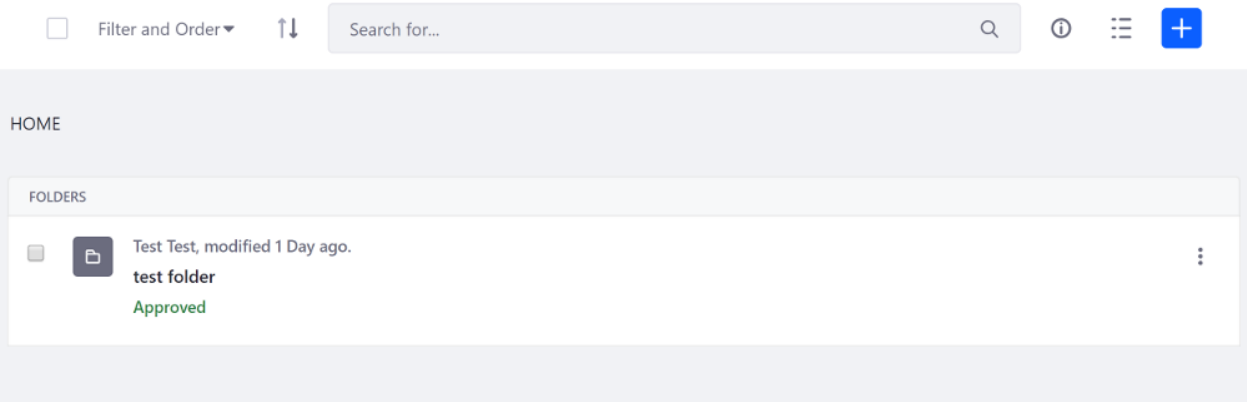


Figure 93.64: The Management Toolbar's List view type gives the content's full description.

Table: The default view. Lists the search result columns from left to right.

An example configuration is shown below:

```
viewTypes="<%=
    new JSPViewTypeItemList(pageContext, baseURL, selectedType) {
        {
            addCardViewTypeItem(
                viewTypeItem -> {
                    viewTypeItem.setActive(true);
                    viewTypeItem.setLabel("Card");
                });

            addListViewTypeItem(
                viewTypeItem -> {
                    viewTypeItem.setLabel("List");
```

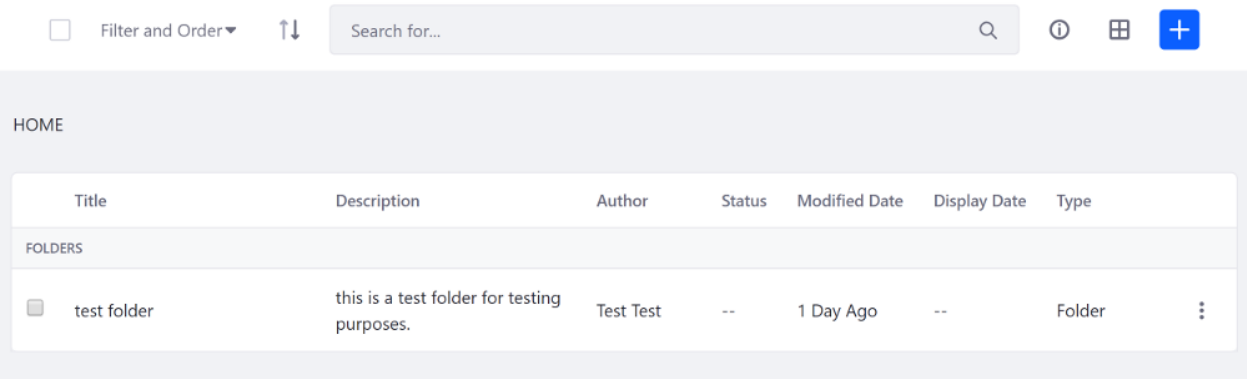


Figure 93.65: The Management Toolbar's Table view type list the content's information in individual columns.

```
                });

            addTableViewTypeItem(
                viewTypeItem -> {
                    viewTypeItem.setLabel("Table");
                });
        }
    }
%>"
```

While the example above shows how to configure the view types in the JSP, you must also specify when to use each view type.

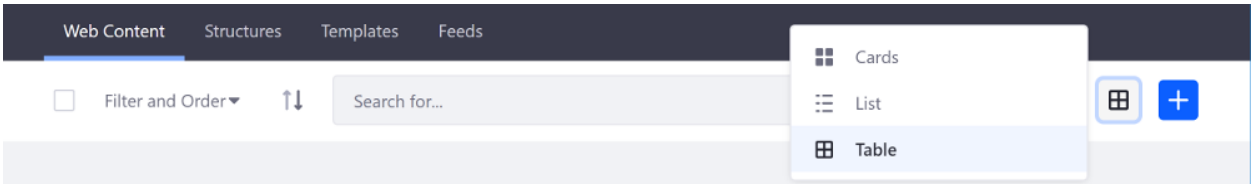


Figure 93.66: : The Management Toolbar offers three view type options.

Creation Menu

The `creationMenu` attribute creates an add menu button for one or multiple items. It's used for creating new entities (e.g. a new blog entry).

Use the `addPrimaryDropdownItem()` method to add the top level items to the dropdown menu, or use the `addFavoriteDropdownItem()` method to add secondary items to the dropdown menu.

The example configuration below adds two primary creation menu items and two secondary creation menu items:

```
creationMenu="<%=
    new JSPCreationMenu(pageContext) {
            {
                addPrimaryDropdownItem(
                    dropdownItem -> {
                        dropdownItem.setHref("#1");
                        dropdownItem.setLabel("Sample 1");
```

```
                });

            addPrimaryDropdownItem(
                dropdownItem -> {
                    dropdownItem.setHref("#2");
                    dropdownItem.setLabel("Sample 2");
                });

            addFavoriteDropdownItem(
                dropdownItem -> {
                    dropdownItem.setHref("#3");
                    dropdownItem.setLabel("Favorite 1");
                });

            addFavoriteDropdownItem(
                dropdownItem -> {
                    dropdownItem.setHref("#4");
                    dropdownItem.setLabel("Other item");
                });
        }
    };
%>"
```

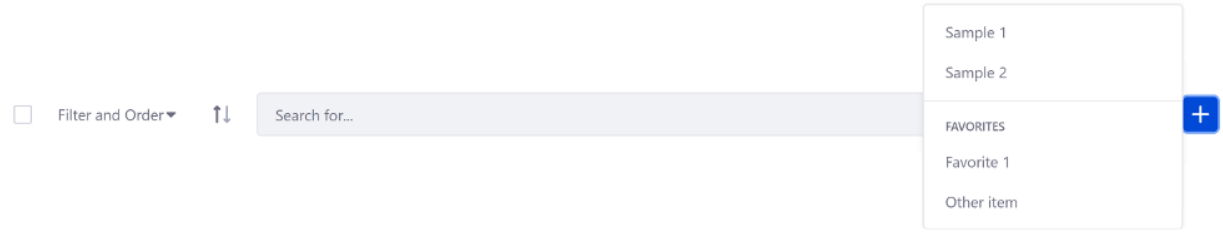


Figure 93.67: : The Management Toolbar lets you optionally add a Creation Menu for creating new entities.

Related Topics

Clay Dropdown Menus and Action Menus
    Clay Icons
    Clay Navigation Bars

93.10 Clay Navigation Bars

Similar to dropdown menus, navigation bars display a list of navigation items. The key difference is navigation bars are displayed in a horizontal bar with all navigation items visible at all times. The navigation bar also indicates the active navigation item with an underline. Navigation bars come in two styles: white background with dark-grey text (default) and dark-grey background with white text (inverted).

Default navigation bar:

```
<clay:navigation-bar
    navigationItems="<%= navigationBarsDisplayContext.getNavigationItems() %>"
/>
```

Inverted navigation bar (set `inverted` attribute to `true`):

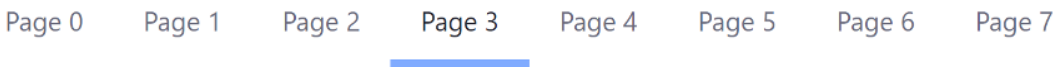


Figure 93.68: You can include navigation bars in your apps.

```
<clay:navigation-bar
    inverted="<%= true %>"
    navigationItems="<%= navigationBarsDisplayContext.getNavigationItems() %>"
/>
```

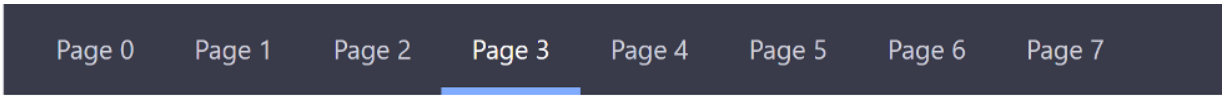


Figure 93.69: Navigation bars can be inverted if you prefer.

Related Topics

Clay Dropdown Menus and Action Menus
Clay Form Elements
Clay Progress Bars

93.11 Clay Progress Bars

You can add progress bars to your app with the `clay:progressbar` tag. These indicate the completion percentage of a task and come in three status styles: `default` (blue), `warning` (red), and `complete` (green with checkmark). You can provide a minimum value (`minValue`) and a maximum value (`maxValue`).

Default progress bar:

```
<clay:progressbar
    maxValue="<%= 100 %>"
    minValue="<%= 0 %>"
    value="<%= 30 %>"
/>
```


Figure 93.70: You can include progress bars in your apps.

Warning progress bar:

```
<clay:progressbar
    maxValue="<%= 100 %>"
    minValue="<%= 0 %>"
    status="warning"
    value="<%= 70 %>"
/>
```

Figure 93.71: warning progress bars indicate that the progress has not completed due to an error.

Complete progress bar:

```
<clay:progressbar
    status="complete"
/>
```

Figure 93.72: The complete progress bar indicates the progress is complete.

Clay taglibs make it easy to track progress in your apps.

Related Topics

Clay Dropdown Menus and Action Menus
Clay Icons
Clay Navigation Bars

93.12 Clay Stickers

Whereas badges display numbers and labels display short information, stickers are small visual indicators of the content (usually the content type). They can include a small label or a Liferay icon, and they come in two shapes: circle and square.

Square sticker with label:

```
<clay:sticker label="JPG" />
```

Figure 93.73: You can include stickers in your apps.

Square sticker with icon:

```
<clay:sticker icon="picture" />
```

Circle sticker:

```
<clay:sticker label="JPG" shape="circle" />
```

Stickers can be positioned in any corner of a div. Indicate their position with the `position` attribute: `top-left`, `bottom-left`, `top-right`, or `bottom-right`:

Figure 93.74: Stickers can include icons.

Figure 93.75: You can also have circle stickers.

```
<div class="aspect-ratio">

    <img class="aspect-ratio-item-fluid" src="https://claycss.com](./images/thumbnail_hot_air_ballon.jpg" />

    <clay:sticker label="PDF" position="top-left" style="danger" />
</div>
```

Figure 93.76: You can specify the position of the sticker within a container.

Now you know how to use Clay stickers in your app!

Related Topics

Clay Badges
    Clay Cards
    Clay Icons

CHAPTER 94

USING THE CHART TAGLIB IN YOUR PORTLETS

Lines, splines, bars, pies and more, the Chart tag Library provides everything you need to model data. Each taglib gives you access to the corresponding Clay component. These components contain the default configuration for the UI.

To use the Chart taglib in your apps, add the following declaration to your JSP:

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

This section of tutorials covers the types of charts you can create with the Chart taglibs. Each tutorial contains a set of chart examples along with sample Java data and a figure displaying the rendered results.

94.1 Bar Charts

Bar charts contain multiple sets of data. A bar chart models the data in bars. Each data series (created with the `addColumns()` method) is defined with a new instance of the `MultiValueColumn` object, which takes an ID and a set of values. This tutorial shows how to configure your portlet to use bar charts.

Follow these steps:

1. Import the chart taglib along with the `BarChartConfig` and `MultiValueColumn` classes into your bundle's `init.jsp` file:

   ```
   <%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
   <%@ page import="com.liferay.frontend.taglib.chart.model.point.bar.BarChartConfig" %>
   <%@ page import="com.liferay.frontend.taglib.chart.model.MultiValueColumn" %>
   ```

2. Add the following Java scriptlet to the top of your `view.jsp`:

   ```
   <%
   BarChartConfig _barChartConfig = new BarChartConfig();

   _barChartConfig.addColumns(
       new MultiValueColumn("data1", 100, 20, 30),
       new MultiValueColumn("data2", 20, 70, 100)
   ```

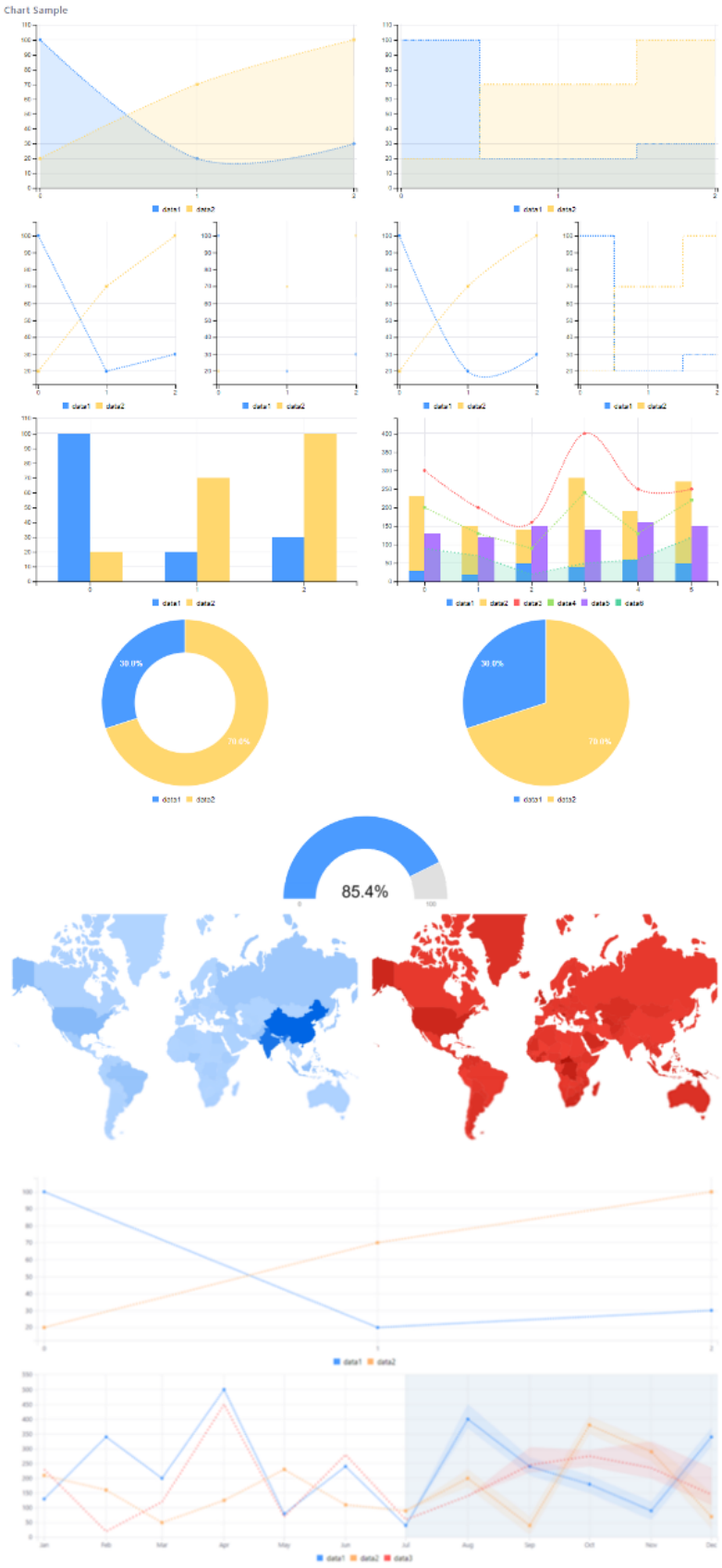


Figure 94.1: You can create many different types of charts with the chart taglibs.

```
);

%>
```

3. Add the `<chart>` taglib to the `view.jsp`, passing the `_barChartConfig` as the `config` attribute's value:

```
<chart:bar
  config="<%= _barChartConfig %>"
/>
```

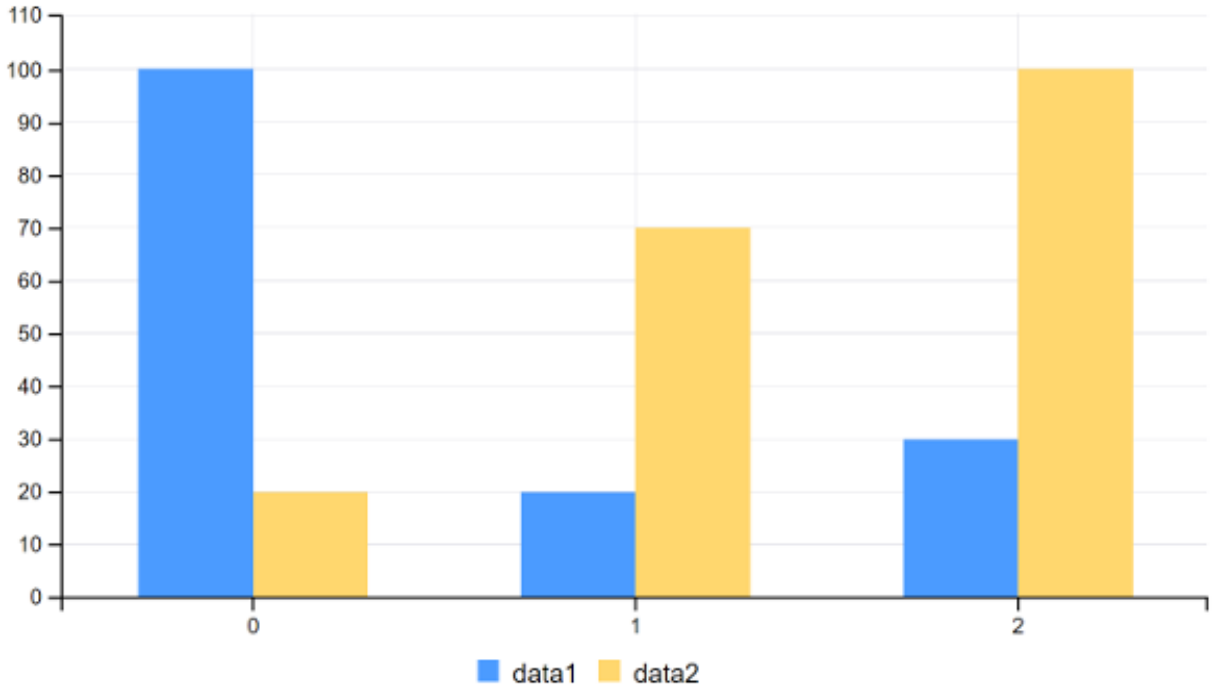


Figure 94.2: A bar chart models the data in bars.

Related Topics

Line Charts
    Donut Charts
    Combination Charts

94.2 Line Charts

Line charts contain multiple sets of data. A Line chart displays the data linearly. Each data series (created with the `addColumns()` method) is defined with a new instance of the `MultiValueColumn` object, which takes an ID and a set of values. This tutorial shows how to configure your portlet to use line charts.

Follow these steps:

1. Import the chart taglib along with the `LineChartConfig` and `MultiValueColumn` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.point.line.LineChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.MultiValueColumn" %>
```

2. Add the following Java scriptlet to the top of your `view.jsp`:

```
<%
LineChartConfig _lineChartConfig = new LineChartConfig();

_lineChartConfig.addColumns(
  new MultiValueColumn("data1", 100, 20, 30),
  new MultiValueColumn("data2", 20, 70, 100)
);
%>
```

3. Add the `<chart>` taglib to the `view.jsp`, passing the `_lineChartConfig` as the `config` attribute's value:

```
<chart:line
  config="<%= _lineChartConfig %>"
/>
```

Related Topics

Spline Charts
Step Charts
Predictive Charts

94.3 Scatter Charts

Scatter charts contain multiple sets of data. A scatter chart models the data as individual points. Each data series (created with the `addColumns()` method) is defined with a new instance of the `MultiValueColumn` object, which takes an ID and a set of values. This tutorial shows how to configure your portlet to use scatter charts.

Follow these steps:

1. Import the chart taglib along with the `ScatterChartConfig` and `MultiValueColumn` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.point.scatter.ScatterChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.MultiValueColumn" %>
```

2. Add the following Java scriptlet to the top of your `view.jsp`:

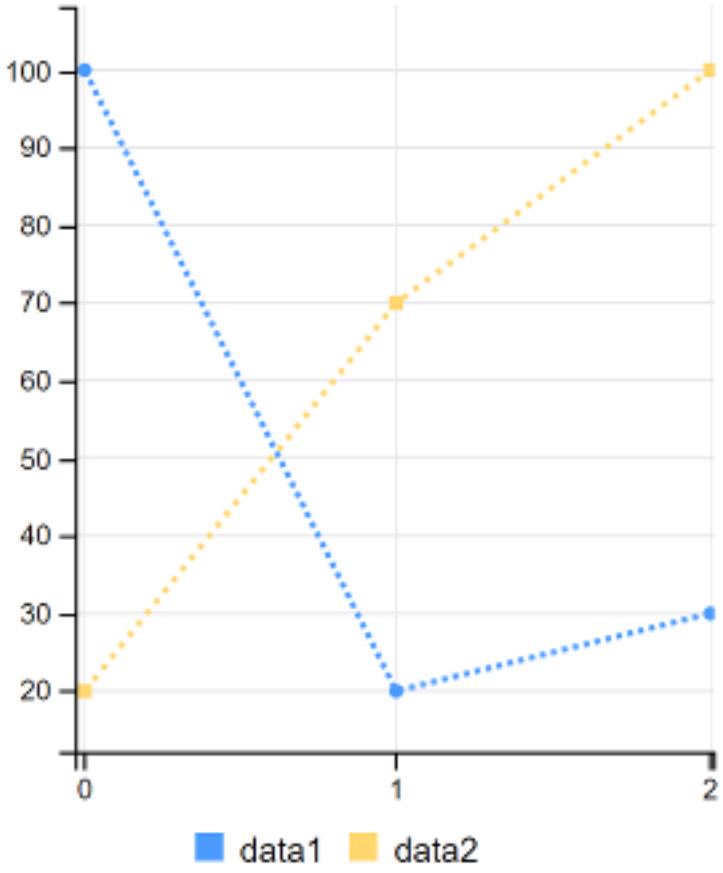


Figure 94.3: A Line chart displays the data linearly.

```
<%
ScatterChartConfig _scatterChartConfig = new ScatterChartConfig();

_scatterChartConfig.addColumns(
  new MultiValueColumn("data1", 100, 20, 30),
  new MultiValueColumn("data2", 20, 70, 100)
);
%>
```

3. Add the <chart> taglib to the `view.jsp`, passing the `_scatterChartConfig` as the `config` attribute's value:

```
<chart:scatter
  config="<%= _scatterChartConfig %>"
/>
```

Related Topics

Line Charts
    Step Charts
    Predictive Charts

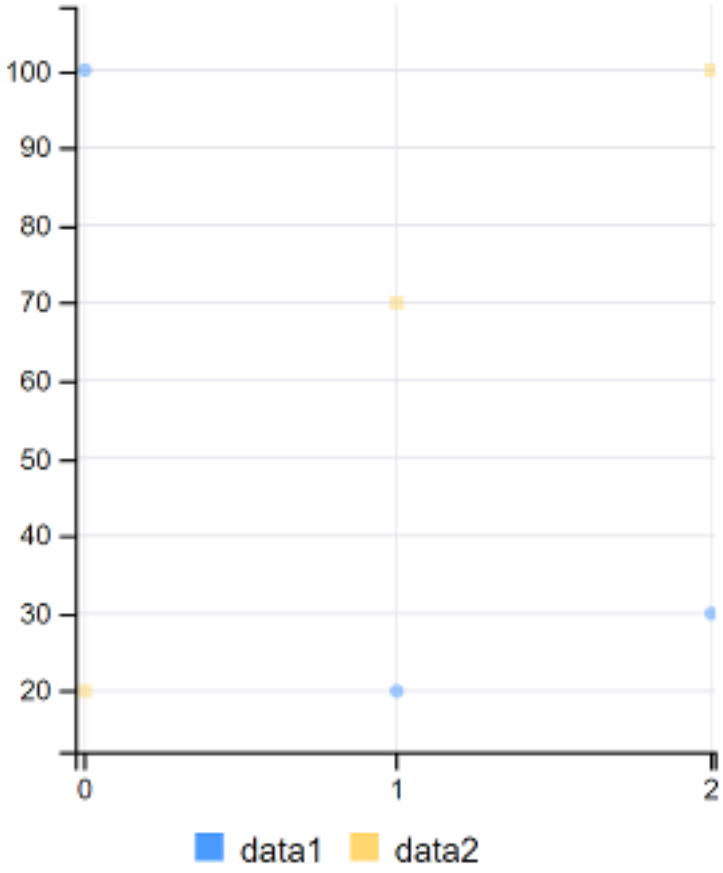


Figure 94.4: A scatter chart models the data as individual points.

94.4 Spline Charts

Spline charts contain multiple sets of data. A spline chart connects points of data with a smooth curve. Each data series (created with the `addColumns()` method) is defined with a new instance of the `MultiValueColumn` object, which takes an ID and a set of values. This tutorial shows how to configure your portlet to use spline charts.

Follow these steps:

1. Import the chart taglib along with the `SplineChartConfig` and `MultiValueColumn` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.point.spline.SplineChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.MultiValueColumn" %>
```

2. Add the following Java scriptlet to the top of your `view.jsp`:

```
<%
SplineChartConfig _splineChartConfig = new SplineChartConfig();
```

```
_splineChartConfig.addColumns(
  new MultiValueColumn("data1", 100, 20, 30),
  new MultiValueColumn("data2", 20, 70, 100)
);
%>
```

3. Add the <chart> taglib to the `view.jsp`, passing the `_splineChartConfig` as the `config` attribute's value:

```
<chart:spline
  config="<%= _splineChartConfig %>"
/>
```

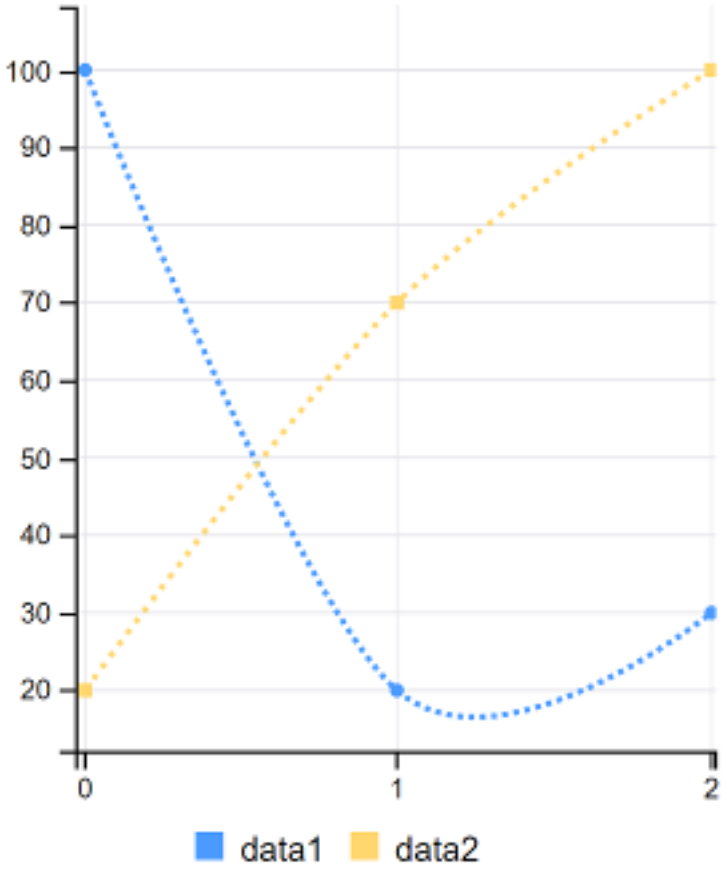


Figure 94.5: A spline chart connects points of data with a smooth curve.

You can also use an area spline chart if you prefer. An area spline chart highlights the area under the spline curve.

```
<chart:area-spline
  config="<%= _splineChartConfig %>"
/>
```

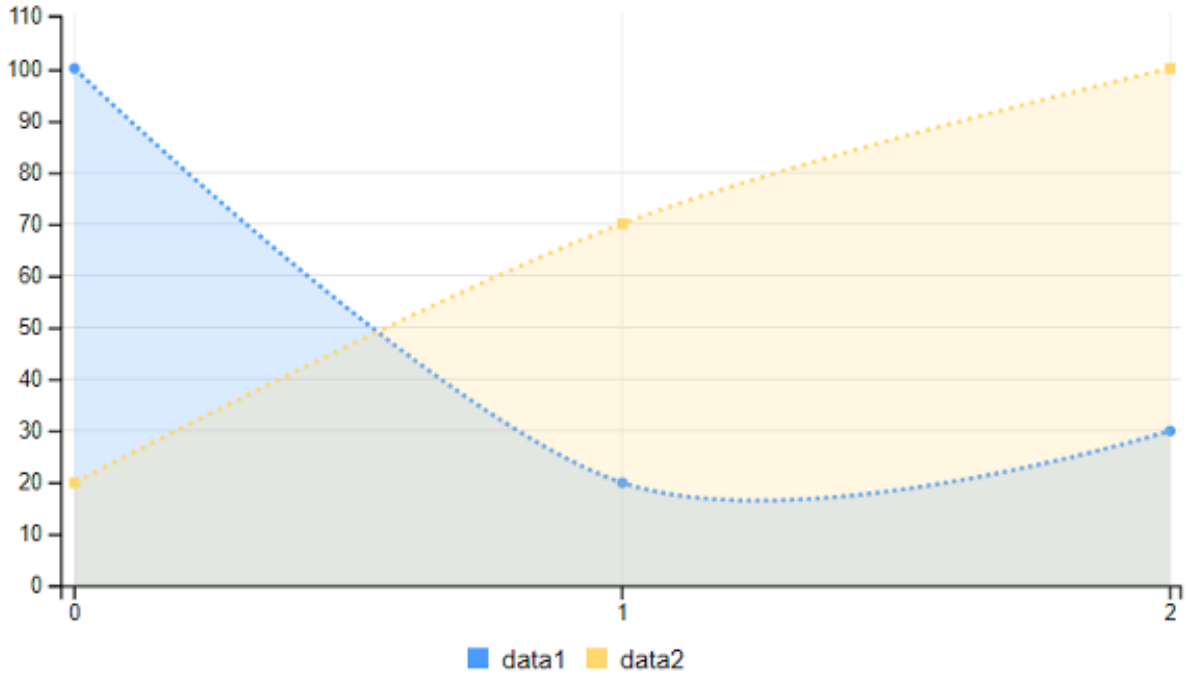


Figure 94.6: An area spline chart highlights the area under the spline curve.

Related Topics

Line Charts
    Step Charts
    Scatter Charts

94.5 Step Charts

Step charts contain multiple sets of data. A step chart steps between the points of data, resembling steps. Each data series (created with the `addColumns()` method) is defined with a new instance of the `MultiValueColumn` object, which takes an ID and a set of values. This tutorial shows how to configure your portlet to use step charts.

Follow these steps:

1. Import the chart taglib along with the `StepChartConfig` and `MultiValueColumn` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.point.step.StepChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.MultiValueColumn" %>
```

2. Add the following Java scriptlet to the top of your `view.jsp`:

```
<%
StepChartConfig _stepChartConfig = new StepChartConfig();
```

```
_stepChartConfig.addColumns(
  new MultiValueColumn("data1", 100, 20, 30),
  new MultiValueColumn("data2", 20, 70, 100)
);
%>
```

3. Add the <chart> taglib to the `view.jsp`, passing the `_stepChartConfig` as the `config` attribute's value:

```
<chart:step
  config="<%= _stepChartConfig %>"
/>
```

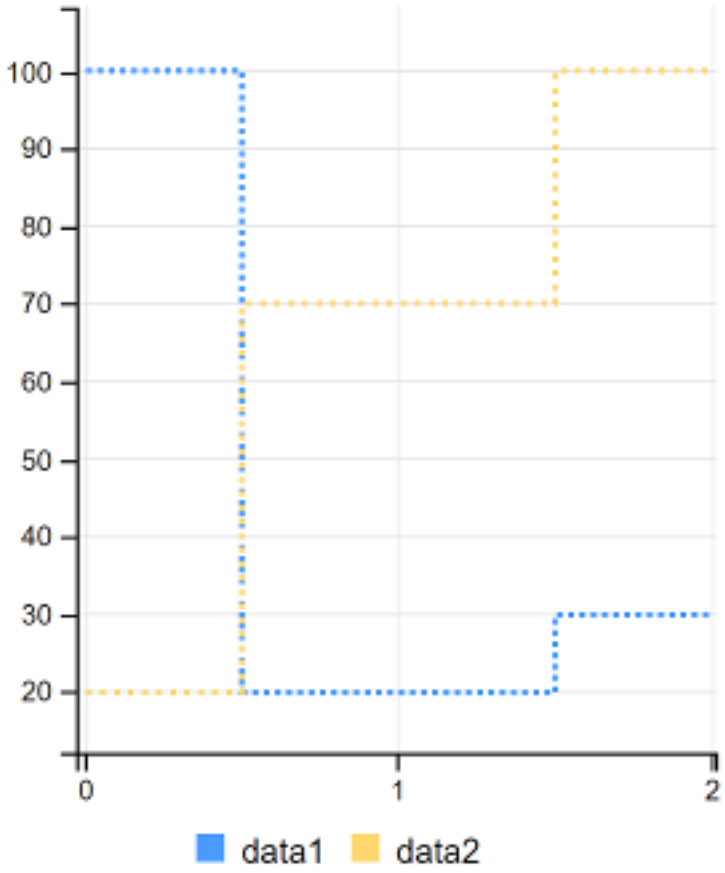


Figure 94.7: A step chart steps between the points of data, resembling steps.

You can also use an area step chart if you prefer. An area step chart highlights the area covered by a step graph.

```
<chart:area-step
  config="<%= _stepChartConfig %>"
/>
```

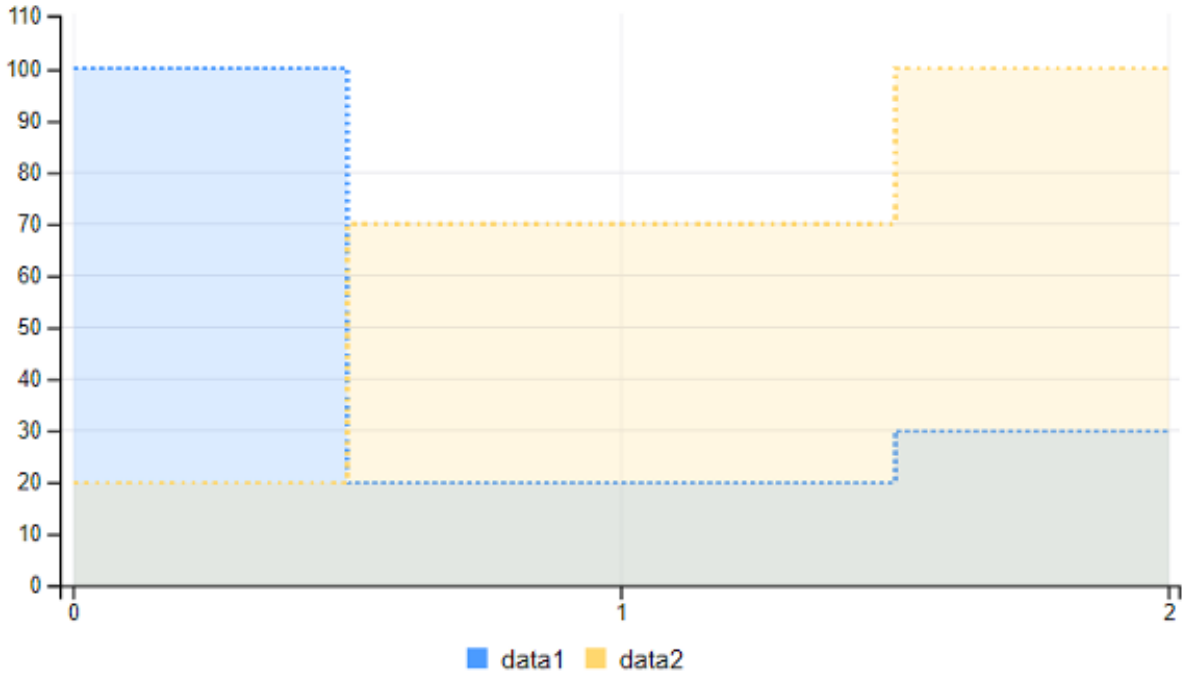


Figure 94.8: An area step chart highlights the area covered by a step graph.

Related Topics

Line Charts
    Spline Charts
    Scatter Charts

94.6 Combination Charts

Combination charts have minor differences from other charts. In a combination chart, you must define the representation type of each data set: `AREA`, `AREA_SPLINE`, `AREA_STEP`, `BAR`, `BUBBLE`, `DONUT`, `GAUGE`, `LINE`, `PIE`, `SCATTER`, `SPLINE`, or `STEP`. Each data set in a combination chart is an instance of the `TypedMultiValueColumn` object. Each object receives an ID, the representation type, and values for the data. This tutorial shows how to configure your portlet to use combination charts.

Follow these steps:

1. Import the chart taglib along with the `CombinationChartConfig`, `MultiValueColumn`, and `MultiValueColumn.Type` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.combination.CombinationChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.MultiValueColumn" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.TypedMultiValueColumn.Type" %>
```

2. Add the following Java scriptlet to the top of your `view.jsp`:

```
<%
CombinationChartConfig _combinationChartConfig =
new CombinationChartConfig();

_combinationChartConfig.addColumns(
  new TypedMultiValueColumn(
    "data1", Type.BAR, 30, 20, 50, 40, 60, 50),
  new TypedMultiValueColumn(
    "data2", Type.BAR, 200, 130, 90, 240, 130, 220),
  new TypedMultiValueColumn(
    "data3", Type.SPLINE, 300, 200, 160, 400, 250, 250),
  new TypedMultiValueColumn(
    "data4", Type.LINE, 200, 130, 90, 240, 130, 220),
  new TypedMultiValueColumn(
    "data5", Type.BAR, 130, 120, 150, 140, 160, 150),
  new TypedMultiValueColumn(
    "data6", Type.AREA, 90, 70, 20, 50, 60, 120)
  );

_combinationChartConfig.addGroup("data1", "data2");
%>
```

3. Add the <chart> taglib to the `view.jsp`, passing the `_combinationChartConfig` as the `config` attribute's value:

```
<chart:combination
  config="<%= _combinationChartConfig %>"
/>
```

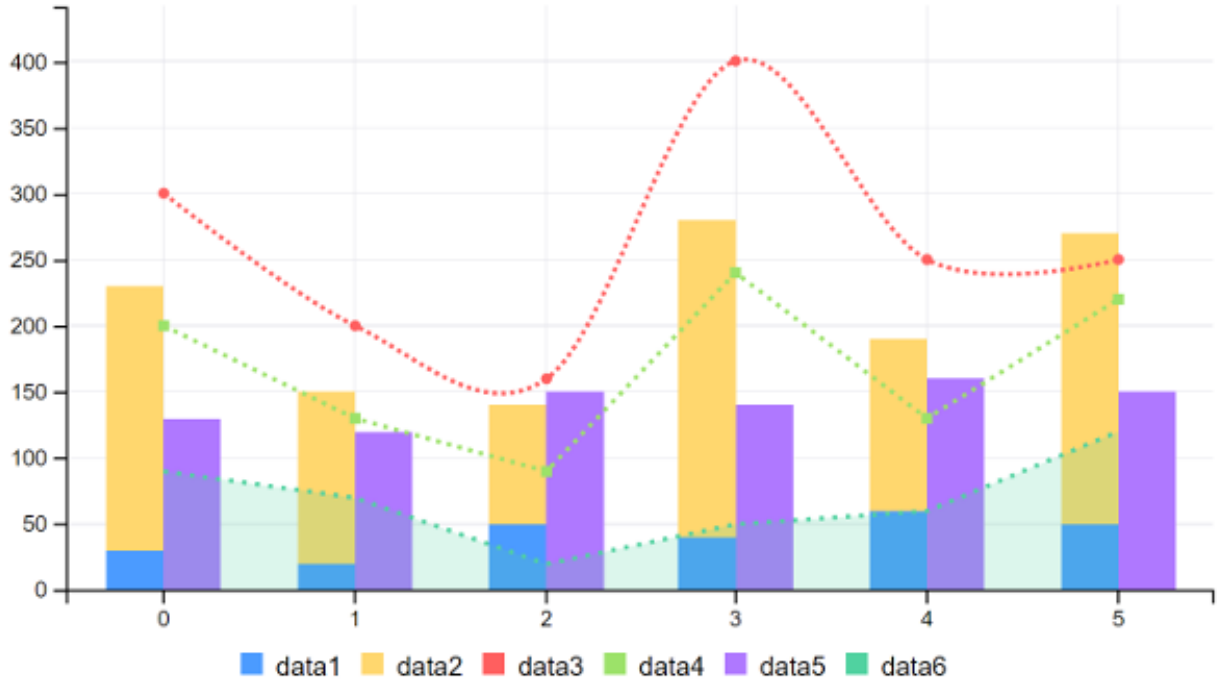


Figure 94.9: A combination chart displays a variety of data set types.

Related Topics

Bar Charts
    Line Charts
    Geomap Charts

94.7 Donut Charts

Donut charts are percentage-based. A donut chart is similar to a pie chart, but it has a hole in the center. Each data set must be defined as a new instance of the `SingleValueColumn` object. This tutorial shows how to configure your portlet to use donut charts.

Follow these steps:

1. Import the chart taglib along with the `DonutChartConfig` and `SingleValueColumn` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.percentage.donut.DonutChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.SingleValueColumn" %>
```

2. Add the following Java scriptlet to the top of your `view.jsp`:

```
<%
DonutChartConfig _donutChartConfig = new DonutChartConfig();

_donutChartConfig.addColumns(
  new SingleValueColumn("data1", 30),
  new SingleValueColumn("data2", 70)
);
%>
```

3. Add the `<chart>` taglib to the `view.jsp`, passing the `_donutChartConfig` as the `config` attribute's value:

```
<chart:donut
  config="<%= _donutChartConfig %>"
/>
```

Related Topics

Pie Charts
    Gauge Charts
    Bar Charts

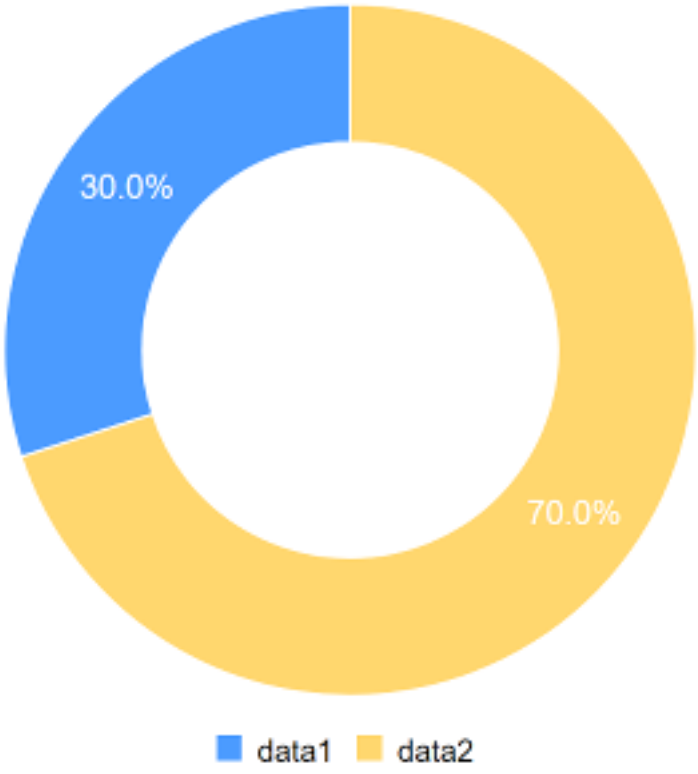


Figure 94.10: A donut chart is similar to a pie chart, but it has a hole in the center.

94.8 Gauge Charts

Gauge charts are percentage-based. A gauge chart shows where percentage-based data falls over a given range. Each data set must be defined as a new instance of the `SingleValueColumn` object. This tutorial shows how to configure your portlet to use gauge charts.

Follow these steps:

1. Import the chart taglib along with the `GaugeChartConfig` and `SingleValueColumn` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.gauge.GaugeChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.SingleValueColumn" %>
```

2. Add the following Java scriptlet to the top of your `view.jsp`:

```
<%
GaugeChartConfig _gaugeChartConfig = new GaugeChartConfig();

_gaugeChartConfig.addColumn(
  new SingleValueColumn("data1", 85.4)
);
%>
```

3. Add the <chart> taglib to the view.jsp, passing the _gaugeChartConfig as the config attribute's value:

```
<chart:gauge
  config="<%= _gaugeChartConfig %>"
/>
```

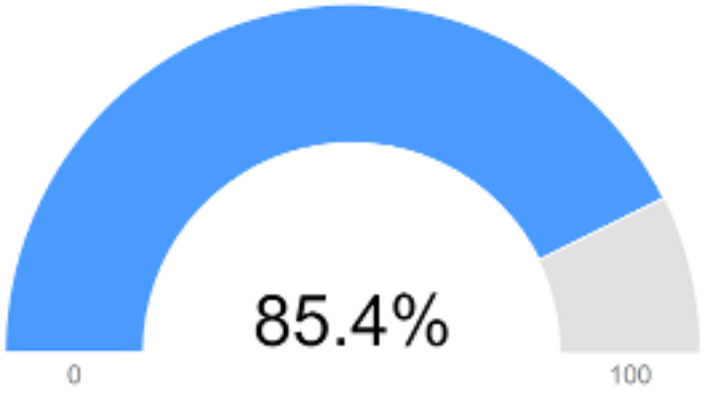


Figure 94.11: A gauge chart shows where percentage-based data falls over a given range.

Related Topics

Pie Charts
    Donut Charts
    Bar Charts

94.9 Pie Charts

Pie charts are percentage-based. A pie chart models percentage-based data as individual slices of pie. Each data set must be defined as a new instance of the SingleValueColumn object. This tutorial shows how to configure your portlet to use pie charts.

Follow these steps:

1. Import the chart taglib along with the PieChartConfig and SingleValueColumn classes into your bundle's init.jsp file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.percentage.pie.PieChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.SingleValueColumn" %>
```

2. Add the following Java scriptlet to the top of your view.jsp:

```
<%
PieChartConfig _pieChartConfig = new PieChartConfig();

_pieChartConfig.addColumn(
  new SingleValueColumn("data1", 85.4)
);
%>
```

3. Add the <chart> taglib to the `view.jsp`, passing the `_pieChartConfig` as the `config` attribute's value:

```
<chart:pie
  config="<%= _pieChartConfig %>"
/>
```

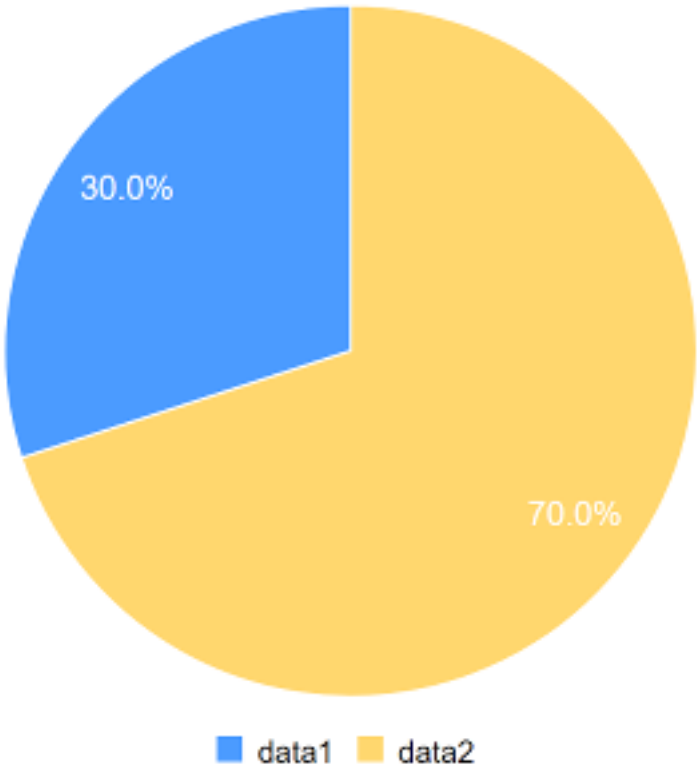


Figure 94.12: A pie chart models percentage-based data as individual slices of pie.

Related Topics

Donut Charts
    Gauge Charts
    Spline Charts

94.10 Geomap Charts

A Geomap Chart lets you visualize data based on geography, given a specified color range—a lighter color representing a lower rank and a darker a higher rank usually. The default configuration comes from the Clay charts geomap component: which ranges from light-blue (#b1d4ff) to dark-blue (#0065e4) and ranks the geography based on the location's `pop_est` value (specified in the geomap's JSON file).

Figure 94.13: A Geomap chart displays a heatmap representing the data.

This tutorial shows how to configure your portlet to use geomap charts. Follow these steps:

1. Import the chart taglib along with the `GeomapConfig`, `GeomapColor`, and `GeomapColorRange` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.geomap.GeomapConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.geomap.GeomapColor" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.geomap.GeomapColorRange" %>
```

2. Add the following Java scriptlet to the top of your `view.jsp`. The colors—a color for minimum and a color for maximum—are completely configurable, as shown in the second example

configuration below: `_geomapConfig2`. Create a new `GeomapColorRange` and set the minimum and maximum color values with the `setMax()` and `setMin()` methods. Specify the highlight color—the color displayed when you mouse over an area—with the `setSelected()` method. use the `geomapColor.setValue()` method to specify the JSON property to determine the geomap's ranking. Specify the JSON filepath with the `setDataHREF()` method. The example below displays a geomap based on the length of each location's name:

```
<%
GeomapConfig _geomapConfig1 = new GeomapConfig();
GeomapConfig _geomapConfig2 = new GeomapConfig();

GeomapColor geomapColor = new GeomapColor();
GeomapColorRange geomapColorRange = new GeomapColorRange();

geomapColorRange.setMax("#b2150a");
geomapColorRange.setMin("#ee3e32");

geomapColor.setGeomapColorRange(geomapColorRange);

geomapColor.setSelected("#a9615c");

geomapColor.setValue("name_len");

_geomapConfig2.setColor(geomapColor);

StringBuilder sb = new StringBuilder();

sb.append(_portletRequest.getScheme());
sb.append(StringPool.COLON);
sb.append(StringPool.SLASH);
sb.append(StringPool.SLASH);
sb.append(_portletRequest.getServerName());
sb.append(StringPool.COLON);
sb.append(_portletRequest.getServerPort());
sb.append(_portletRequest.getContextPath());
sb.append(StringPool.SLASH);
sb.append("geomap.geo.json");

_geomapConfig1.setDataHREF(sb.toString());
_geomapConfig2.setDataHREF(sb.toString());
%>
```

3. Add the `<chart>` taglib to the `view.jsp` along with any styling information for the geomap, such as the size and margins as shown below:

```
<style type="text/css">
    .geomap {
        margin: 10px 0 10px 0;
    }
    .geomap svg {
        width: 100%;
        height: 500px !important;
    }
</style>

<chart:geomap
    config="<%= _geomapConfig1 %>"
    id="geomap-default-colors"
/>

<chart:geomap
    config="<%= _geomapConfig2 %>"
    id="geomap-custom-colors"
/>
```

Figure 94.14: Geomap charts can be customized to fit the look and feel you desire.

Related Topics

Bar Charts
    Pie Charts
    Combination Charts

94.11 Predictive Charts

Predictive charts let you visualize current data along with predicted/forecasted data within a given value range.

This tutorial shows how to configure your portlet to use predictive charts. Follow these steps:

1. Import the chart taglib along with the `PredictiveChartConfig` and `MixedDataColumn` classes into your bundle's `init.jsp` file:

```
<%@ taglib prefix="chart" uri="http://liferay.com/tld/chart" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.predictive.PredictiveChartConfig" %>
<%@ page import="com.liferay.frontend.taglib.chart.model.MixedDataColumn" %>
```

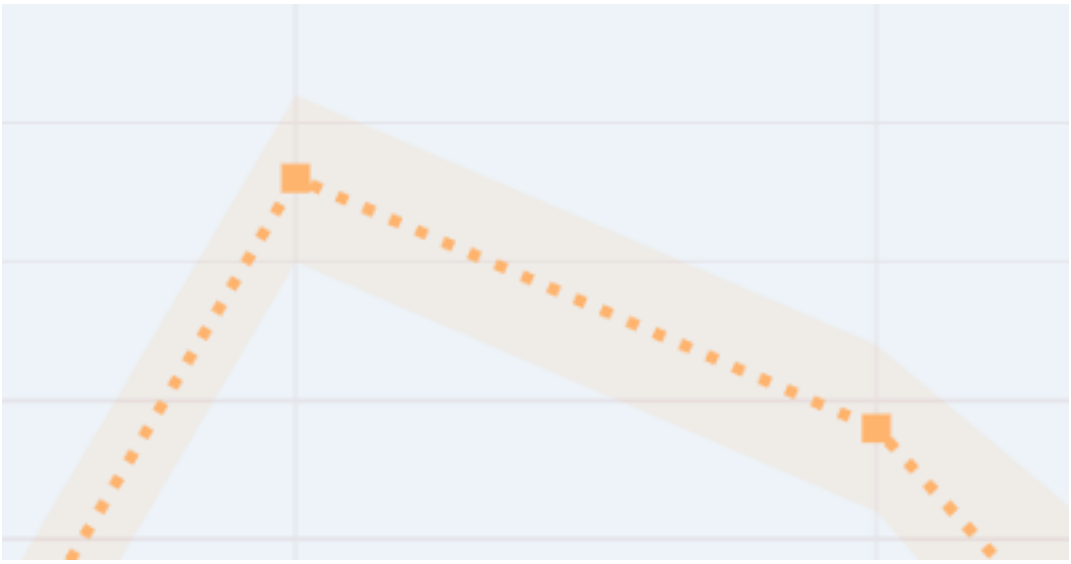


Figure 94.15: Predicted/forecasted data is surrounded by a highlighted area of possible values.

2. Add the following Java scriptlet to the top of your `view.jsp`. Add a `MixedDataColumn` object —a column that supports both single number values and arrays of three numbers—for each data series. Single number values define existing data. Arrays of numbers are used as the prediction/forecast data and contain three numbers: a minimum value, an estimated value, and a maximum value. The estimated value is rendered solid and surrounded by a highlighted area with borders specified by the minimum and maximum values. This lets you visualize your estimated values, while also giving you an idea of the possible value ranges. Use the `addDataColumn()` method to add each data series:

```
<%
private PredictiveChartConfig _predictiveChartConfig = new
PredictiveChartConfig();

MixedDataColumn mixedDataColumn1 = new MixedDataColumn(
  "data1", 130, 340, 200, 500, 80, 240, 40,
  new Number[] {370, 400, 450}, new Number[] {210, 240, 270},
  new Number[] {150, 180, 210}, new Number[] {60, 90, 120},
  new Number[] {310, 340, 370}
);

_predictiveChartConfig.addDataColumn(mixedDataColumn1);

MixedDataColumn mixedDataColumn2 = new MixedDataColumn(
  "data2", 210, 160, 50, 125, 230, 110, 90,
  Arrays.asList(170, 200, 230), Arrays.asList(10, 40, 70),
  Arrays.asList(350, 380, 410), Arrays.asList(260, 290, 320),
  Arrays.asList(30, 70, 150)
);

_predictiveChartConfig.addDataColumn(mixedDataColumn2);

_predictiveChartConfig.setAxisXTickFormat("%b");

_predictiveChartConfig.setPredictionDate("2018-07-01");

List<String> timeseries = new ArrayList<>();

timeseries.add("2018-01-01");
```

```
timeseries.add("2018-02-01");
timeseries.add("2018-03-01");
timeseries.add("2018-04-01");
timeseries.add("2018-05-01");
timeseries.add("2018-06-01");
timeseries.add("2018-07-01");
timeseries.add("2018-08-01");
timeseries.add("2018-09-01");
timeseries.add("2018-10-01");
timeseries.add("2018-11-01");
timeseries.add("2018-12-01");

_predictiveChartConfig.setTimeseries(timeseries);
%>
```

Predictive charts have the following properties:

axisXTickFormat: An optional string which specfies the time formatting on the X axis. For more information on which formats can be specified please refer to d3's time format README. This value is set using the `setAxisXTickFormat()` method.

Prediction Date: A date as a string that represents the point in the timeline from when the forecast/prediction is shown. This value is parsed as a Date object in JavaScript and set using the `setPredictionDate()` method.

Time Series: A timeline for the data which is displayed on the X axis of the chart. This value is set as an array of dates (`2018-01-01` for example).

3. Add the `<chart>` taglib to the `view.jsp`, passing the `_predictiveChartConfig` as the `config` attribute's value:

```
<chart:predictive
  config="<%= _predictiveChartConfig %>"
/>
```

The area contained within the light-blue rectangle is the point from which the predicted/forecasted values are shown:

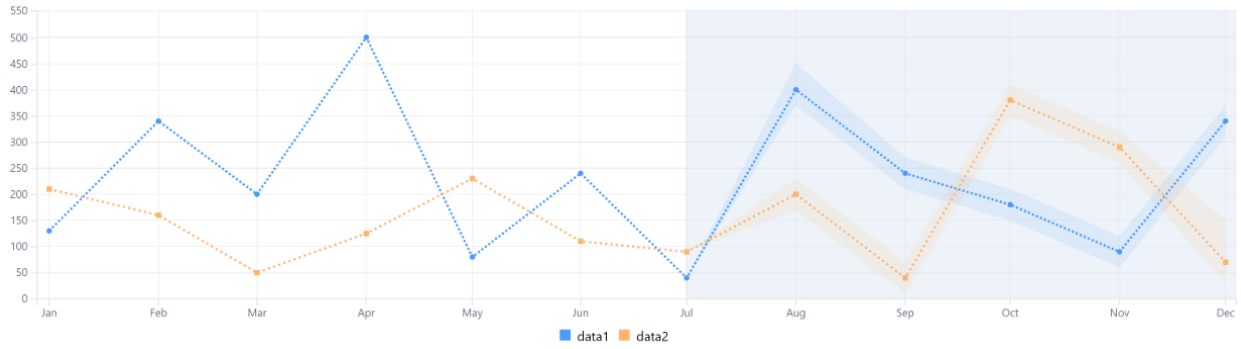


Figure 94.16: A predictive chart lets you visualize estimated future data alongside existing data.

Related Topics

Line Charts
    Combination Charts
    Geomap Charts

94.12 Refreshing Charts to Reflect Real Time Data

The polling interval property is an optional property for all charts. It specifies the time in milliseconds for the chart's data to refresh. You can use this for charts that receive any kind of real time data, such as a JSON file that changes periodically. This ensures that the chart is up to date, reflecting the most recent data. This tutorial shows how to configure your portlet's chart to reflect real time data.

Use the `setPollingInterval()` method for your chart's configuration object to specify the refresh rate. An example `view.jsp` configuration is shown below:

```
<%
LineChartConfig _pollingIntervalLineChartConfig = new LineChartConfig();

_pollingIntervalLineChartConfig.put("data", "/foo.json");

_pollingIntervalLineChartConfig.setPollingInterval(2000);

%>

<chart:line
    componentId="polling-interval-line-chart"
    config="<%= _pollingIntervalLineChartConfig %>"
/>
```

Figure 94.17: The polling interval property lets you refresh charts at a given interval to reflect real time data.

Related Topics

Bar Charts
    Scatter Charts
    Donut Charts

CHAPTER 95

USING AUI TAGLIBS

The AUI tag library provides tags that implement commonly used UI components. These tags make your markup consistent, responsive, and accessible.

You can find a list of the available <aui> taglibs in the AUI taglibdocs. 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. You'll find the full markup generated by the tags in their JSPs in their Liferay Github Repo folders.

To use the AUI taglib library in your apps, you must add the following declaration to your JSP:

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

The AUI taglib is also available via a macro for your FreeMarker theme templates and web content templates. Follow this syntax:

```
<@liferay_aui["tag-name"] attribute="string value" attribute=10 />
```

This section of tutorials covers how to create UI components with the AUI taglibs. Each tutorial contains code examples along with a screenshot of the resulting UI.

95.1 Building Forms with AUI Tags

The AUI tag library provides all the components you need to build forms for your applications. AUI tags provide many benefits to standard form elements, such as custom namespacing, localization, and even validation. They provide multiple attributes that let you create the experience you want for your users.

This tutorial shows you how to build forms using AUI tags. Follow these steps to build a form:

1. Add the aui taglib declaration to your portlet's `view.jsp` if you haven't already:

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

2. Build your form using the tags shown below. Each tag links to the corresponding taglibdoc that list the available attributes:

- <aui:input>
- <aui:button>
- <aui:button-row>
- <aui:container>
- <aui:col>
- <aui:row>
- <aui:field-wrapper>
- <aui:fieldset>
- <aui:fieldset-group>
- <aui:form>
- <aui:select>
- <aui:option>

An example form is shown below:

```
<aui:form name="fm">
    <aui:fieldset-group markupView="lexicon">
        <aui:fieldset label="Personal Information">
            <aui:row>
                <aui:col width="50">
                    <aui:input label="First Name" name="firstName" type="text" />
                </aui:col>
                <aui:col width="50">
                    <aui:input label="Last Name" name="lastName" type="text" />
                </aui:col>
            </aui:row>
            <aui:row>
                <aui:col width="50">
                    <aui:input label="Username" name="username" type="text" />
                </aui:col>
                <aui:col width="50">
                    <aui:input label="Email" name="email" type="email" />
                </aui:col>
            </aui:row>
        </aui:fieldset>
    </aui:fieldset-group>
    <aui:fieldset-group markupView="lexicon">
        <aui:fieldset label="Miscellaneous">
            <aui:input label="Hobbies" name="hobbies" type="textarea" />
            <aui:input label="Receive email updates" name="emailUpdates" type="checkbox" />
        </aui:fieldset>
    </aui:fieldset-group>
    <aui:button-row>
        <aui:button name="submitButton" type="submit" value="Submit" />
    </aui:button-row>
</aui:form>
```

3. Optionally add validation to your form fields. Nest a <aui:validator> tag inside each form field that you want to validate. Specify the validation rule with the <aui:validator> tag's name attribute (The available validation rules are shown in the table below). You can override a field's default validation error message with the errorMessage attribute. An example configuration is shown below:

```
<aui:form name="myForm">
    <aui:input name="password" id="password" label="Password"
    required="true" />
    <aui:input name="confirmPassword" id="password"
    label="Confirm Password" required="true">
```

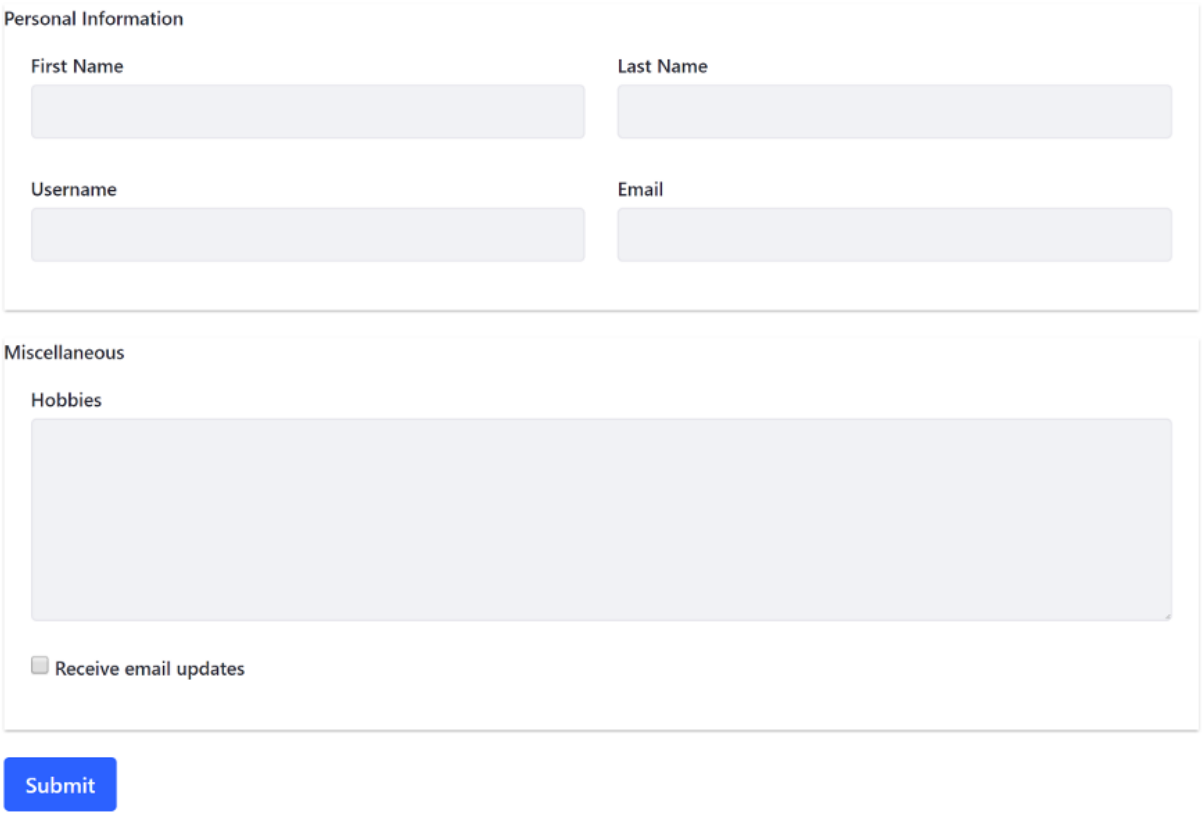


Figure 95.1: The AUI tags provide everything you need to build forms for your applications.

```
        <aui:validator name="equalTo"
        errorMessage="The passwords much match. Please try again." >
        '#<portlet:namespace>password'
        </aui:validator>
    </aui:input>
</aui:form>
```

The full list of available validation rules is shown in the table below:

```
Rule | Description | Default Error Message |
--- | --- | --- |
`acceptFiles` | Specifies that the field can only contain the file types given. Each file extension must be separated by a comma. For example </br>
`alpha` | Permits alphabetic characters | 'Please enter only alpha characters.' |
`alphanum` | Permits alphanumeric characters | 'Please enter only alphanumeric characters.' |
`date` | Permits dates | 'Please enter a valid date.' |
`digits` | Permits digits | 'Please enter only digits.' |
`email` | Permits an email address | 'Please enter a valid email address.' |
`equalTo` | Permits contents equal to another field with the specified field ID. For example, </br> `<aui:validator name="equalTo">'#<portlet:namesp
`max` | Permits an integer value less than the specified value. For example, a max value of 20 is specified with </br> `<aui:validator name="max">20
`maxLength` | Permits a maximum field length of the specified size (follows the same syntax as `max`) | 'Please enter no more than [max] characters.
`min` | Permits an integer value greater than the specified minimum value (follows the same syntax as `max`) | 'Please enter a value greater than or
`minLength` | Permits a field length longer than the specified size (follows the same syntax as `max`). | 'Please enter at least [min] characters.'
`number` | Permits numerical values | 'Please enter a valid number.' |
`range` | Permits a number between the specified range. For example, a range between 1.23 and 10 is specified here </br> `<aui:validator name="range
`rangeLength` | Permits a field length between the specified range (follows the same syntax as `range`)  | 'Please enter a value between [0] and [1]
```

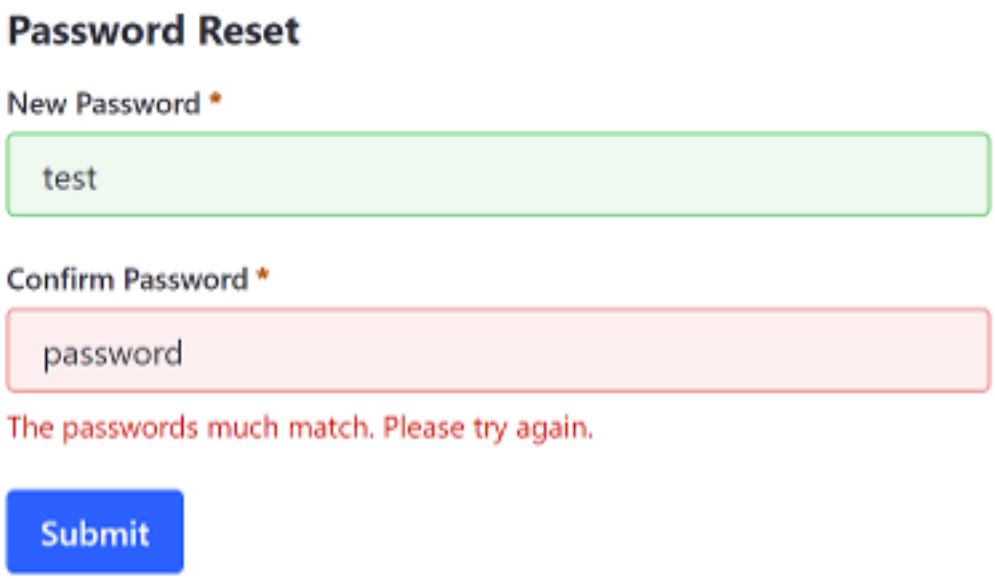


Figure 95.2: The AUI tags also provide validation for form fields.

```
`required` | Prevents a blank field  | 'This field is required.' |
`url` | Permits a URL value | 'Please enter a valid URL.' |
```

Now you know how to build user-friendly forms for your applications.

Related Topics

Using the Chart Taglib in Your Portlets
Using Liferay Front-end Taglibs in Your Portlet
Using the Clay Taglib in Your portlets

CHAPTER 96

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 97

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.

97.1 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.

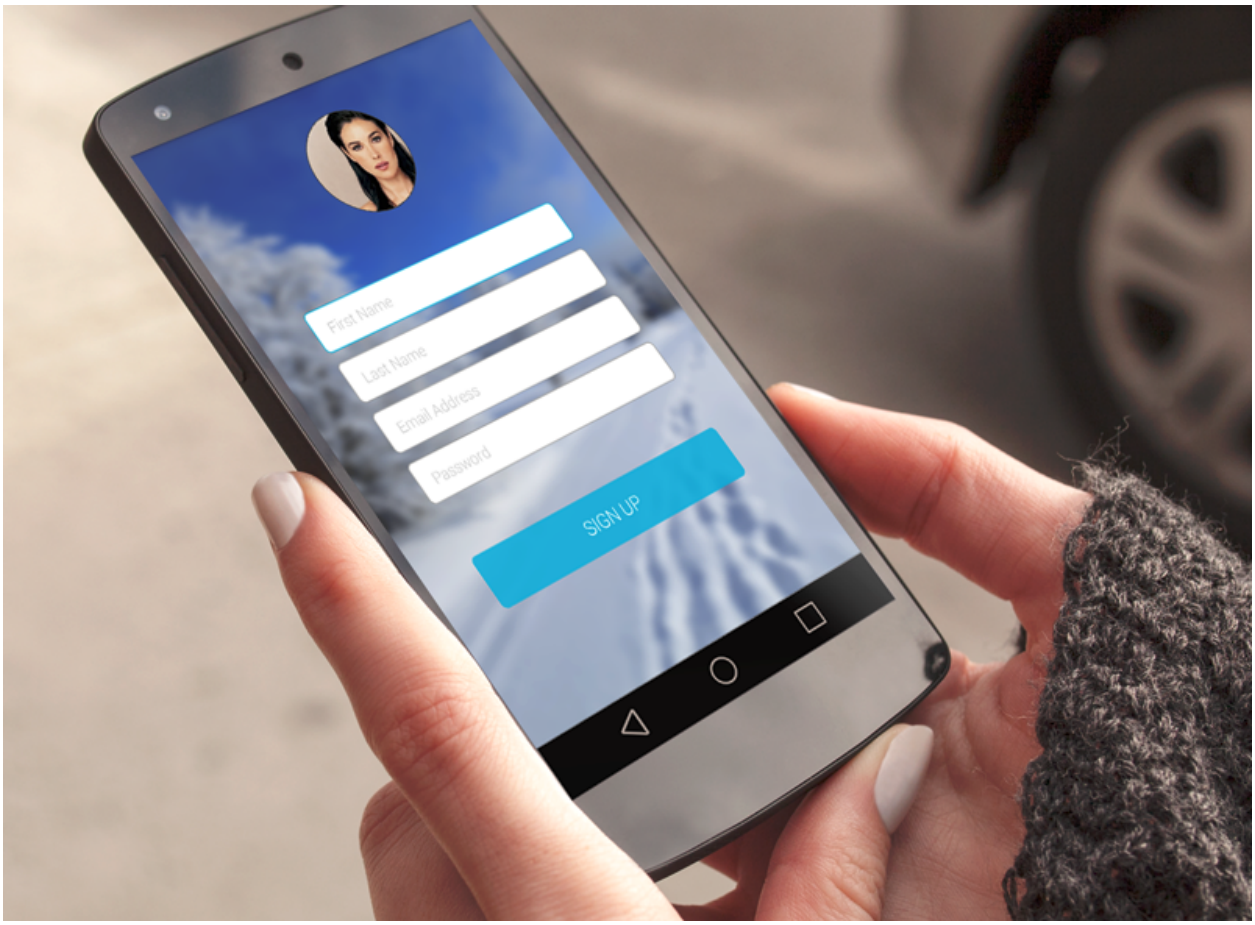


Figure 97.1: Here's an app that uses a Liferay Screens Sign Up Screenlet.

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.
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
- 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.
- 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:

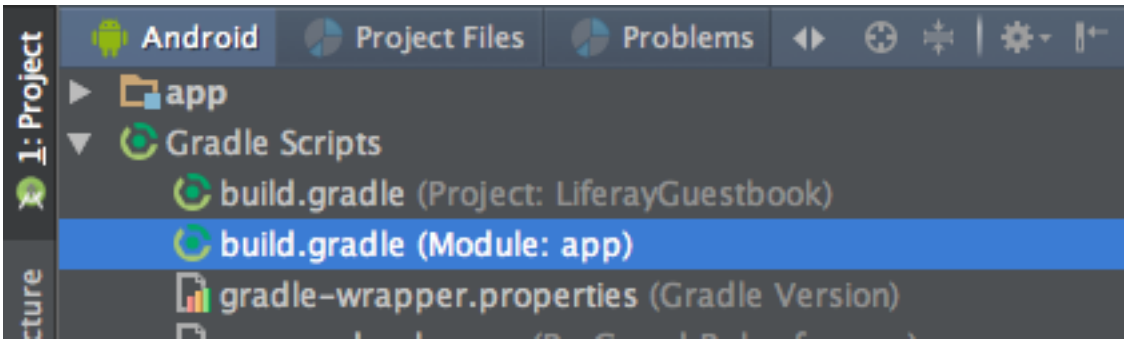


Figure 97.2: The app module's `build.gradle` file.

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:

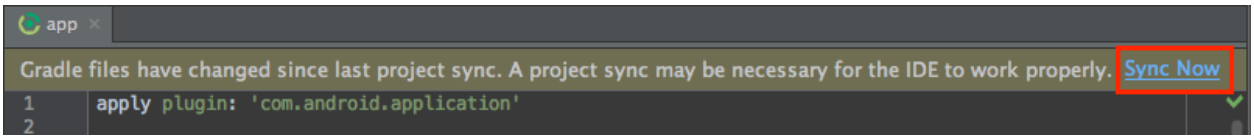


Figure 97.3: After editing the app module's `build.gradle` file, click *Sync Now* to incorporate the changes in your app.

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">71</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 `71` tells Screens that it's communicating with a Liferay CE Portal 7.1 or Liferay DXP 7.1 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

97.2 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`.

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 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.

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

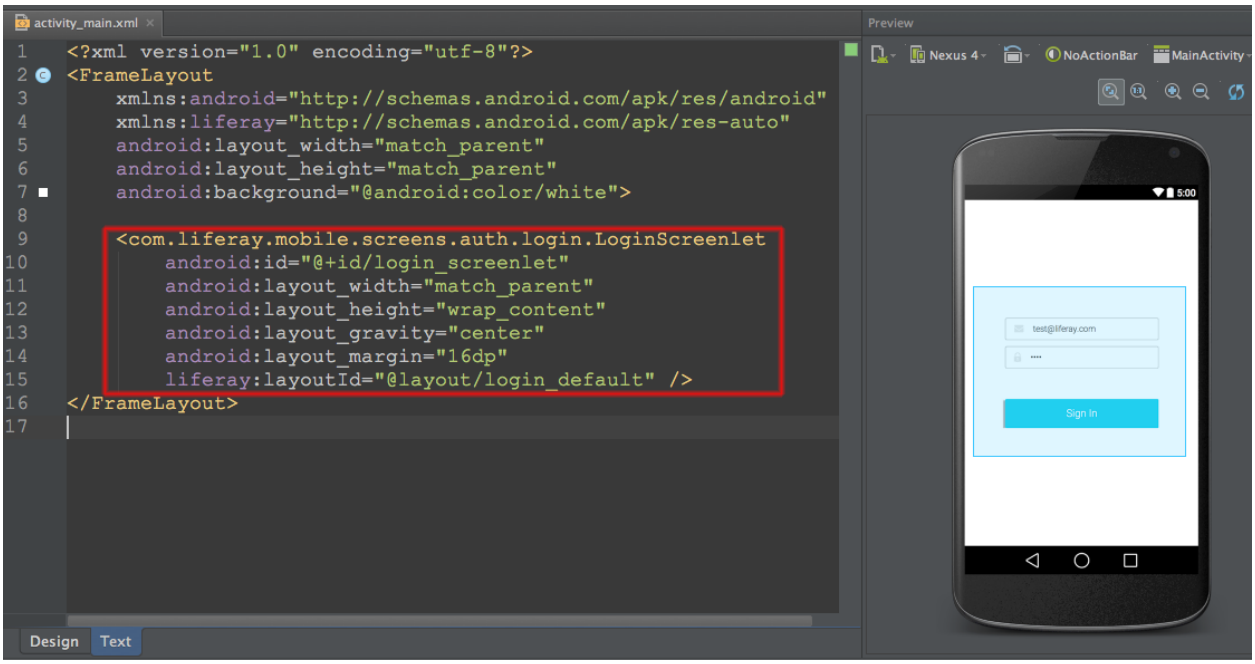


Figure 97.4: Here's the Login Screenlet in an activity's layout in Android Studio.

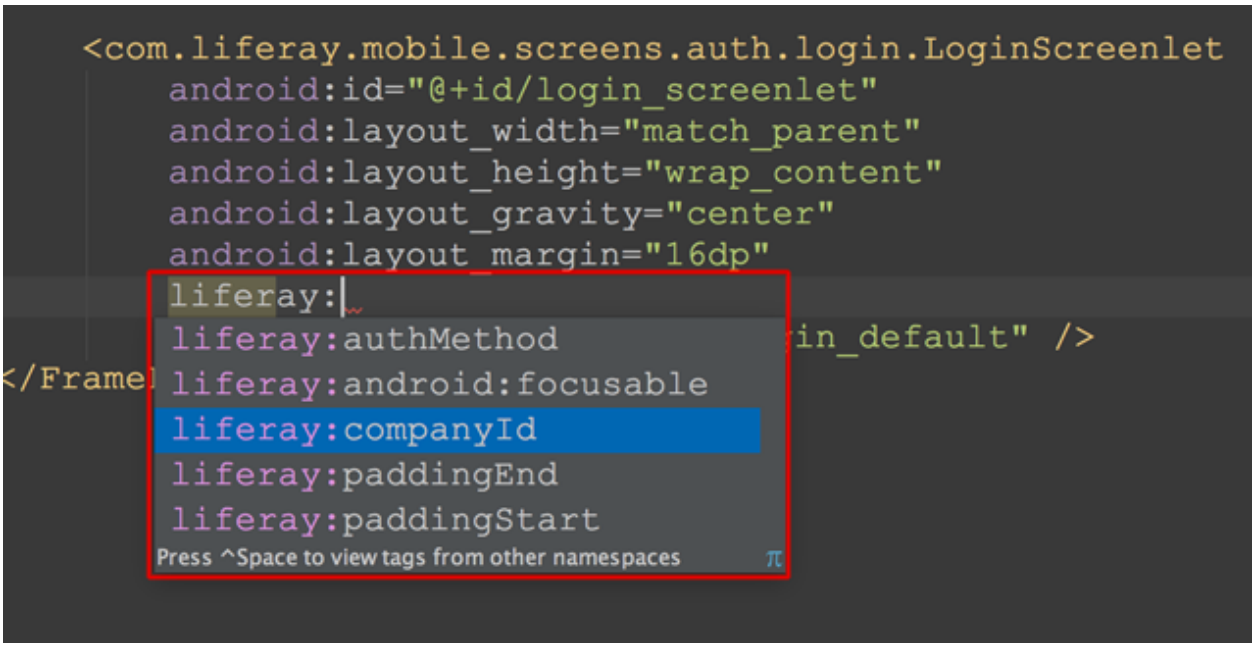


Figure 97.5: You can set a Screenlet's attributes via the app's layout XML file.

```
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 97.6: Implement the Screenlet's listener in your activity or fragment class.

97.3 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 (e.g., Material, Lexicon, 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 here specifies the Material, Lexicon, and Westeros View Sets as dependencies:

```
dependencies {
    ...
    implementation 'com.liferay.mobile:liferay-material-viewset:+'
    implementation 'com.liferay.mobile:liferay-lexicon-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.

Lexicon: Demonstrates Views built from scratch. It follows Liferay's Lexicon Design guidelines.

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
- Lexicon
- 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, Lexicon, or Westeros View Set, inherit `default_theme`, `lexicon_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

97.4 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

CHAPTER 98

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!

98.1 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.

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.

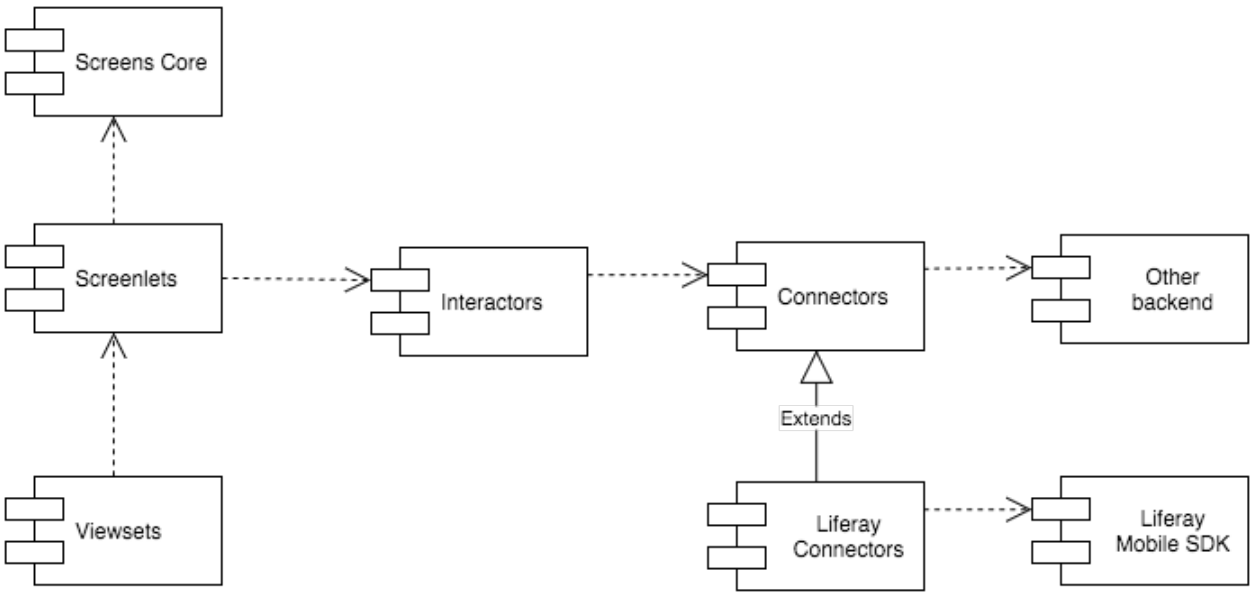


Figure 98.1: Here are the high-level components of Liferay Screens for Android. The dashed arrow connectors represent a "uses" relationship, in which a component uses the component its pointing to.

Views: a set of layouts and accompanying custom view classes that present Screenlets to the user.

Next, the core layer is described in detail.

Related Topics

Core Layer
- Screenlet Layer
- View Layer
- Screenlet Lifecycle

98.2 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.

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:

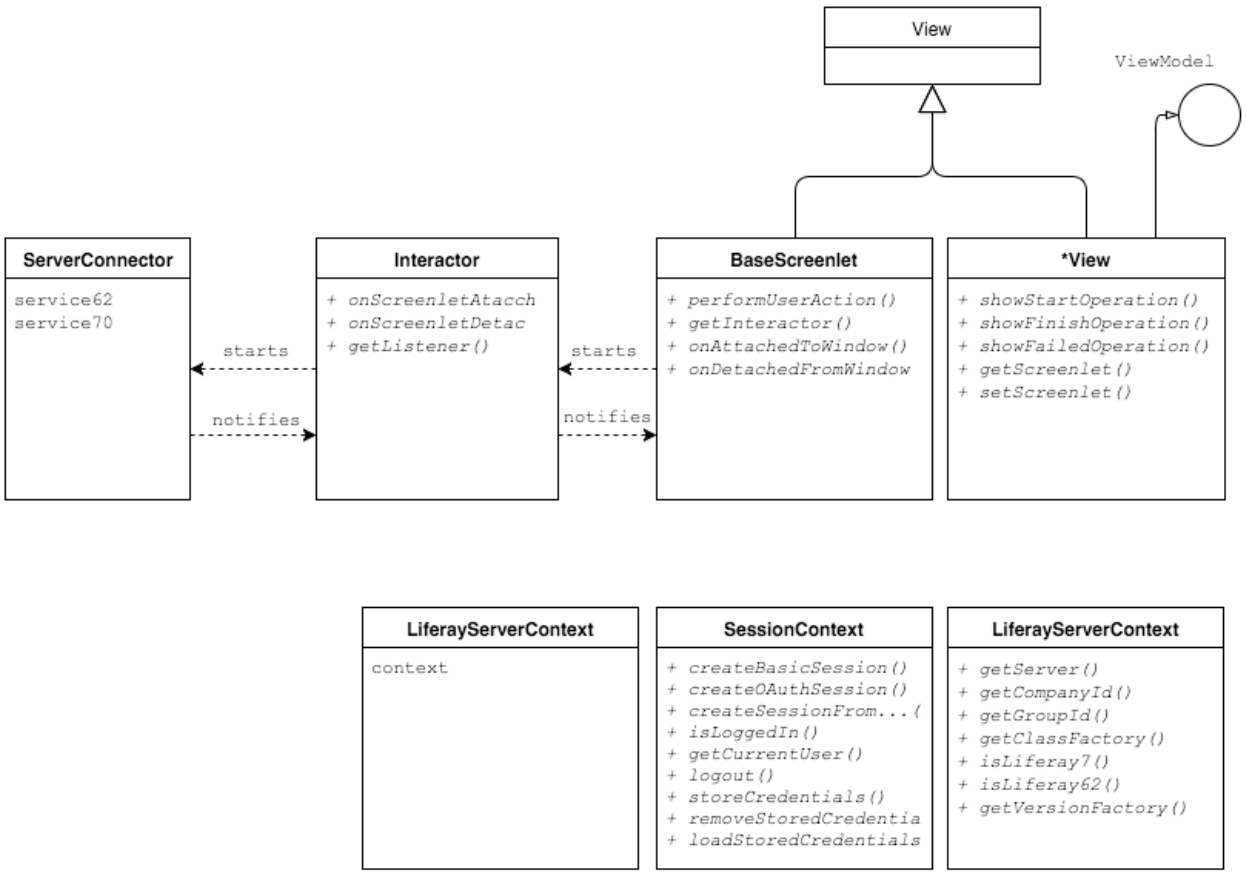


Figure 98.2: Here's the core layer of Liferay Screens for Android.

- *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.

Related Topics

High-Level Architecture
Screenlet Layer
View Layer
Screenlet Lifecycle

98.3 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.

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

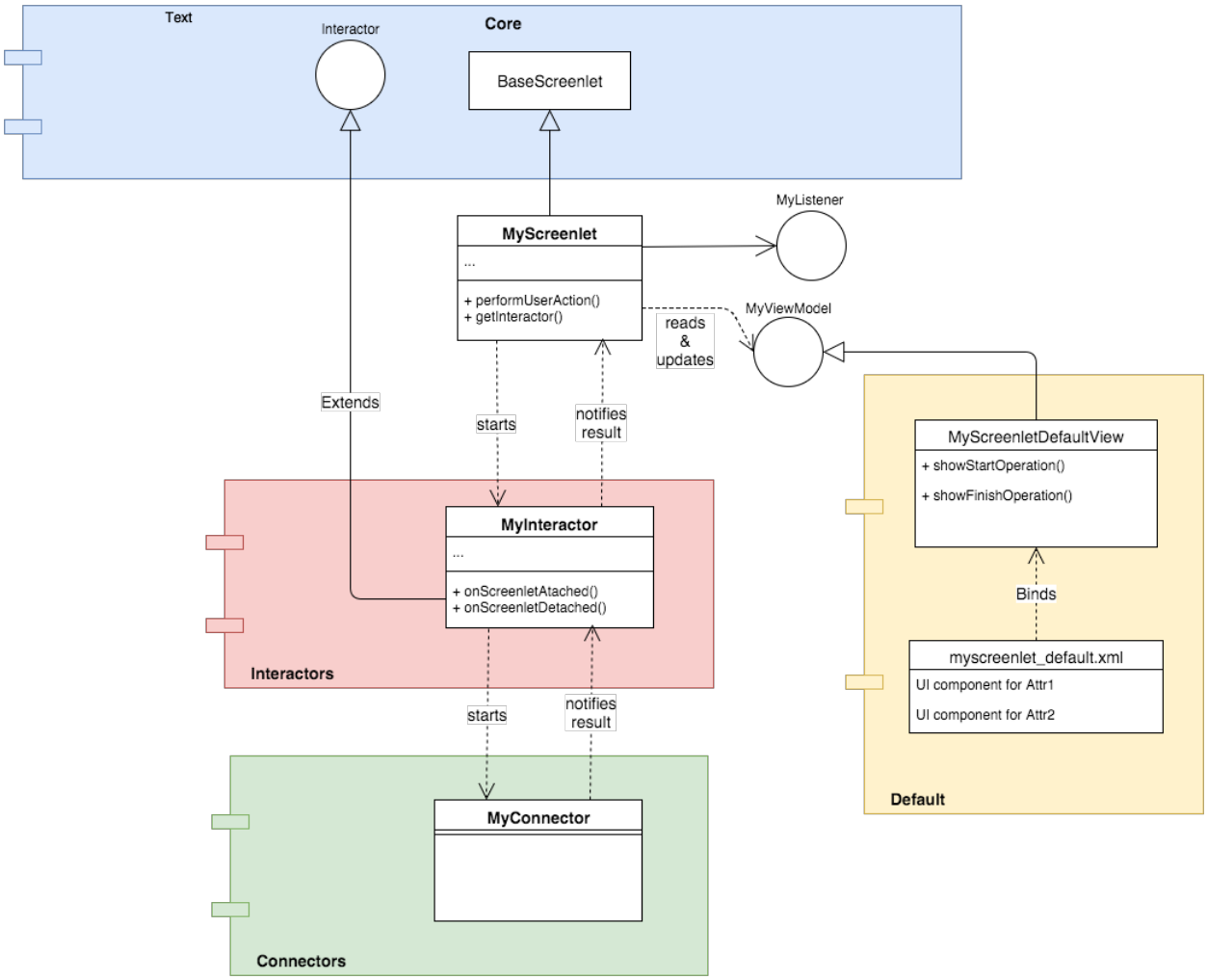


Figure 98.3: This diagram illustrates the Android Screenlet layer's relationship to other Screens components.

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.

Related Topics

High-Level Architecture
    Core Layer
    View Layer
    Screenlet Lifecycle

98.4 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.

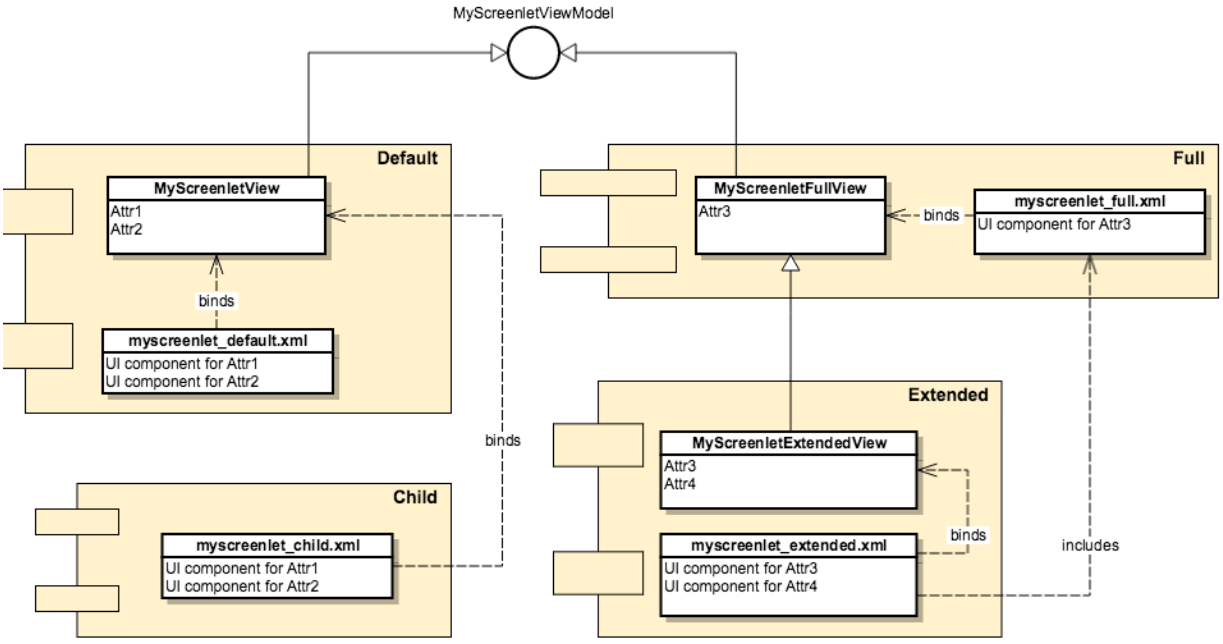


Figure 98.4: This diagram illustrates the View layer of Liferay Screens for Android.

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.

Related Topics

High-Level Architecture

Core Layer
Screenlet Layer
Screenlet Lifecycle

98.5 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.

Related Topics

High-Level Architecture
Core Layer
Screenlet Layer
View Layer

98.6 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:

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

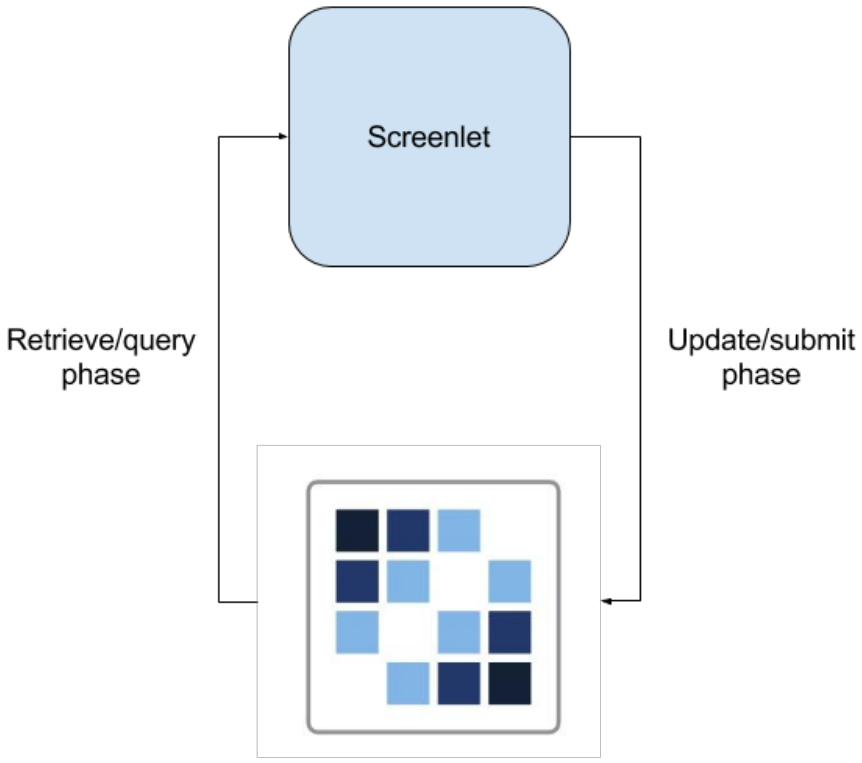


Figure 98.5: A Screenlet's basic phases when requesting and submitting data to the portal.

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:

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

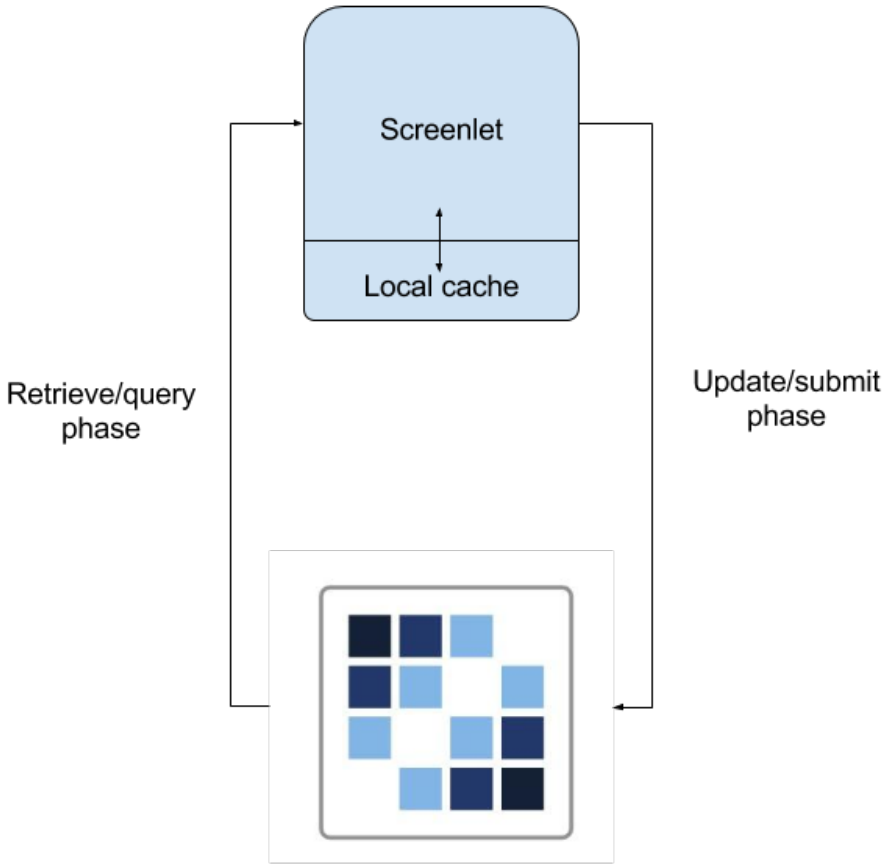


Figure 98.6: This is the same diagram as before, with the addition of the local cache for offline mode.

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.

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:

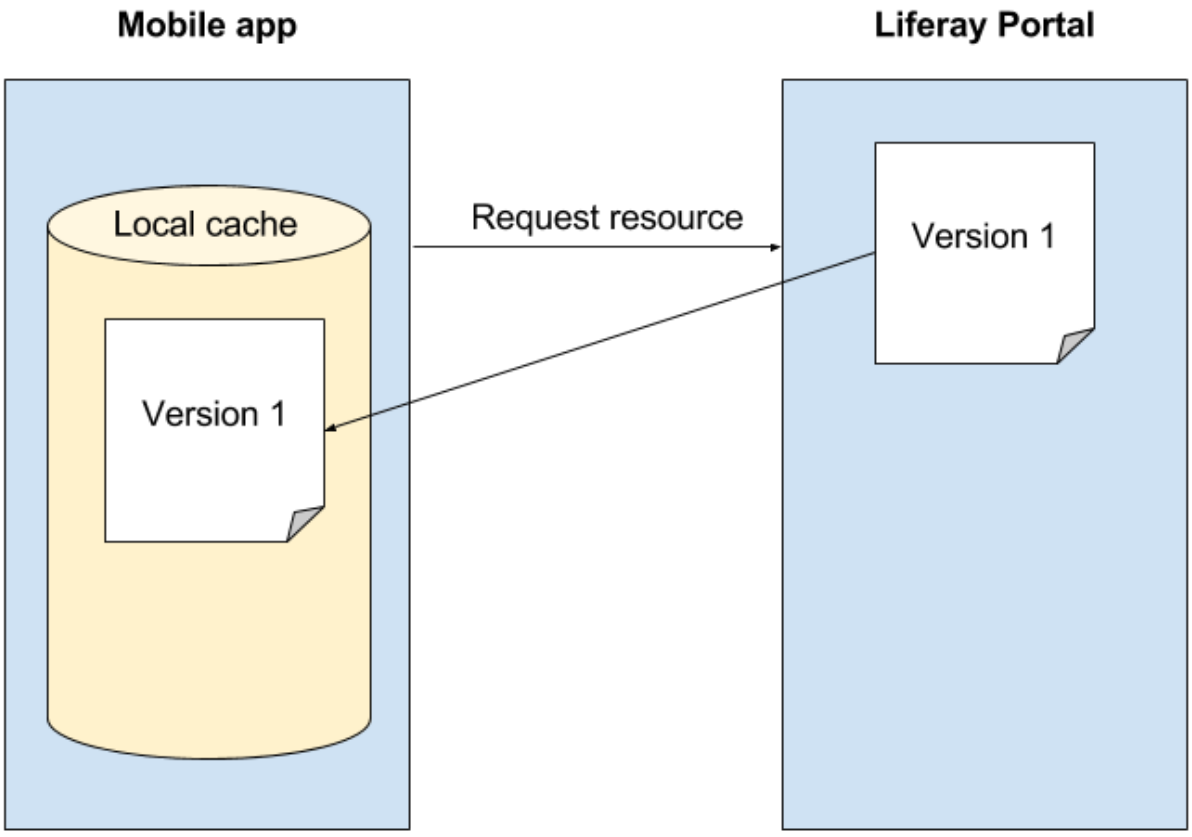


Figure 98.7: The Screenlet requests the resource from the portal and stores it in the app's local cache.

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

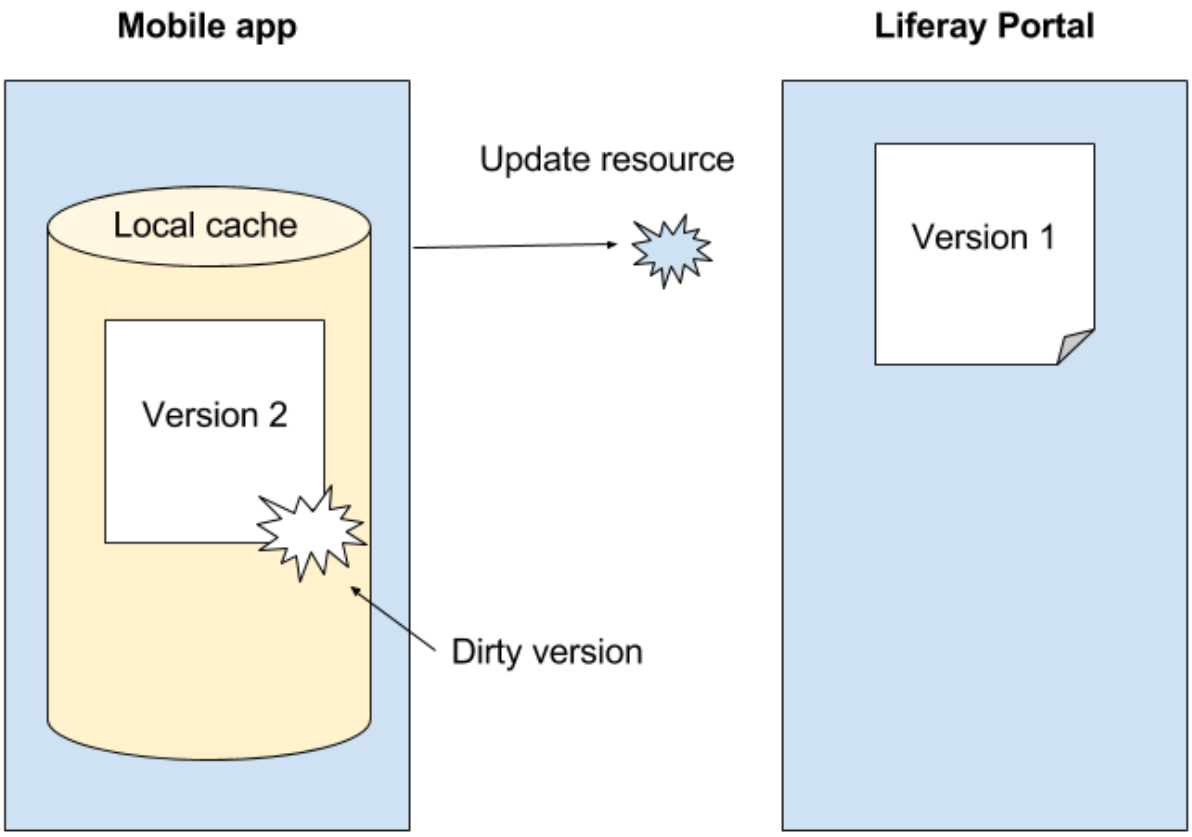


Figure 98.8: The updated data is said to be dirty when the Screenlet can't send it to the portal.

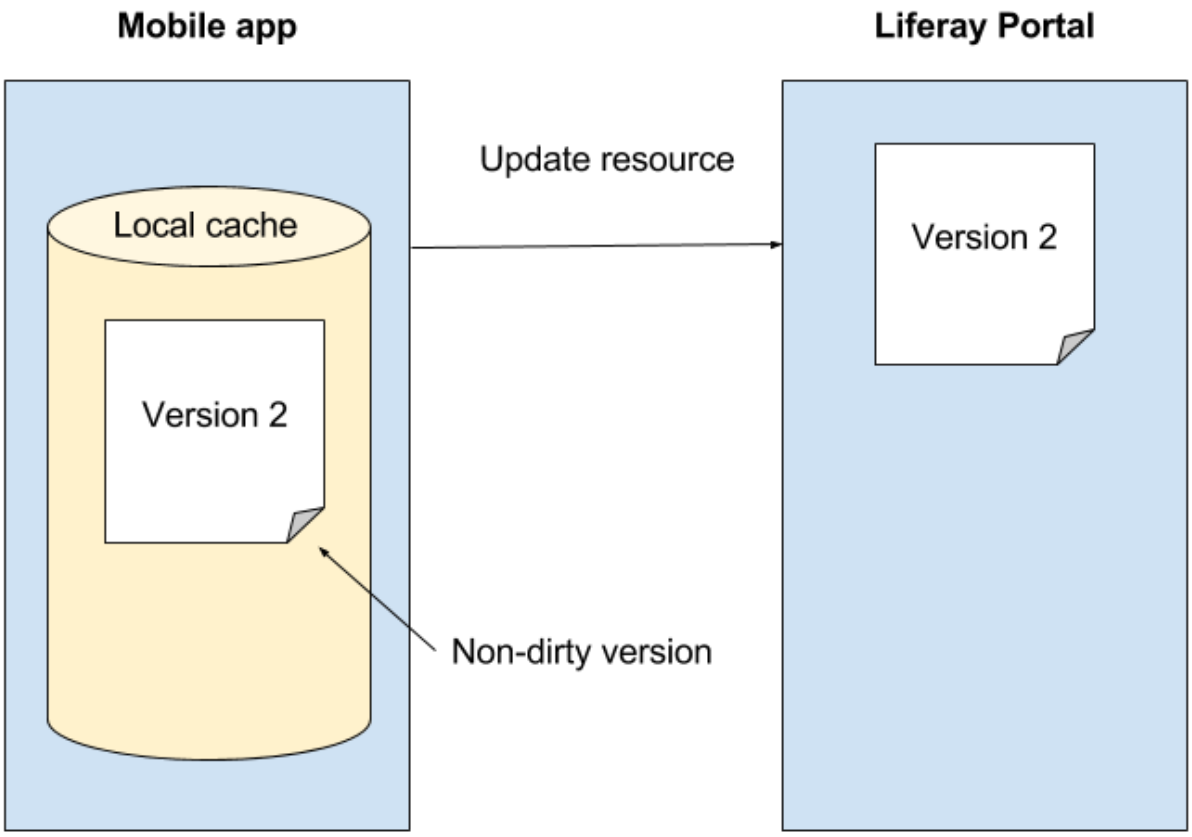


Figure 98.9: The dirty flag is removed once synchronization completes.

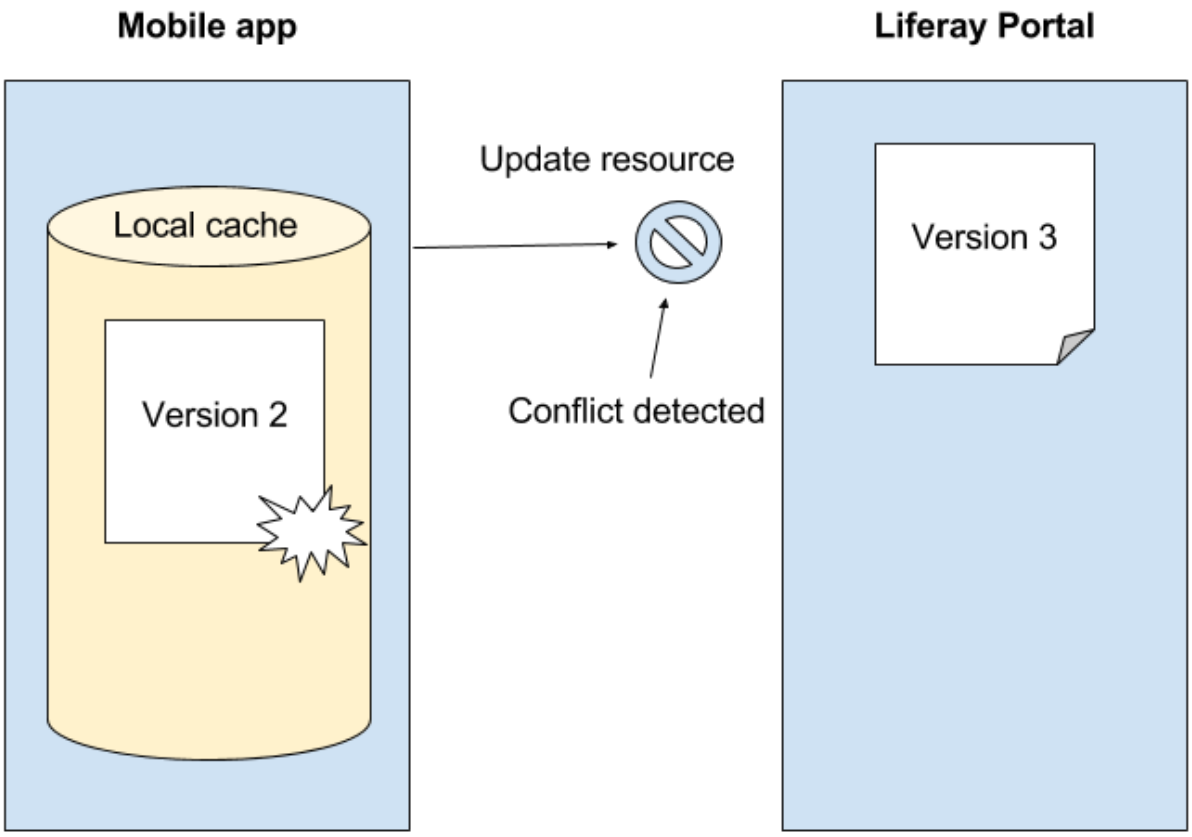


Figure 98.10: Users have changed the data independently in the app and in the portal, causing a synchronization conflict.

CHAPTER 99

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.

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!

99.1 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.

99.2 Creating the 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.

Creating the Screenlet's View Model and Layout

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);
}
```

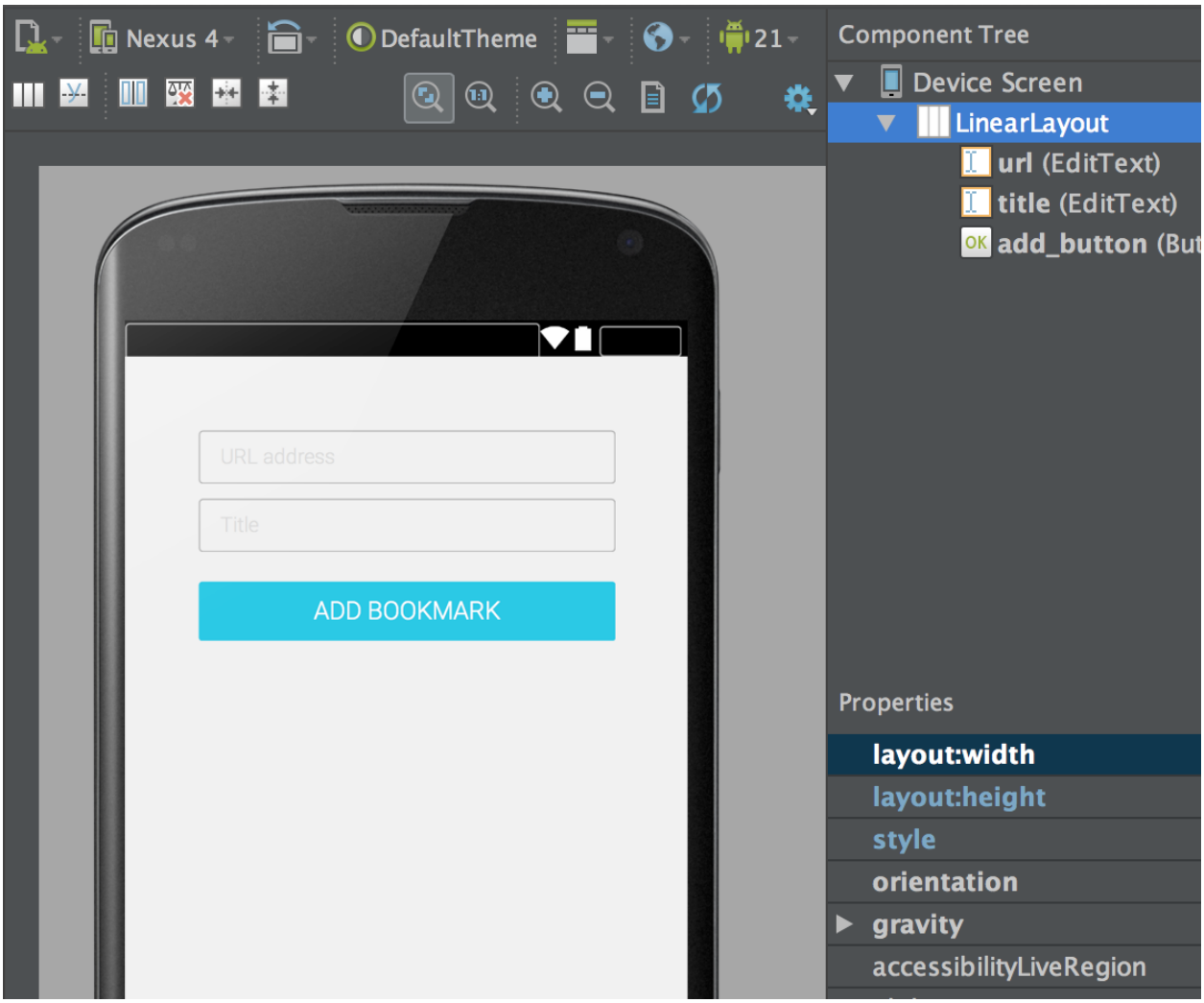


Figure 99.1: Add Bookmark Screenlet's layout contains two text fields and a button.

```
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.

Related Topics

Creating the Interactor
- Defining the Attributes
- Creating the Screenlet Class
- Packaging Your Screenlets

99.3 Creating the 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.

Related Topics

Creating the UI
- Defining the Attributes
- Creating the Screenlet Class
- Packaging Your Screenlets

99.4 Defining the 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 `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.

Related Topics

Creating the UI
    Creating the Interactor
    Creating the Screenlet Class
    Packaging Your Screenlets

99.5 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.

Related Topics

Creating the UI
    Creating the Interactor
    Defining the Attributes
    Packaging Your Screenlets

99.6 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`.

Related Topics

Using Screenlets in Android Apps

99.7 Packaging Your 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

CHAPTER 100

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.

100.1 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!

100.2 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.

Related Topics

Creating the View
    Creating the Interactor
    Creating the Screenlet Class

100.3 Creating the 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.

Related Topics

Creating the Model Class
    Creating the Interactor
    Creating the Screenlet Class

100.4 Creating the 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.

Related Topics

Creating the Model Class
    Creating the View
    Creating the Screenlet Class

100.5 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.

Related Topics

Creating the Model Class
    Creating the View
    Creating the Interactor

100.6 Using the List 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

Using Screenlets in Android Apps

CHAPTER 101

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!

101.1 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.

101.2 Themed Views

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`
- **Lexicon View Set:** `lexicon_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"
    ...
>
```

Related Topics

Child Views
    Extended Views
    Full Views
    Packaging Your Views

101.3 Child Views

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.

Related Topics

Themed Views
    Extended Views
    Full Views
    Packaging Your Views

101.4 Extended Views

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`.

Related Topics

Themed Views
- Child Views
- Full Views
- Packaging Your Views

101.5 Full Views

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`.

Related Topics

Themed Views
    Child Views
    Extended Views
    Packaging Your Views

101.6 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.

Related Topics

Creating Android Views

CHAPTER 102

SUPPORTING OFFLINE MODE

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:

1. Create an event class (or update it if your Screenlet already has one).
2. Update your Screenlet's classes to leverage the offline mode cache.

How you implement these steps depends on how your Screenlet communicates with the server:

- **Write Screenlets:** Write 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.
- **Read Screenlets:** Read 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.

Offline mode implementation differs only slightly between write and read Screenlets. The tutorials in this section use a write Screenlet (Add Bookmark Screenlet) to show you how to support offline mode, and point out any differences needed for a read Screenlet.

Before getting started, be sure to read the basic Screenlet creation tutorial to familiarize yourself with Add Bookmark Screenlet's code.

102.1 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.

Related Topics

Update the Listener
    Update the Interactor Class
    Update the Screenlet Class
    Sync the Cache with the Server

102.2 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.

Related Topics

Create or Update the Event Class
Update the Interactor Class
Update the Screenlet Class
Sync the Cache with the Server

102.3 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> {...
```

If your Screenlet is a write Screenlet, you must change the Interactor's `execute` method to take the event instead of var args (in read Screenlets, this method can still take 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;
}
```

If your Screenlet is a read Screenlet, then you must also implement the `getIdFromArgs` method of `BaseCacheReadInteractor`. 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);
 }
```

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");
}
```

Related Topics

Create or Update the Event Class
    Update the Listener
    Update the Screenlet Class
    Sync the Cache with the Server

102.4 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.

Related Topics

Create or Update the Event Class
Update the Listener
Update the Interactor Class
Sync the Cache with the Server

102.5 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.

Note: You don't have to do this when adding offline mode support to read Screenlets.

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<>());
```

Related Topics

Create or Update the Event Class
    Update the Listener
    Update the Interactor Class
    Update the Screenlet Class

102.6 Supporting Offline Mode in 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

Related Topics

Supporting Offline Mode
    Creating Android List Screenlets

102.7 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.

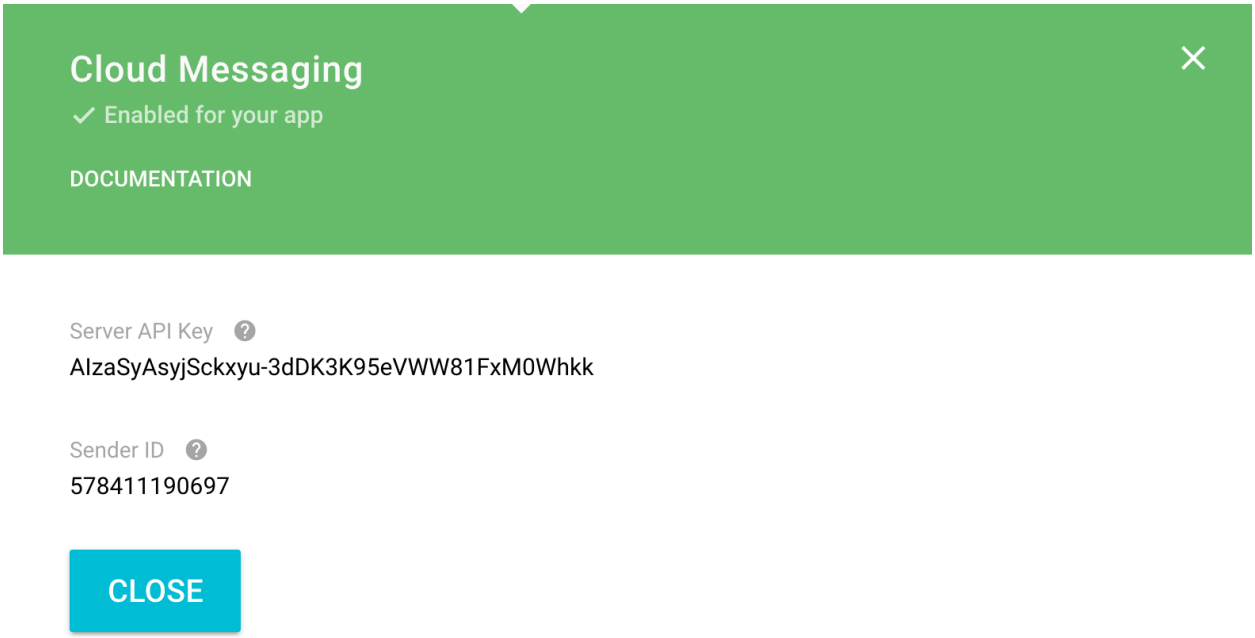


Figure 102.1: You need the Server API Key and Sender ID to enable Liferay Push.

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.

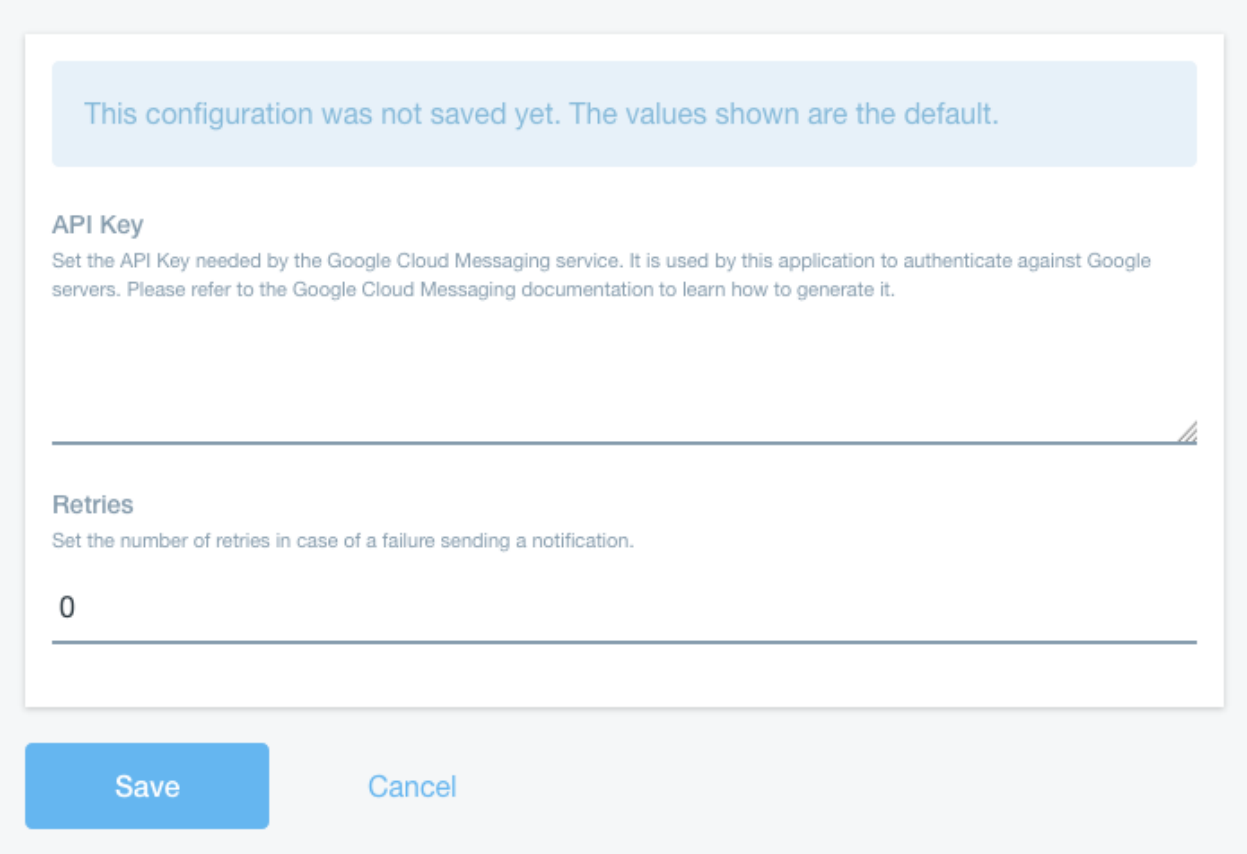


Figure 102.2: Set the API key and number of retries in your Liferay DXP instance.

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

102.8 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

102.9 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

102.10 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

102.11 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.WebListen`

   ```
   ...
   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

102.12 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

102.13 Using OAuth 2 in Liferay Screens for Android

You can use OAuth 2 to authenticate using Login Screenlet with the following OAuth 2 grant types:

- **Authorization Code (PKCE for native apps):** Redirects users to a page in their mobile browser where they enter their credentials. Following login, the browser redirects users back to the mobile app. User credentials can't be compromised via the app because it never accesses them—it uses a revocable token. This is also useful if users don't want to enter their credentials in the app. For example, users may not want to enter their Twitter credentials directly in a 3rd-party Twitter app, preferring instead to authenticate via Twitter's official site. Note that the site you redirect to for authentication must have OAuth 2 implemented.
- **Resource Owner Password:** Users authenticate by entering their credentials directly in the app.
- **Client Credentials:** Authenticates without requiring user interaction. This is useful when the app needs to access its own resources, not those of a specific user.

This tutorial shows you how to use these grant types with Login Screenlet. Note that before getting started, you may want to see Liferay DXP's OAuth 2.0 documentation for instructions on registering an OAuth 2.0 application in the portal.

Authorization Code (PKCE)

Follow these steps to use the Authorization Code grant type with Login Screenlet:

1. Configure the URL where the mobile browser redirects after the user authenticates. To do this, follow the first two steps in the Mobile SDK's Authorization Code instructions. Note that you must configure this URL in both the portal and your Android app.
2. Set Login Screenlet's `loginMode` attribute to `oauth2Redirect`. There are two ways to do this:

    - In code, as the Login Screenlet instance's `authenticationType` variable. You must set this variable via Login Screenlet's `setAuthenticationType` method, using the `AuthenticationType` enum constant `OAUTH2REDIRECT`:

```
loginScreenlet.setAuthenticationType(AuthenticationType.OAUTH2REDIRECT);
```

- When inserting Login Screenlet's XML, set the `loginMode` attribute to `oauth2Redirect`.

3. In Login Screenlet's XML, set Login Screenlet's `oauth2ClientId` attribute to the ID of the portal's OAuth 2 application that you want to use. To find this value, navigate to that application in the portal's OAuth 2 Admin portlet.

4. In Login Screenlet's XML, set Login Screenlet's `oauth2Redirect` attribute to the URL you configured in step 1.

   Here's an example of Login Screenlet's XML with the attributes from the preceding steps:

```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:loginMode="oauth2Redirect"
    app:oauth2Redirect="my-app://my-app"
    app:oauth2ClientId="54321"
    app:credentialsStorage="shared_preferences"
/>
```

5. In your activity that uses Login Screenlet, you must override the `onActivityResult` method to implement the redirect you configured in the first step. You do this by calling Login Screenlet's `resumeOAuth2RedirectFlow` method:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    super.onActivityResult(requestCode, resultCode, intent);

    if (requestCode == OAuth2SignIn.REDIRECT_REQUEST_CODE) {
        loginScreenlet.resumeOAuth2RedirectFlow(intent);
    }
}
```

Resource Owner Password

Follow these steps to use the Resource Owner Password grant type with Login Screenlet:

1. Set Login Screenlet's `loginMode` attribute to `oauth2UsernameAndPassword`. There are two ways to do this:

   - In code, as the Login Screenlet instance's `authenticationType` variable. You must set this variable via Login Screenlet's `setAuthenticationType` method, using the `AuthenticationType` enum constant `OAUTH2USERNAMEANDPASSWORD`:

```
loginScreenlet.setAuthenticationType(AuthenticationType.OAUTH2USERNAMEANDPASSWORD);
```

   - When inserting Login Screenlet's XML, set the `loginMode` attribute to `oauth2UsernameAndPassword`.

2. In Login Screenlet's XML, set Login Screenlet's `oauth2ClientId` attribute to the ID of the OAuth 2 application that you want to use. To find this value, navigate to that application in the OAuth 2 Admin portlet.

3. In Login Screenlet's XML, set Login Screenlet's `oauth2ClientSecret` attribute to the same OAuth 2 application's client secret.

   Here's an example of Login Screenlet's XML with the attributes from the preceding steps:

```
<com.liferay.mobile.screens.auth.login.LoginScreenlet
    android:id="@+id/login_screenlet"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:loginMode="oauth2UsernameAndPassword"
    app:oauth2ClientId="54321"
    app:oauth2ClientSecret="12345"
    app:basicAuthMethod="email"
    app:credentialsStorage="shared_preferences"
/>
```

Client Credentials

The OAuth 2 Client Credentials grant type authenticates without requiring user interaction. This is useful when the app needs to access its own resources, not those of a specific user.

Warning: The Client Credentials grant type poses a security risk to the portal. To authenticate without user credentials, the mobile app must contain the OAuth 2 application's client ID and client secret. Anyone who can access those values via the mobile app can also authenticate without user credentials.

Follow these steps to use the Client Credentials grant type in your Android app:

1. Follow the Android Mobile SDK instructions for using the Client Credentials grant type.

2. The session object contains a valid authentication object. Pass the session as an argument to the `SessionContext` method `createOAuth2Session`:

   ```
   SessionContext.createOAuth2Session(session);
   ```

   This initializes the Screens `SessionContext` object, authenticating any Screenlets that you use in the Android app.

Related Topics

Using OAuth 2 in the Android Mobile SDK
Using Screenlets in Android Apps
OAuth 2.0

102.14 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.

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

102.15 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

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 are tutorials 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

CHAPTER 103

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 DXP'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. The 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 DXP's web 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 iOS Screenlets and their Themes. There's even a tutorial that explains the Liferay Screens architecture.

To get started, prepare your iOS project to use Liferay Screens.

103.1 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!

Figure 103.1: The Liferay Screens Sign Up Screenlet lets users create an account in the portal.

First, you'll review the requirements for Liferay Screens.

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
- 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.

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.

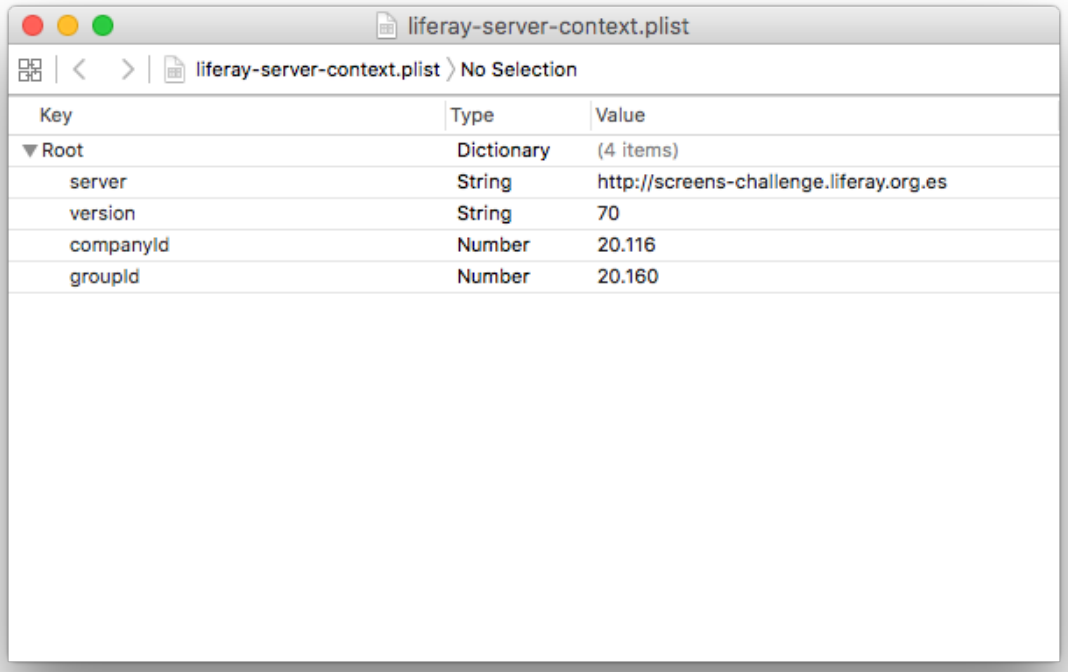


Figure 103.2: Here's a property list file, called `liferay-context.plist`.

Related Topics

Using Screenlets in iOS Apps
    Using Themes in iOS Screenlets
    Preparing Android Projects for Liferay Screens

103.2 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!

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.

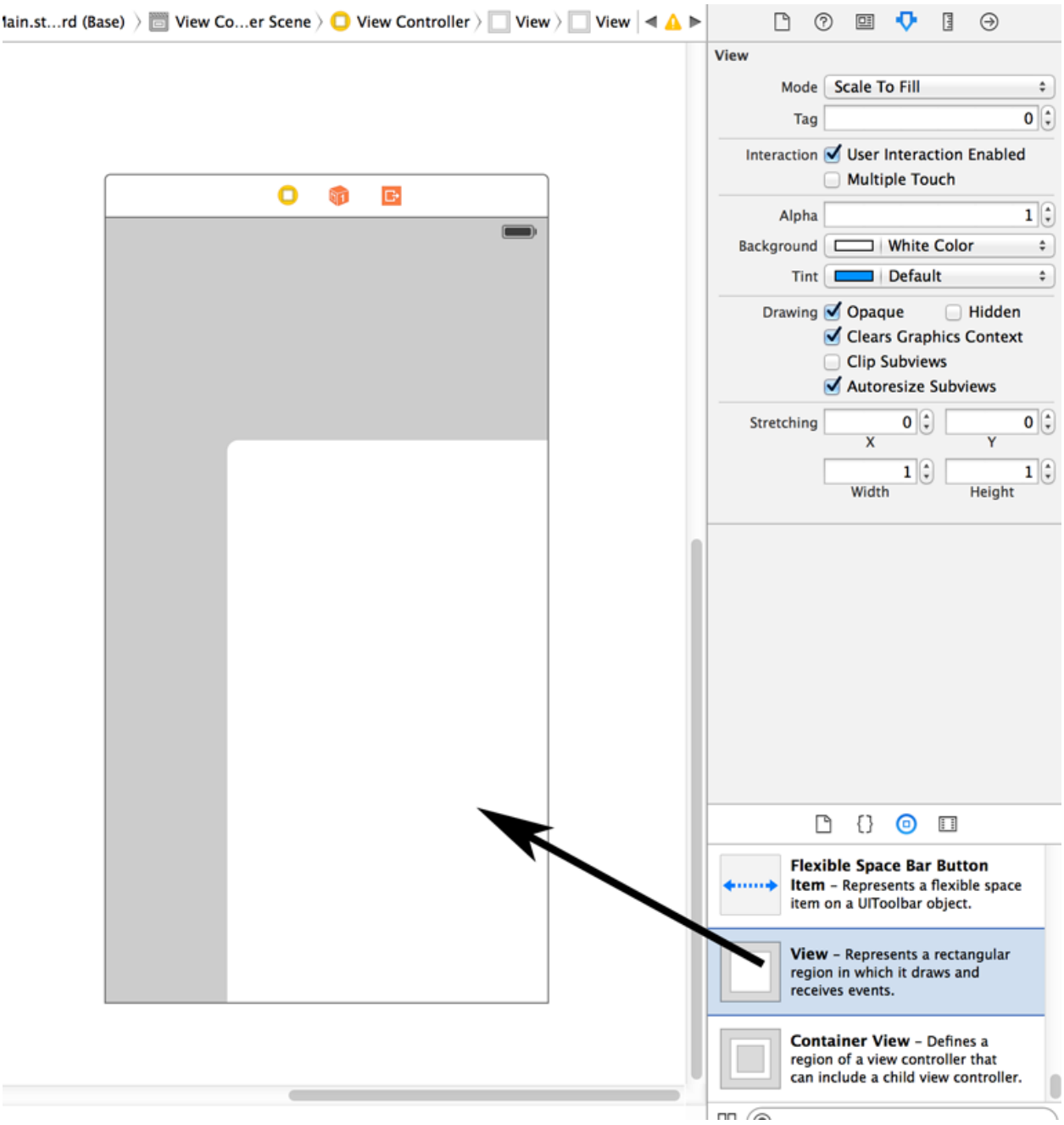


Figure 103.3: Add a new UIView to your project.

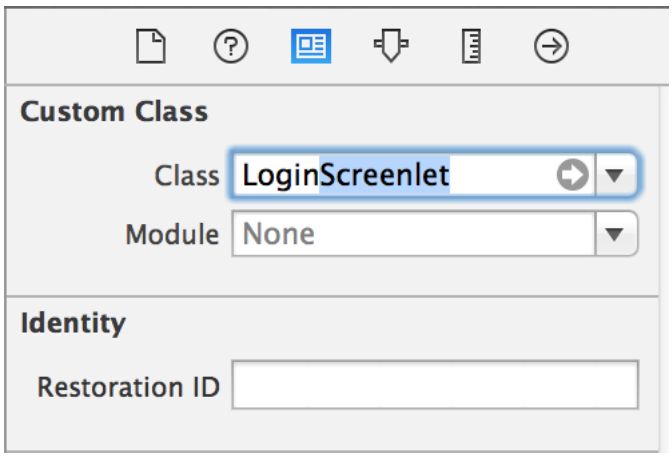


Figure 103.4: Change the Custom Class to match the Screenlet.

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
    }

    ...
```

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.

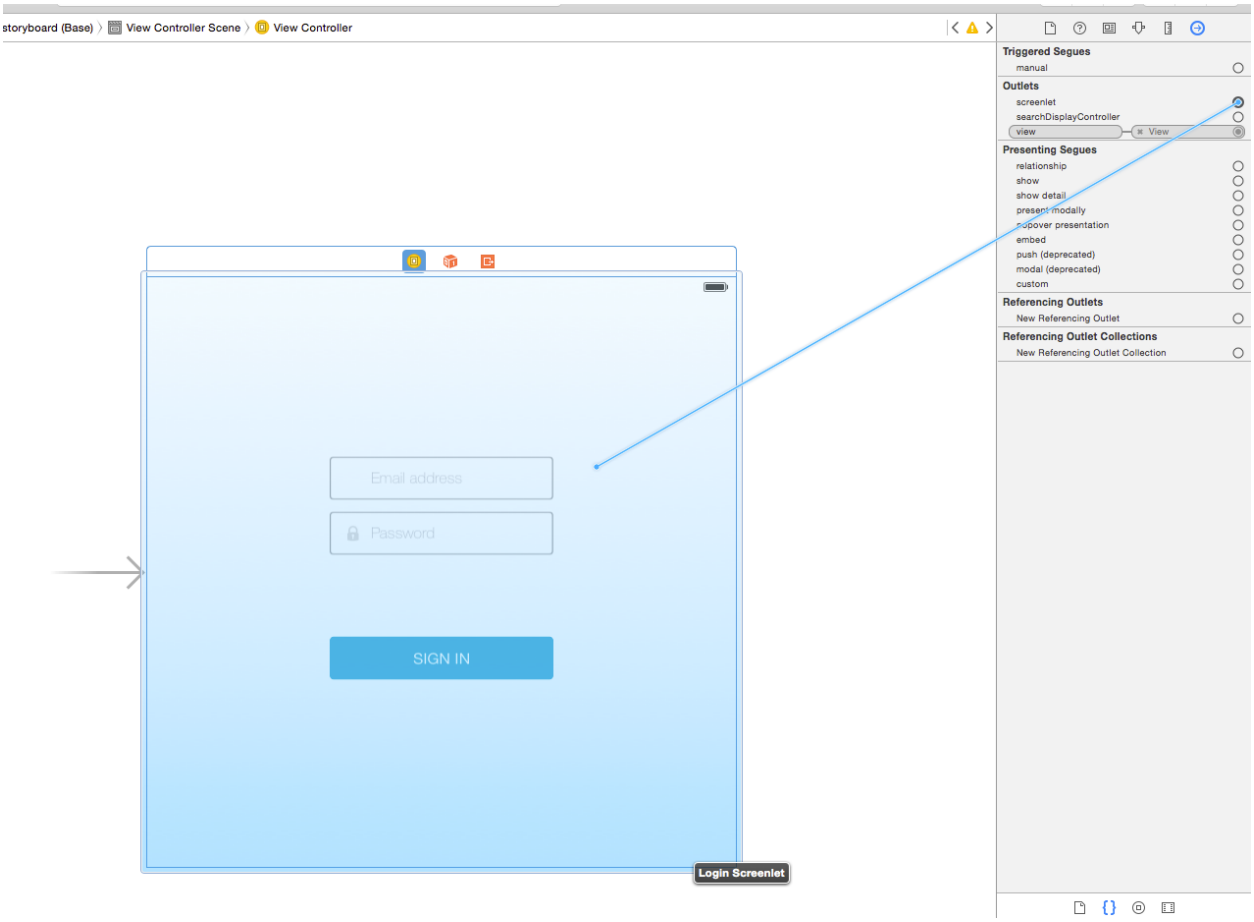


Figure 103.5: Connect the outlet with the Screenlet reference.

```
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.

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"
```

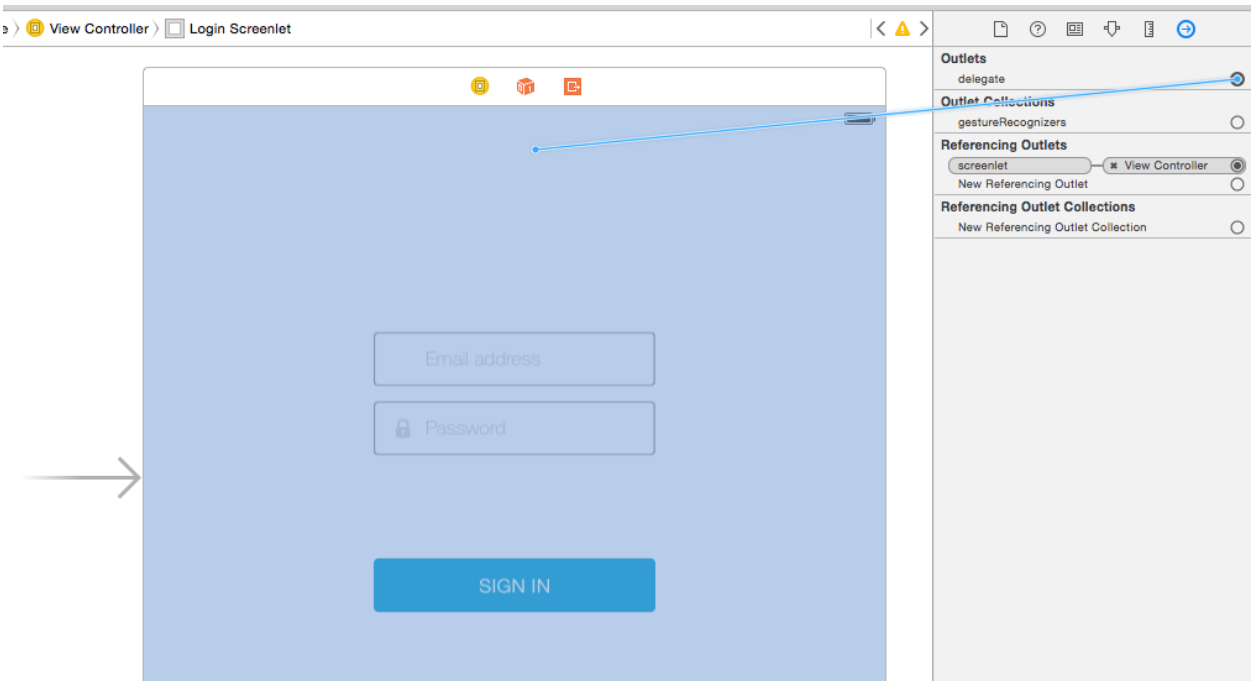


Figure 103.6: Connect the Screenlet's delegate in Interface Builder.

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`

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.

Way to go! You now know how to use Screenlets in your iOS apps.

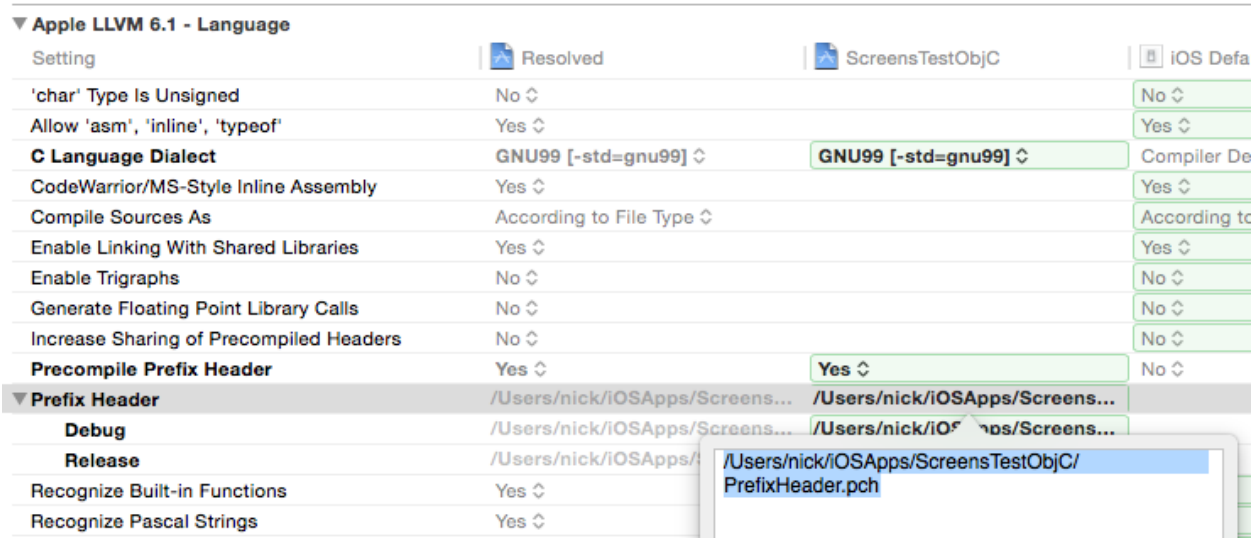


Figure 103.7: The `PrefixHeader.pch` configuration in Xcode settings.

Related Topics

Preparing iOS Projects for Liferay Screens
- Using Themes in iOS Screenlets
- Creating iOS Screenlets
- Using Screenlets in Android apps

103.3 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.

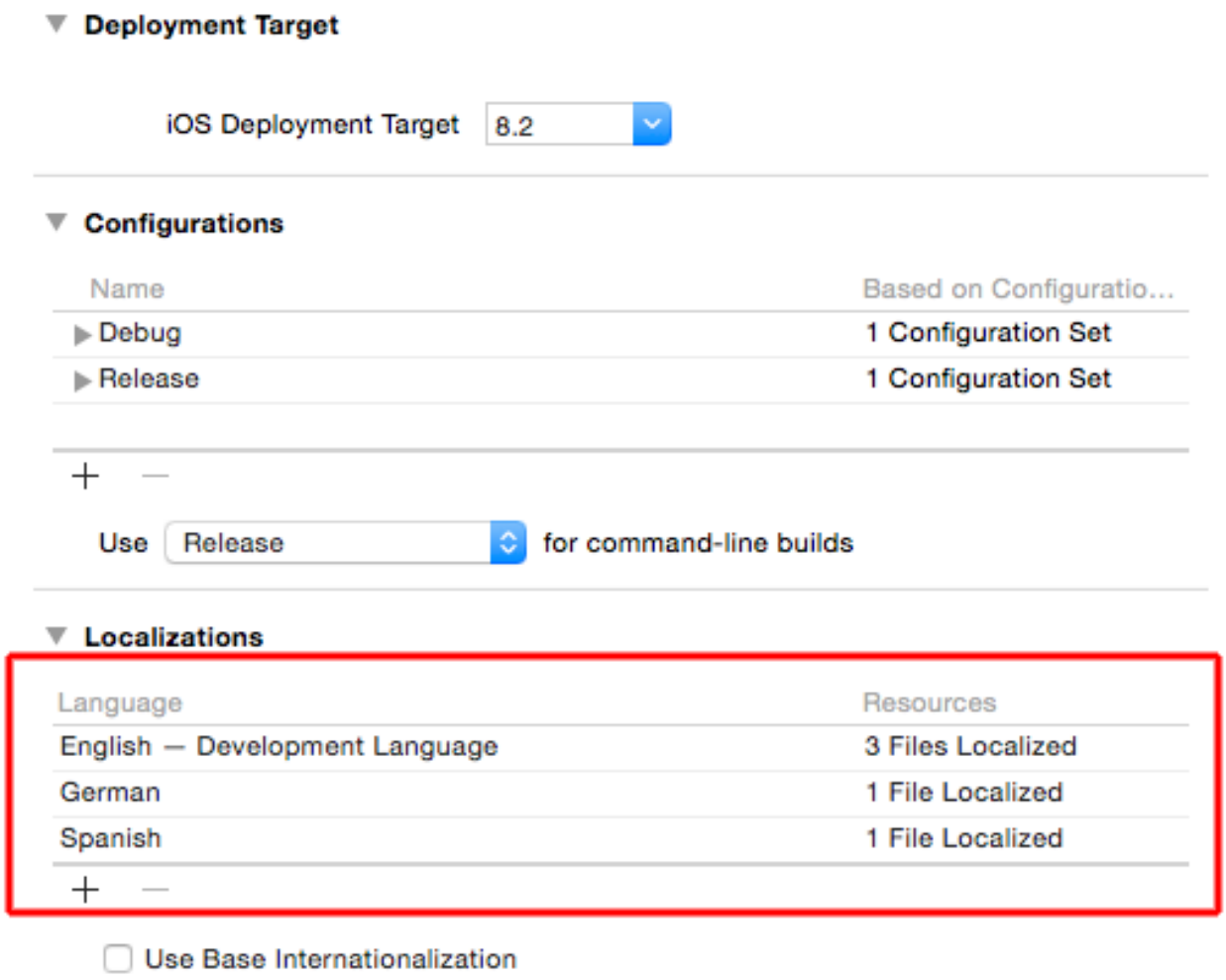


Figure 103.8: The Xcode localizations in the project's settings.

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

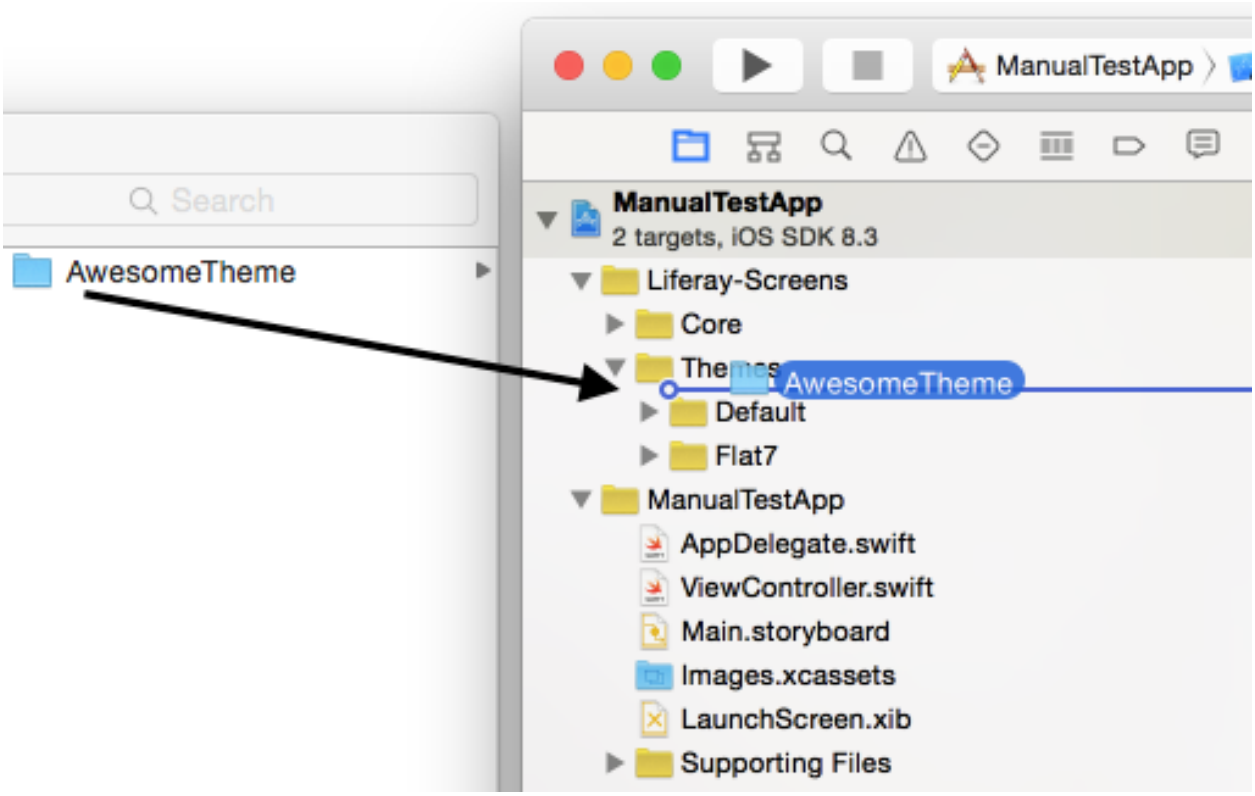


Figure 103.9: To install a Theme into an Xcode project, drag and drop the Theme's folder into it.

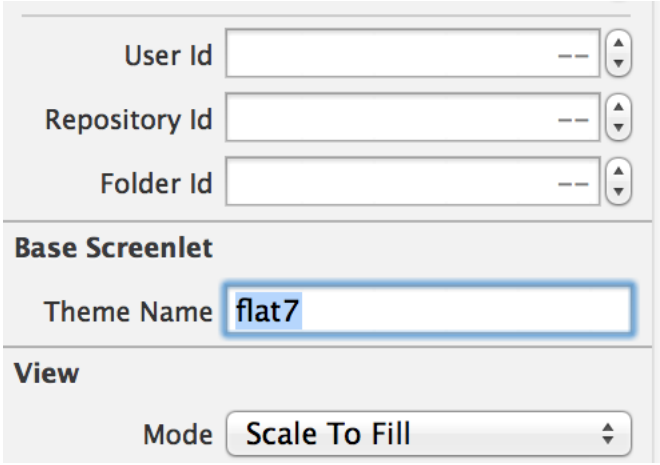


Figure 103.10: In Interface Builder, you specify a Screenlet's Theme by entering its name in the *Theme Name* field; this sets the Screenlet's `themeName` property.

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

103.4 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

CHAPTER 104

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.

The tutorials in this section explain the architecture of Liferay Screens for iOS. First is 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 tutorials that follow. After you get done examining these building blocks, you'll be ready to create some amazing Screenlets and Themes!

104.1 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.

Each component is described here:

- **Core:** A micro-framework that lets developers write their own Screenlets, views, and Server Connector classes. The Core includes all the base classes for developing Screens components.
- **Screenlets:** Swift classes to insert in any `UIView`. Screenlets 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:** Implementations of specific use cases for communicating with servers or any other data store. They 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.

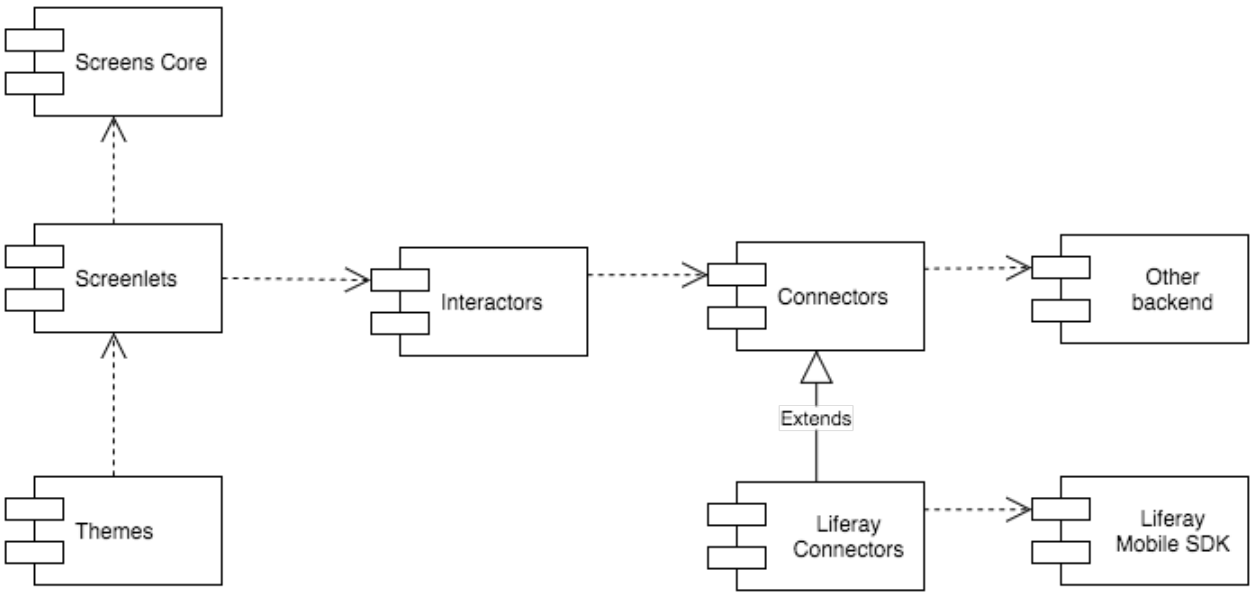


Figure 104.1: The high level components of Liferay Screens for iOS.

- **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 Connectors, use the Liferay Mobile SDK. All Server Connectors can run concurrently since they use the `NSOperation` framework. It's straightforward to define priorities and dependencies between Connectors, so you can build your own graph of Connectors (operations) that can the framework can resolve. 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.

104.2 Core Layer of Liferay Screens for iOS

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.

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

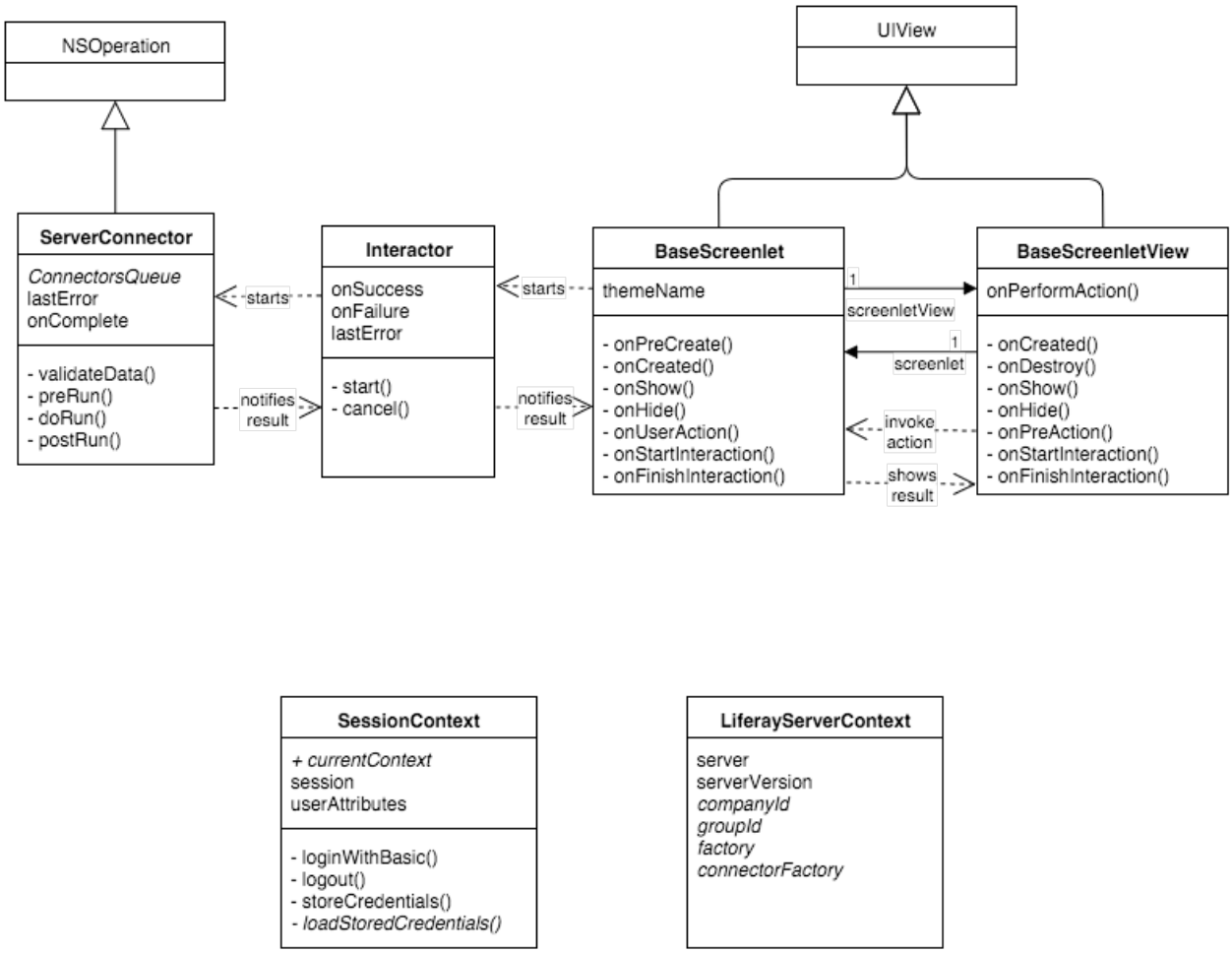


Figure 104.2: Here's the core layer of Liferay Screens for iOS.

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. For more information, see the tutorial on accessing the session in iOS.

- **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.

104.3 Screenlet Layer of Liferay Screens for iOS

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.

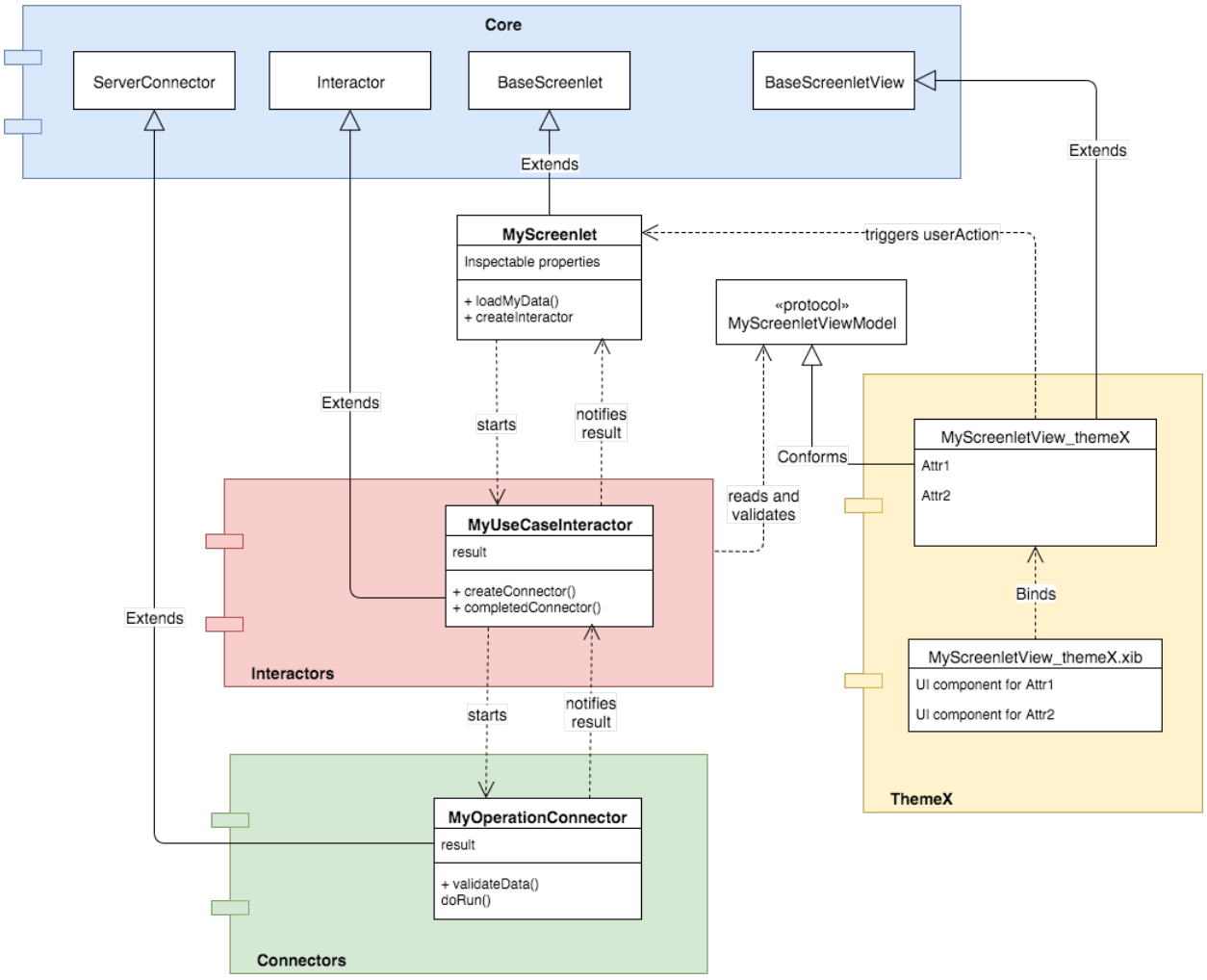


Figure 104.3: This diagram illustrates the iOS Screenlet Layer's relationship to other Screens components.

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 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 tutorials on creating iOS Screenlets.

104.4 Theme Layer of Liferay Screens for iOS

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 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 series Creating iOS Themes.

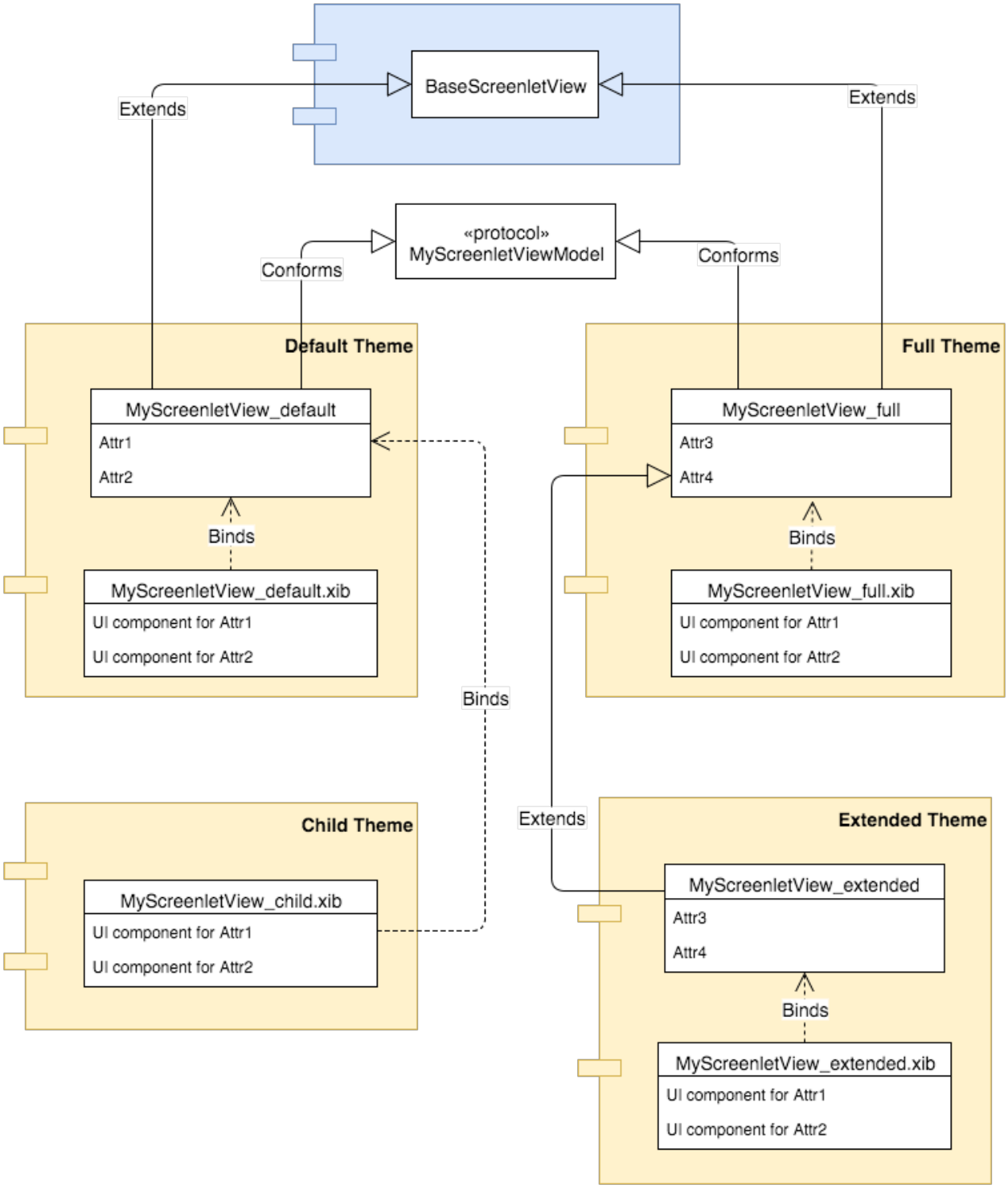


Figure 104.4: The Theme Layer of Liferay Screens for iOS.

CHAPTER 105

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 series 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 portal services you'll call.
2. **Create Your Screenlet's UI (its Theme):** Although these tutorials present 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.
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 using a Screenlet.

The tutorials that follow walk you through these steps. Before getting started, make sure that you're familiar with the architecture of Liferay Screens.

105.1 Planning Your iOS 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`.

105.2 Creating the iOS 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

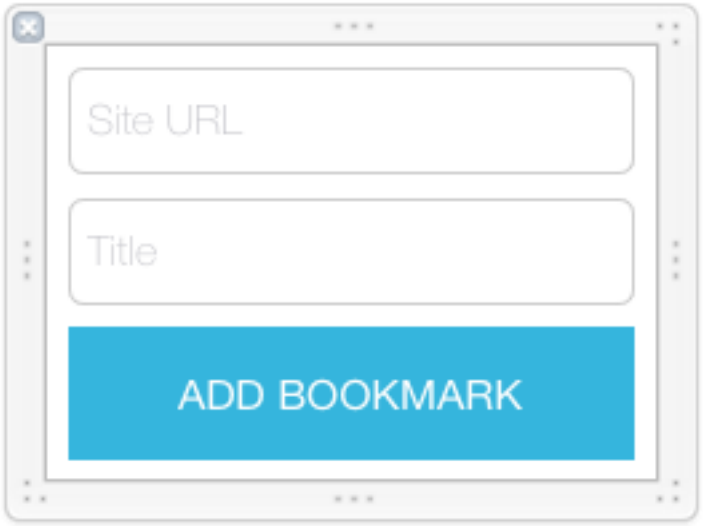


Figure 105.1: Here's the sample Add Bookmark Screenlet's XIB file rendered in Interface Builder.

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"`.

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.

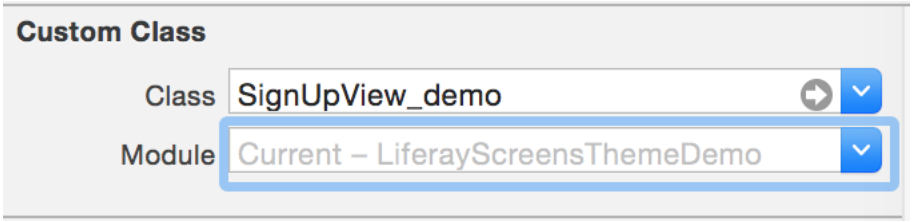


Figure 105.2: In this XIB file, the custom class's module is NOT specified.

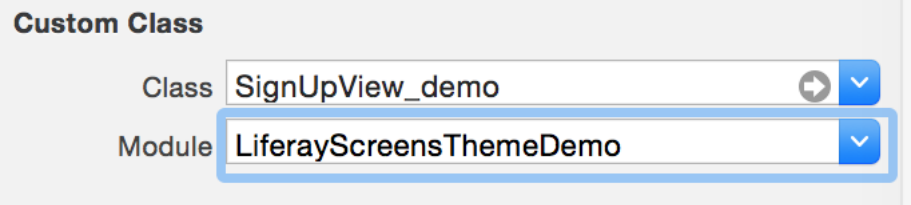


Figure 105.3: The XIB file is bound to the custom class name, with the specified module.

105.3 Creating the iOS Screenlet's 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.

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()
}
```

105.4 Creating the iOS Screenlet's 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.

Chapter 106

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.

The tutorials in this section use 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:

- 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

You'll create a list Screenlet by following these steps:

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.

106.1 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.

106.2 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 list Screenlet's theme.

106.3 Creating the iOS List Screenlet's 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.

106.4 Creating the iOS List Screenlet's 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.

106.5 Creating the iOS List Screenlet's 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.

106.6 Creating the iOS List Screenlet's 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.

106.7 Creating the iOS List Screenlet's 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)
  }
  ```

CHAPTER 107

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.

The tutorials in this section explain how to create all three Theme types. To understand Theme concepts and components, you might want to examine the architecture of Liferay Screens for iOS. The tutorials on creating iOS Screenlets can help you create any Screenlet classes your Theme requires.

107.1 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.

107.2 Creating an iOS 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.

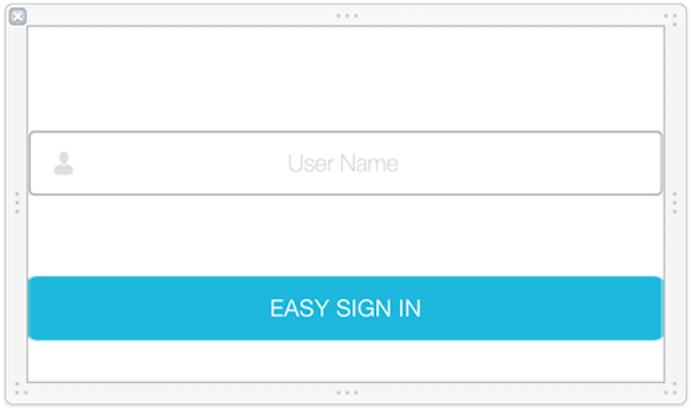


Figure 107.1: The UI components are enlarged in the example Child Theme's XIB file.

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.

```
**Note:** 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.

107.3 Creating an iOS 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.

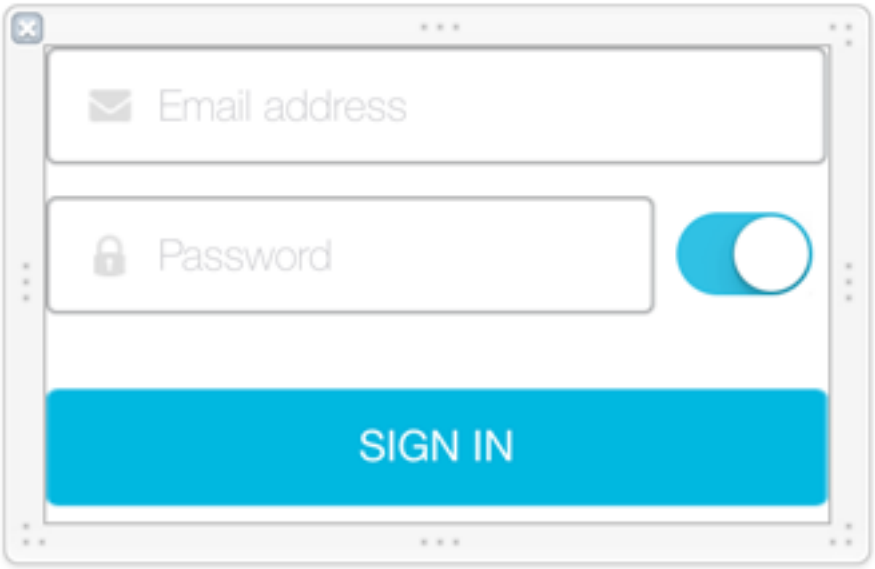


Figure 107.2: This example Extended Theme's XIB file extends the Login Portlet's UI and behavior with a switch that lets the user show or hide the password field value.

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.

107.4 Creating an iOS 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.

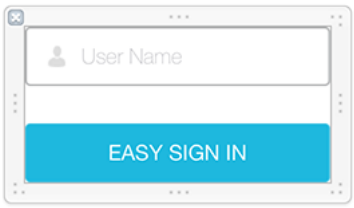


Figure 107.3: This Full Theme for the Login Screenlet, includes a text field for entering the user name, uses the UDID for the password, and adds a *Sign In* button with the same `restorationIdentifier` as the Default Theme.

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.

Now that your theme is finished, you can optionally package and/or start using it. Note that a Full Theme can serve as a parent to a Child and Extended Theme.

107.5 Packaging iOS Themes

Once you've created a Theme, you can package it for installation and use with its Screenlet. Your Theme is a code library that you can package using CocoaPods.

Follow the steps here 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.

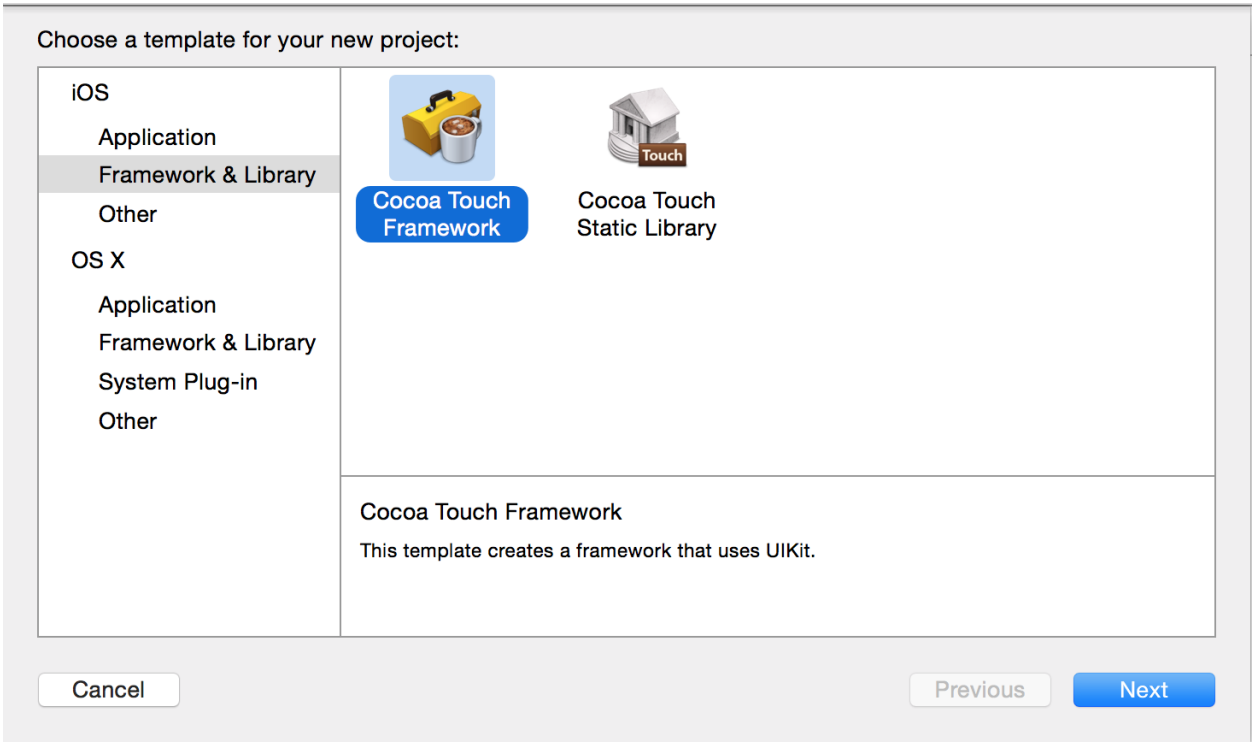


Figure 107.4: Choose *Cocoa Touch Framework* when creating a project for your Theme.

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.

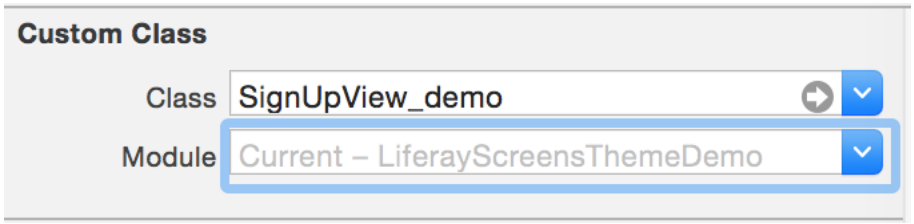


Figure 107.5: This XIB file's custom class's module is NOT specified.

In the following screenshot, the setting for the custom class is correct:

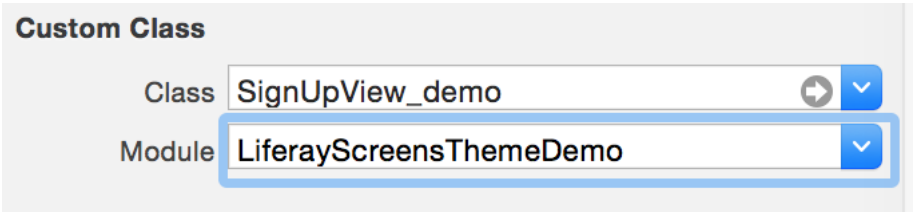


Figure 107.6: The XIB file is bound to the custom class name, with the specified module.

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

107.6 Supporting Multiple Themes in Your iOS 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 tutorials 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 tutorials is used as an example. Note that you can also 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 tutorials on 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

107.7 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 tutorials. 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 tutorials.
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 tutorials. 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:

```
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:

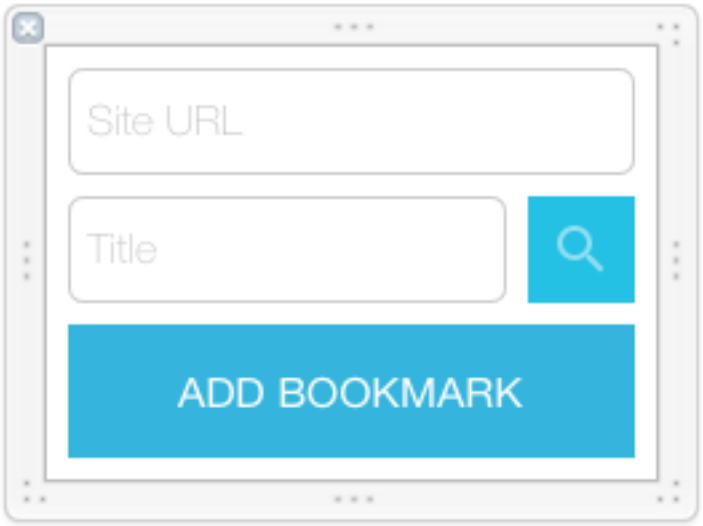


Figure 107.7: The sample Add Bookmark Screenlet's XIB file contains a new button next to the *Title* field for retrieving the URL's title.

```
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. The next section shows you how to trigger your actions programmatically.

Related Topics

Creating iOS Screenlets
- Create and Use a Connector with Your Screenlet
- Creating iOS List Screenlets
- Architecture of Liferay Screens for iOS

107.8 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

107.9 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

107.10 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:

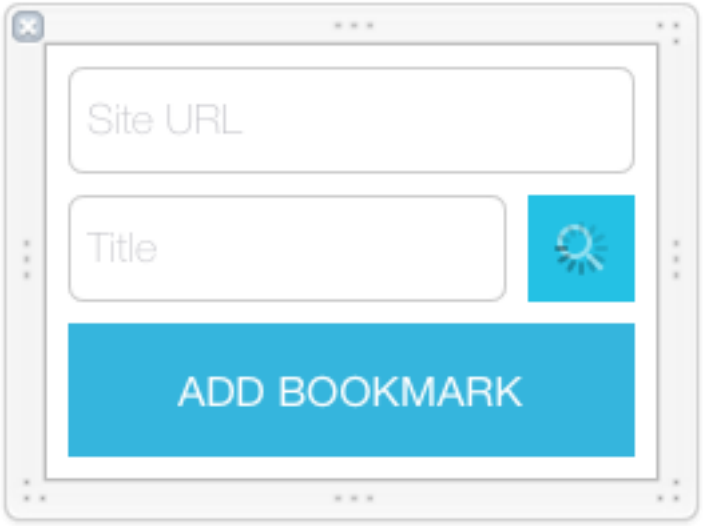


Figure 107.8: The updated Add Bookmark Screenlet's XIB file contains a new activity indicator over the get title button.

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

107.11 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.

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

107.12 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 107.9: 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

107.13 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:

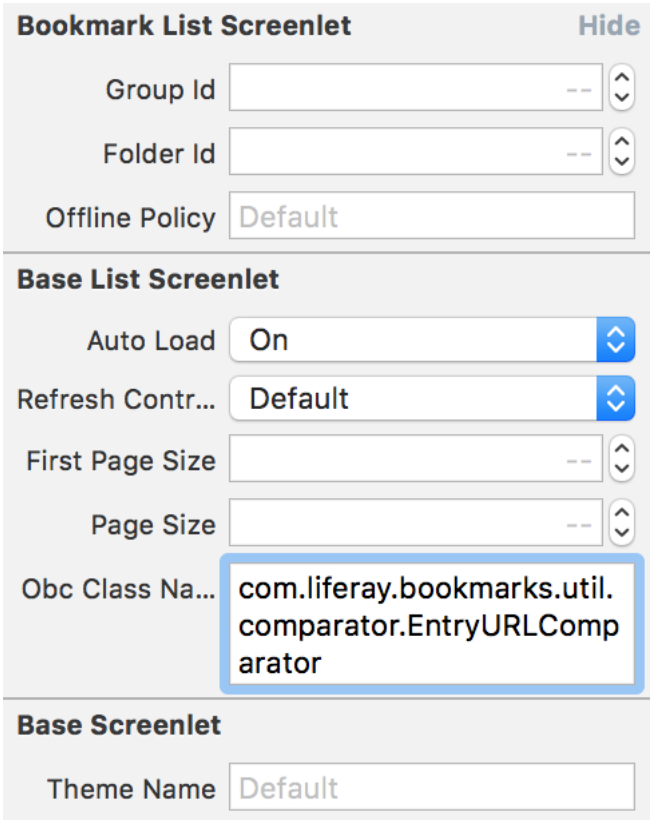


Figure 107.10: To use a comparator, set the *Obc Class Name* property in Interface Builder to the comparator's fully qualified class name.

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

107.14 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.

Figure 107.11: The XIB file for the cell in Bookmark List Screenlet's custom View.

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:

```
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

107.15 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. | `loginWithOAuth2(authentication, userAttributes)` | `LRSession` | Creates a Liferay Session using the default server and the supplied OAuth 2 tokens. This is intended to be used together with OAuth 2 for Liferay Screens. | `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 `loginWithOAuth2` 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
Using OAuth 2 in Liferay Screens for iOS

107.16 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

107.17 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

107.18 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

107.19 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

107.20 Using OAuth 2 in Liferay Screens for iOS

You can use OAuth 2 to authenticate using Login Screenlet with the following OAuth 2 grant types:

- **Authorization Code (PKCE for native apps):** Redirects users to a page in their mobile browser where they enter their credentials. Following login, the browser redirects users back to the mobile app. User credentials can't be compromised via the app because it never accesses them—it uses a token that can be revoked. This is also useful if users don't want to enter their credentials in the app. For example, users may not want to enter their Twitter credentials directly in a 3rd-party Twitter app, preferring instead to authenticate via Twitter's official site. Note that the site you redirect to for authentication must have OAuth 2 implemented.
- **Resource Owner Password:** Users authenticate by entering their credentials directly in the app.

- **Client Credentials:** Authenticates without requiring user interaction. This is useful when the app needs to access its own resources, not those of a specific user.

This tutorial shows you how to use these grant types with Login Screenlet. Note that before getting started, you may want to see Liferay DXP's OAuth 2.0 documentation for instructions on registering an OAuth 2.0 application in the portal.

Authorization Code (PKCE)

Follow these steps to use the Authorization Code grant type with Login Screenlet:

1. Configure the URL where the mobile browser redirects after the user authenticates. To do this, follow the first two steps in the Mobile SDK's Authorization Code instructions. Note that you must configure this URL in both the portal and your iOS app.

2. Set Login Screenlet's `loginMode` attribute to `oauth2Redirect`. There are two ways to do this:

    - In code, as the Login Screenlet instance's `authType` or `loginMode` property:

      ```
      loginScreenlet.authType = .oauth2Redirect
      // or
      loginScreenlet.loginMode = "oauth2redirect"
      ```

      Note that `oauth2redirect` must be a string when set to `loginMode`.

    - In Interface Builder, as the value of the *Login Mode* attribute. Do this the same way you set other Screenlet attributes (via the Attributes inspector, with the Screenlet selected in the storyboard). Be sure to enter `oauth2redirect` with no period preceding it.

3. Set Login Screenlet's `oauth2clientId` attribute to the ID of the portal's OAuth 2 application that you want to use. To find this value, navigate to that application in the portal's OAuth 2 Admin portlet.

4. Set Login Screenlet's `oauth2redirectUrl` attribute to the URL you configured in step 1.

5. In your `AppDelegate`'s `application(_:open:options:)` method, call the `SessionContext` method `oauth2ResumeAuthorization` with the URL. This notifies Liferay Screens when the redirect has been performed. For more information on the `application(_:open:options:)` method, see the section *Handle Incoming URLs* in Apple's documentation on using custom URLs:

   ```
   func application(_ app: UIApplication, open url: URL,
       options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {
           return SessionContext.oauth2ResumeAuthorization(url: url)
   }
   ```

Note that you can cancel the authorization at any time by calling `SessionContext.oauth2Cancel()`.

Resource Owner Password

Follow these steps to use the Resource Owner Password grant type with Login Screenlet:

1. Set Login Screenlet's `loginMode` attribute to `oauth2UsernameAndPassword`. There are two ways to do this:

    - In code, as the Login Screenlet instance's `authType` or `loginMode` property:

      ```
      loginScreenlet.authType = .oauth2UsernameAndPassword
      // or
      loginScreenlet.loginMode = "oauth2UsernameAndPassword"
      ```

      Note that `oauth2UsernameAndPassword` must be a string when setting `loginMode`.

    - In Interface Builder, as the value of the *Login Mode* attribute. Do this the same way you set other Screenlet attributes (via the Attributes inspector, with the Screenlet selected in the storyboard). Be sure to enter `oauth2UsernameAndPassword` with no period preceding it.

2. Set Login Screenlet's `oauth2clientId` attribute to the ID of the OAuth 2 application that you want to use. To find this value, navigate to that application in the OAuth 2 Admin portlet.

3. Set Login Screenlet's `oauth2clientSecret` attribute to the same OAuth 2 application's client secret.

Client Credentials

The OAuth 2 Client Credentials grant type authenticates without requiring user interaction. This is useful when the app needs to access its own resources, not those of a specific user.

Warning: The Client Credentials grant type poses a security risk to the portal. To authenticate without user credentials, the mobile app must contain the OAuth 2 application's client ID and client secret. Anyone who can access those values via the mobile app can also authenticate without user credentials.

Follow these steps to use the Client Credentials grant type in your Screens app:

1. Follow the iOS Mobile SDK instructions for using the Client Credentials grant type.

2. The session object's `authentication` property contains a valid authentication object. Cast it to `LROAuth2Authentication` then pass it to the `authentication` argument of the `SessionContext` method `loginWithOAuth2`:

    ```
    let auth = session.authentication as! LROAuth2Authentication

    SessionContext.loginWithOAuth2(authentication: auth, userAttributes: [:])
    ```

    This initializes the Screens `SessionContext` object, authenticating any Screenlets that you use in the iOS app.

Related Topics

Using OAuth 2 in the iOS Mobile SDK
    Using Screenlets in iOS Apps
    OAuth 2.0

107.21 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.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 iOS 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 iOS Screenlet
    Adding Screenlet Actions
    Create and Use a Connector with Your Screenlet
    Architecture of Liferay Screens for iOS

CHAPTER 108

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. Since 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.

108.1 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 Portal CE 7.0/7.1, or Liferay DXP
- 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.Login`).
    - 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">71</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>71</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 `71` tells Screens that it's communicating with a Liferay CE Portal 7.1 or Liferay DXP 7.1 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

108.2 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:

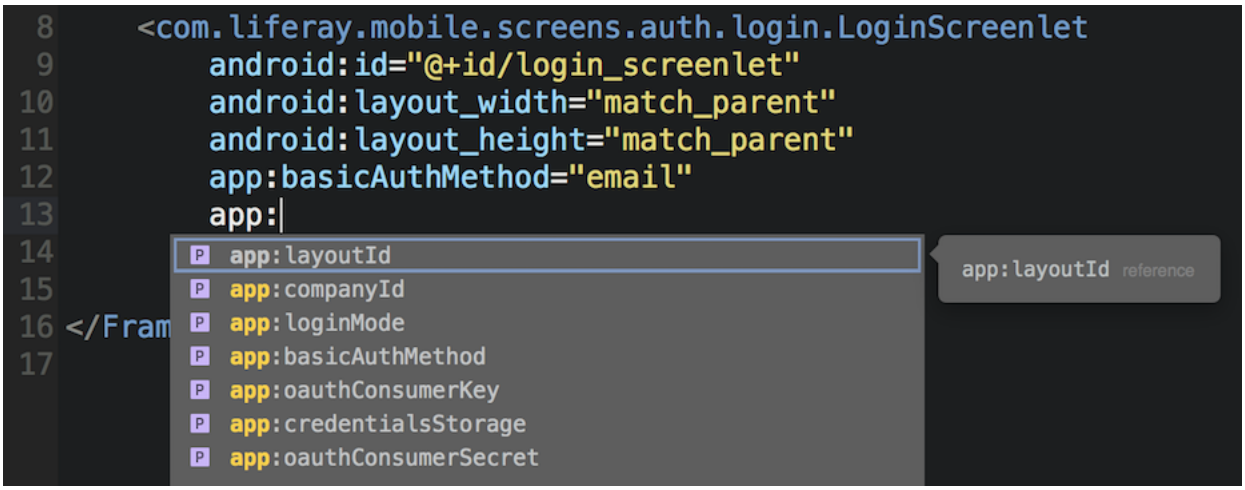


Figure 108.1: You can set a Screenlet's attributes via the app's layout AXML file.

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

108.3 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

108.4 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

108.5 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:

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

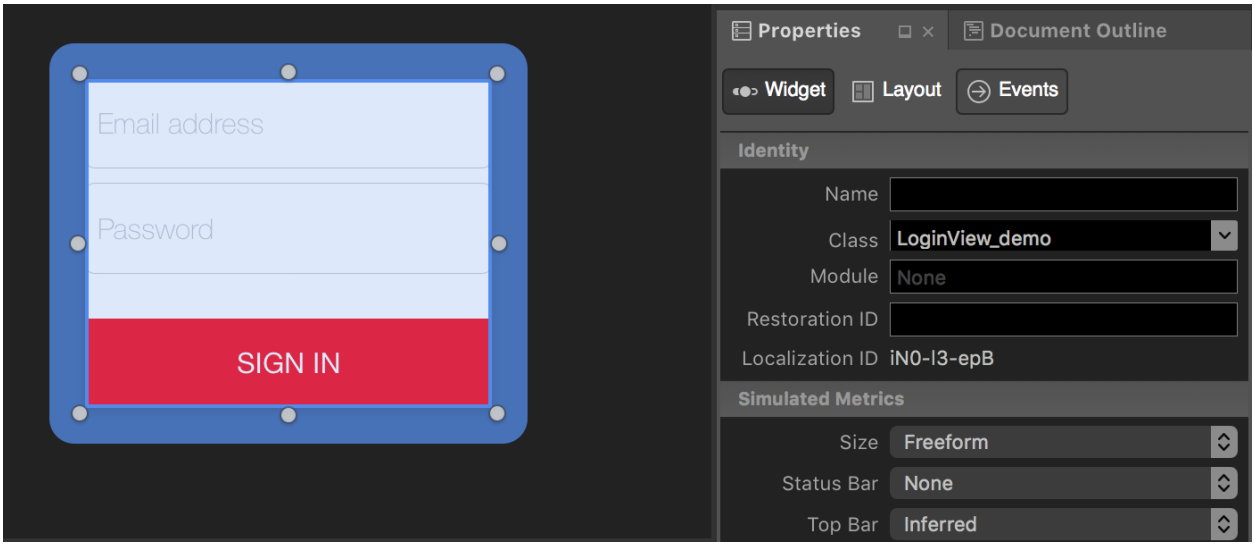


Figure 108.2: Set new View class in XIB Theme file.

108.6 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 109

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:

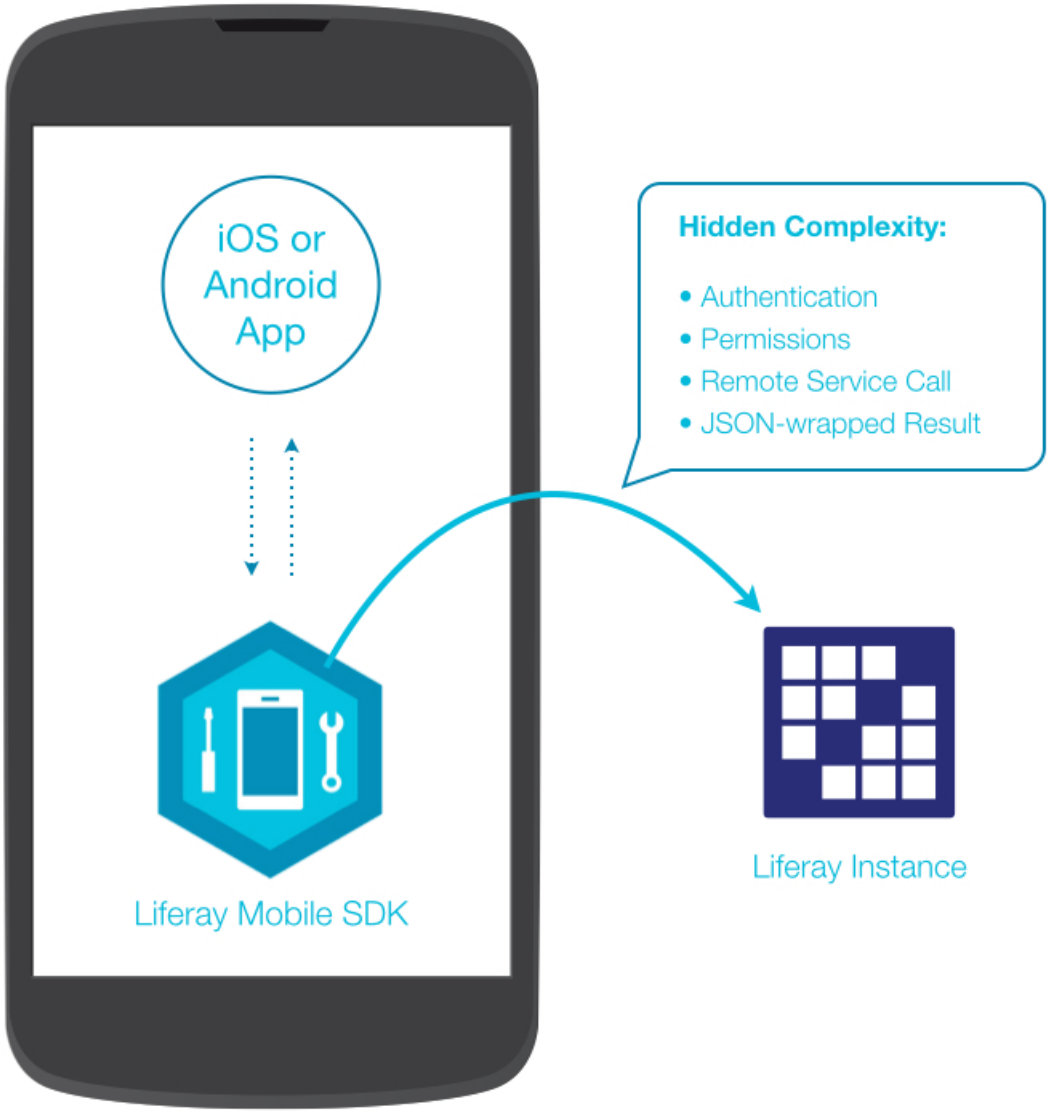


Figure 109.1: Liferay's Mobile SDK enables your native app to communicate with Liferay.

- Building Mobile SDKs

Fasten your seatbelt–it's time to go mobile with Liferay's Mobile SDK!

Related Topics:

Android Apps with Liferay Screens
iOS Apps with Liferay Screens
Building Mobile SDKs

CHAPTER 110

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.

110.1 Related Topics

Invoking Liferay Services in Your Android App

Creating iOS Apps that Use the Mobile SDK

Building Mobile SDKs

110.2 Making Liferay and Custom Portlet Services Available in Your Android App

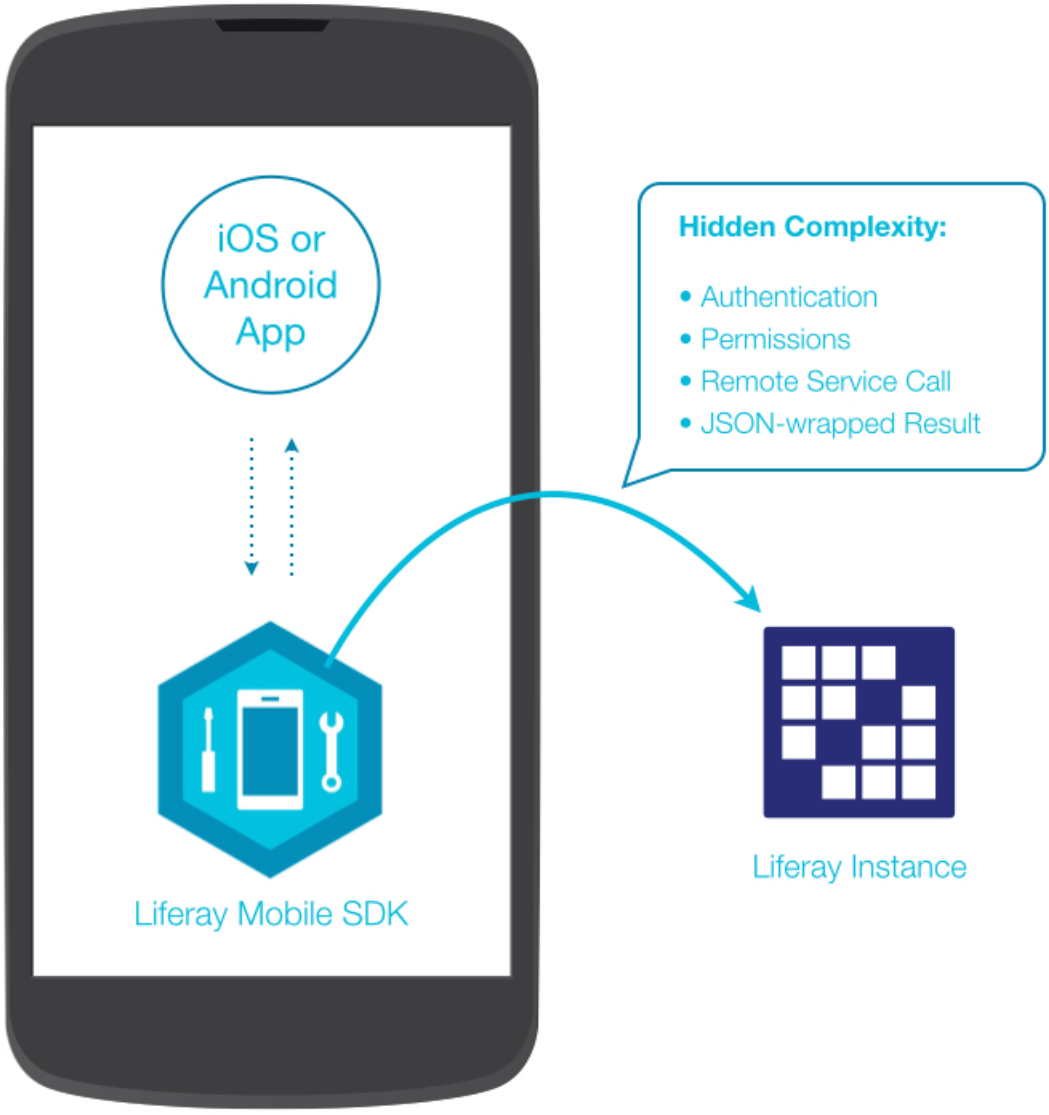


Figure 110.1: Liferay's Mobile SDK enables your native app to communicate with Liferay DXP.

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:

```
repositories {
  jcenter()
  mavenCentral()
}

dependencies {
  compile group: 'com.liferay.mobile', name: 'liferay-android-sdk', version: '7.1.+'
}
```

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`
    - `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

110.3 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

110.4 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

110.5 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

110.6 Using OAuth 2 in the Android Mobile SDK

You can use OAuth 2 to authenticate with the following OAuth 2 grant types:

- **Authorization Code (PKCE for native apps):** Redirects users to a page in their mobile browser where they enter their credentials. Following login, the browser redirects users back to the mobile app. User credentials can't be compromised via the app because it never accesses them—it uses a token that can be revoked. This is also useful if users don't want to enter their credentials in the app. For example, users may not want to enter their Twitter credentials directly in a 3rd-party Twitter app, preferring instead to authenticate via Twitter's official site. Note that the site you redirect to for authentication must have OAuth 2 implemented.
- **Resource Owner Password:** Users authenticate by entering their credentials directly in the app.
- **Client Credentials:** Authenticates without requiring user interaction. This is useful when the app needs to access its own resources, not those of a specific user.

This tutorial shows you how to use these grant types with the Mobile SDK. Note that before getting started, you may want to see Liferay DXP's OAuth 2.0 documentation for instructions on registering an OAuth 2.0 application in the portal.

Authorization Code (PKCE)

To authenticate via the Authorization Code grant type, you must call this `OAuth2SignIn` method:

```
OAuth2SignIn.signInWithRedirect(Activity activity, Session session, String clientId,
    List<String> scopes, Uri redirectUri, CustomTabsIntent customTabsIntent)
```

Here are descriptions of this method's parameters:

- `activity`: The activity to use to present the mobile browser.
- `session`: The session that you want to authenticate. Its `server` property must be set.
- `clientId`: The ID of the portal's OAuth 2 application that you want to use. To find this value, navigate to that application in the portal's OAuth 2 Admin portlet.

- `scopes`: The portal permissions to request. You can define a set of permissions associated with an OAuth 2 application in the portal's OAuth 2 Admin portlet. Use this property to request a subset of those permissions.
- `redirectUri`: The URI that the user is redirected to after successful login in the mobile browser. You must configure this URI in the portal via the OAuth 2 Admin portlet, and associate the URI with the Android app.
- `customTabsIntent`: The object used to customize the appearance of the mobile browser window shown for authentication.

This `signInWithRedirect` method opens the mobile browser to initiate authentication. You must also configure the redirect URI in your Android app, which sends the user back to the Android app when authentication completes.

Here's an example of this workflow:

1. Configure the redirect URI in the portal via the OAuth2 Administration portlet. In the portal, navigate to *Control Panel → Configuration → OAuth2 Administration* and select or create the OAuth 2 application you want to use. Then enter the redirect URI in the *Callback URIs* field. The redirect URI in this example is `my-app://my-app`:

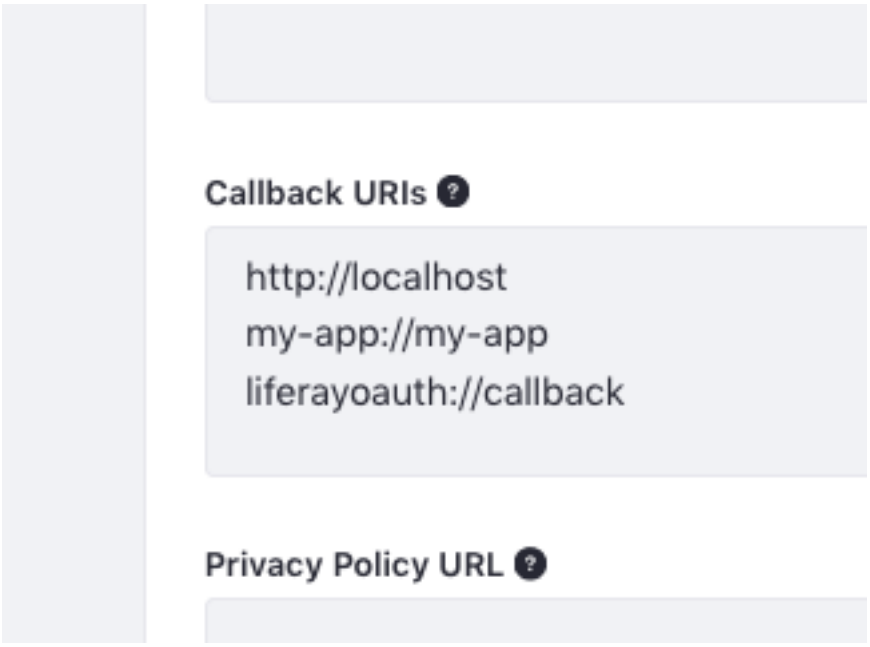


Figure 110.2: Enter the redirect URI in the portal's OAuth 2 application you want to use.

2. In your Android app, register your redirect URI by editing the `AndroidManifest.xml` file. Add the following code to this file:

```
<activity
android:name="com.liferay.mobile.android.auth.oauth2.OAuth2RedirectActivity"
    tools:node="replace">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
```

```
        <data android:scheme="<your-scheme>"/>
    </intent-filter>
</activity>
```

If you don't want to use a custom scheme and you need to use HTTPS, add this instead:

```
<activity
    android:name="com.liferay.mobile.android.auth.oauth2.OAuth2RedirectActivity"
    tools:node="replace">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="https"
              android:host="your.custom.domain"
              android:path="/oauth2redirect"/>
    </intent-filter>
</activity>
```

3. In the activity in which you want to perform the authentication, call `OAuth2SignIn.signInWithRedirect`:

```
private void doLogin() {
    Session session = new SessionImpl("http://my-server.com");
    OAuth2SignIn.signInWithRedirect(this, session, "54321", new ArrayList<>(),
    Uri.parse("my-app://my-app"), null);
}
```

This opens the mobile browser with the login page.

4. In the same activity, you must override the method `onActivityResult` to receive the authentication's result. In this method, you do this by calling the method `OAuth2SignIn.resumeAuthorizationFlowWithIntent`:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent intent) {
    super.onActivityResult(requestCode, resultCode, intent);

    if (requestCode == OAuth2SignIn.REDIRECT_REQUEST_CODE) {
        Session session = new SessionImpl("http://my-server.com");
        OAuth2SignIn.resumeAuthorizationFlowWithIntent(this, session, intent,
        new SessionCallback() {
            @Override
            public void onSuccess(Session session) {
                // Login success
            }

            @Override
            public void onFailure(Exception e) {
                // Login error
            }
        );
    }
}
```

Resource Owner Password

Authenticating via the Resource Owner Password grant type is similar to authenticating via the PKCE grant type, except you don't need to configure a redirect URL. You instead handle the user's credentials directly in your Android app via a different `OAuth2SignIn` method:

```
public static Session signInWithUsernameAndPassword(String username, String password, Session session,
    String clientId, String clientSecret, List<String> scopes, SessionCallback callback)
```

Compared to the `OAuth2SignIn.signInWithRedirect` method used for the PKCE grant type, this one requires the user's credentials instead of a redirect URI. It also requires the OAuth 2 application's client secret from the portal, and a callback.

Here are descriptions of this method's parameters:

- `username`: The user's username.
- `password`: The user's password.
- `session`: The session that you want to authenticate. Its `server` property must be set.
- `clientId`: The ID of the portal's OAuth 2 application that you want to use. To find this value, navigate to that application in the portal's OAuth 2 Admin portlet.
- `clientSecret`: The client secret of the same OAuth 2 application in the portal.
- `scopes`: The portal permissions to request. You can define a set of permissions associated with an OAuth 2 application in the portal's OAuth 2 Admin portlet. Use this property to request a subset of those permissions.
- `callback`: A `SessionCallback` object containing the authentication's result. If authentication succeeds, you receive a non-null session containing the authentication; otherwise you receive an error.

Note: You can call the `OAuth2SignIn.signInWithUsernameAndPassword` method without a callback by passing `null` in place of the callback. This causes the request to execute synchronously. If you provide a callback, the request is executed asynchronously in another thread and the callback receives the response.

Here's an example of calling the `OAuth2SignIn.signInWithUsernameAndPassword` method for the Resource Owner Password grant type. After creating the session, the method is called with a `SessionCallback` created as an anonymous inner class:

```
Session session = new SessionImpl("http://my-server.com");

OAuth2SignIn.signInWithUsernameAndPassword("username", "password", session, "12345", "12345",
    new ArrayList<>(), new SessionCallback() {

    @Override
    public void onSuccess(Session session) {
        // Login correct
    }

    @Override
    public void onFailure(Exception e) {
        // Login error
    }
});
```

Client Credentials

The OAuth 2 Client Credentials grant type authenticates without requiring user interaction. This is useful when the app needs to access its own resources, not those of a specific user.

Warning: The Client Credentials grant type poses a security risk to the portal. To authenticate without user credentials, the mobile app must contain the OAuth 2 application's client ID and client

secret. Anyone who can access those values via the mobile app can also authenticate without user credentials.

To authenticate with the Client Credentials grant type, you must call the `OAuth2SignIn.clientCredentialsSignIn` method. Note that this method lacks arguments for user credentials or redirect URIs:

```
public static Session clientCredentialsSignIn(Session session, String clientId, String clientSecret,
    List<String> scopes, SessionCallback callback)
```

Here are descriptions of this method's parameters:

- `sesssion`: The session that you want to authenticate. Its `server` property must be set.
- `clientId`: The ID of the portal's OAuth 2 application that you want to use. To find this value, navigate to that application in the portal's OAuth 2 Admin portlet.
- `clientSecret`: The client secret of the same OAuth 2 application in the portal.
- `scopes`: The portal permissions to request. You can define a set of permissions associated with an OAuth 2 application in the portal's OAuth 2 Admin portlet. Use this property to request a subset of those permissions.
- `callback`: A `SessionCallback` object containing the authentication's result. If authentication succeeds, you receive a non-null session containing the authentication; otherwise you receive an error.

Note: You can call the `OAuth2SignIn.clientCredentialsSignIn` method without a callback by passing `null` in place of the callback. This causes the request to execute synchronously. If you provide a callback, the request is executed asynchronously in another thread and the callback receives the response.

Here's an example of calling the `OAuth2SignIn.clientCredentialsSignIn` method for the Resource Owner Password grant type. After creating the session, the method is called with a `SessionCallback` created as an anonymous inner class:

```
Session session = new SessionImpl("http://my-server.com");

OAuth2SignIn.clientCredentialsSignIn(session, "12345", "12345", new ArrayList<>(),
    new SessionCallback() {

    @Override
    public void onSuccess(Session session) {
        // Login correct
    }

    @Override
    public void onFailure(Exception e) {
        // Login error
    }
});
```

Related Topics

Using OAuth 2 in Liferay Screens for Android

OAuth 2.0

CHAPTER 111

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 your iOS project. This makes Liferay DXP's services available in your app.

111.1 Related Topics

Invoking Liferay Services in Your iOS App
Building Mobile SDKs
Creating Android Apps that Use the Mobile SDK

111.2 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.

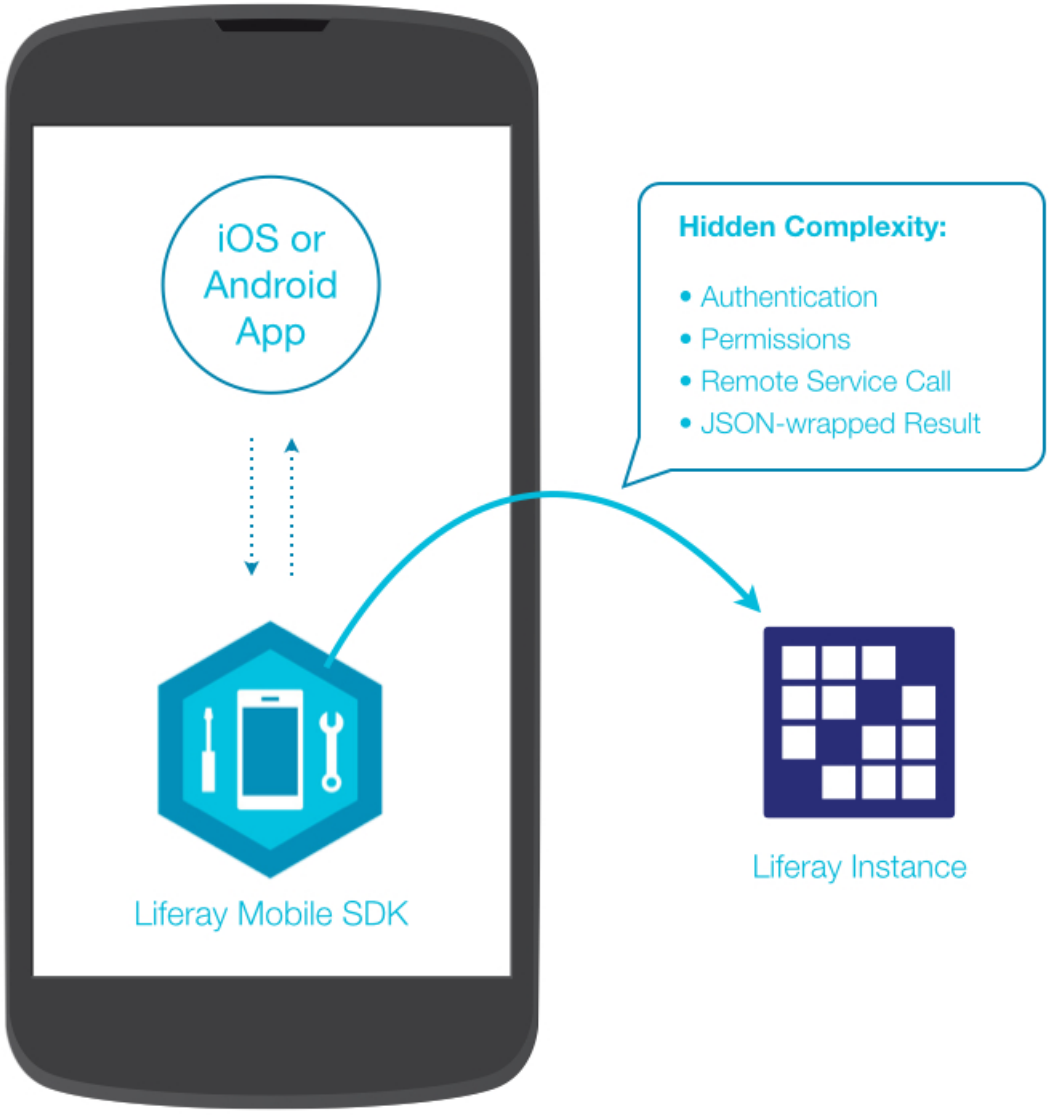


Figure 111.1: Liferay's Mobile SDK enables your native app to communicate with Liferay.

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 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

111.3 Invoking Liferay Services in Your iOS App

Once the appropriate Mobile SDKs are set up in your iOS project, you can access and invoke @product 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 @product 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 2 authentication. For instructions on this, see the tutorial Using OAuth 2 in the iOS Mobile SDK.

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

111.4 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

111.5 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

111.6 Using OAuth 2 in the iOS Mobile SDK

You can use OAuth 2 to authenticate using the following OAuth 2 grant types:

- **Authorization Code (PKCE for native apps):** Redirects users to a page in their mobile browser where they enter their credentials. Following login, the browser redirects users back to the mobile app. User credentials can't be compromised via the app because it never accesses them—it uses a token that can be easily revoked. This is also useful if users don't want to enter their credentials in the app. For example, users may not want to enter their Twitter credentials directly in a 3rd-party Twitter app, preferring instead to authenticate via Twitter's official site. Note that the site you redirect to for authentication must have OAuth 2 implemented.
- **Resource Owner Password:** Users authenticate by entering their credentials directly in the app.
- **Client Credentials:** Authenticates without requiring user interaction. This is useful when the app needs to access its own resources, not those of a specific user.

This tutorial shows you how to use these grant types with the Mobile SDK. Note that before getting started, you may want to see Liferay DXP's OAuth 2.0 documentation for instructions on registering an OAuth 2.0 application in the portal.

Authorization Code (PKCE)

To authenticate via the Authorization Code grant type, you must call the following `LROAuth2SignIn` method:

```
LROAuth2SignIn.signIn(withRedirectURL: URL, session: LRSession, clientId: String,
    scopes: [String], callback: (LRSession?, Error?) -> Void) -> LROAuth2AuthorizationFlow
```

Here are descriptions of this method's parameters:

- `withRedirectURL`: The URL that the user is redirected to after successful login in the mobile browser. You must configure this URL in the portal via the OAuth 2 Admin portlet, and associate the URL with the iOS app.
- `session`: The session that you want to authenticate. Its `server` property must be set.
- `clientId`: The ID of the portal's OAuth 2 application that you want to use. To find this value, navigate to that application in the portal's OAuth 2 Admin portlet.
- `scopes`: The portal permissions to request. You can define a set of permissions associated with an OAuth 2 application in the portal's OAuth 2 Admin portlet. Use this property to request a subset of those permissions.
- `callback`: A function called with the authentication's result. If authentication succeeds, you receive a non-null session containing the authentication; otherwise you receive an error.

This `LROAuth2SignIn.signIn` method returns an `LROAuth2AuthorizationFlow` object that represents an ongoing authentication request. You must save this as an `AppDelegate` property and then call the `LROAuth2AuthorizationFlow` method `resumeAuthorizationFlowWithURL` in the `application(_:open:options:)` method.

Here's an example of this workflow:

1. Configure the redirect URI in the portal via the OAuth2 Administration portlet. In the portal, navigate to *Control Panel → Configuration → OAuth2 Administration* and select or create the OAuth 2 application you want to use. Then enter the redirect URI in the *Callback URIs* field. The redirect URI in this example is `my-app://my-app`:

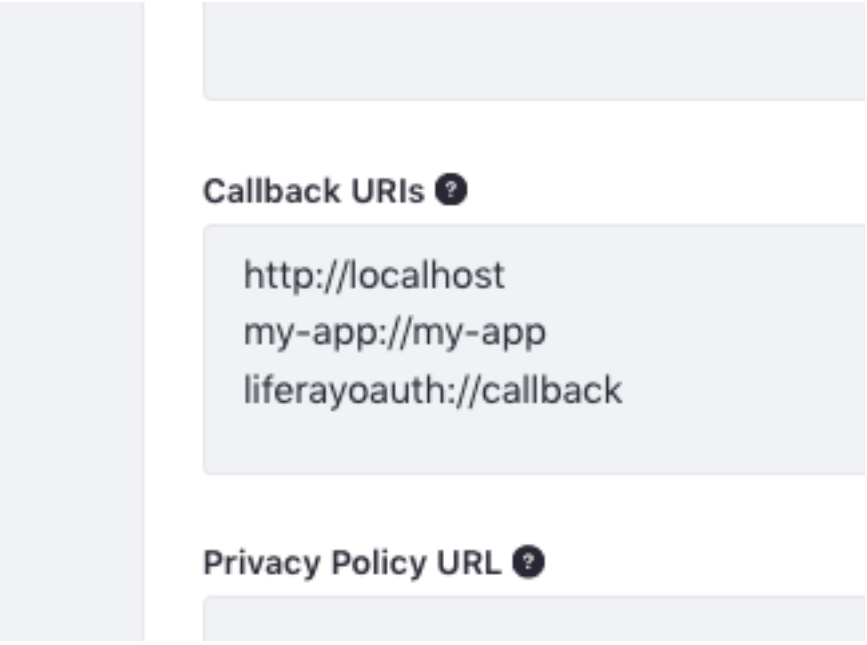


Figure 111.2: Enter the redirect URL in the portal's OAuth 2 application you want to use.

2. In your iOS app, register your redirect URL via the *Info* tab in your project settings. For instructions on this, see the section *Register Your URL Scheme* in Apple's documentation on using custom URLs.

Figure 111.3: Register the redirect URL in your iOS app.

3. In your `AppDelegate`, create a `LROAuth2AuthorizationFlow` property. You'll set this property later when you call the `LROAuth2SignIn.signIn` method:

```
var authorizationFlow: LROAuth2AuthorizationFlow?
```

4. In the view controller in which you'll call `LROAuth2SignIn.signIn`, define the callback that runs with the authentication's result. This callback can perform any action you need. In this example, if the authentication succeeds the callback prints a success message and calls a sample method that tests the session's user credentials; otherwise it prints an error message:

```
let oauth2Callback: (LRSession?, Error?) -> Void = { session, error in
    if let session = session {
        print("Login successful")
        testCredentials(session: session)
    }
    else {
        print(error!)
    }
}
```

5. In the same view controller, call the `LROAuth2SignIn.signIn` method with the above parameters. Set the resulting `LROAuth2AuthorizationFlow` to the `AppDelegate` property you created in step 3. This example does this in a `loginWithRedirect()` method:

```
func loginWithRedirect() {
    let session = LRSession(server: "http://my-server.com")
    let redirectUrl = URL(string: "my-app://my-app")!
    let clientIdRedirect = "54321"

    let authorizationFlow = LROAuth2SignIn.signIn(withRedirectURL: redirectUrl,
        session: session, clientId: clientIdRedirect, scopes: [], callback: oauth2Callback)

    (UIApplication.shared.delegate as! AppDelegate).authorizationFlow = authorizationFlow
}
```

6. In your `AppDelegate`'s `application(_:open:options:)` method, call the `LROAuth2AuthorizationFlow` method `resumeAuthorizationFlow` with the URL. For more information on the `application(_:open:options:)` method, see the section *Handle Incoming URLs* in Apple's documentation on using custom URLs:

```
@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    ...

    var authorizationFlow: LROAuth2AuthorizationFlow?

    func application(_ app: UIApplication, open url: URL,
        options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {

        if let authorizationFlow = authorizationFlow {
            return authorizationFlow.resumeAuthorizationFlow(with: url)
        }
    }

    ...

}
```

Resource Owner Password

Authenticating via the Resource Owner Password grant type is similar to authenticating via the PKCE grant type, except you don't need a redirect URL. You instead handle the user's credentials directly in your iOS app via a slightly different `LROAuth2SignIn.signIn` method:

```
LROAuth2SignIn.signIn(withUsername: String, password: String, session: LRSession, clientId: String,
    clientSecret: String, scopes: [String], callback: (LRSession?, Error?) -> Void) -> LRSession?
```

Compared to the `LROAuth2SignIn.signIn` method used for the PKCE grant type, this one requires the user's credentials instead of a redirect URL. It also requires the OAuth 2 application's client secret from the portal.

Here are descriptions of this method's parameters:

- `withUsername`: The user's username.
- `password`: The user's password.
- `session`: The session that you want to authenticate. Its `server` property must be set.
- `clientId`: The ID of the portal's OAuth 2 application that you want to use. To find this value, navigate to that application in the portal's OAuth 2 Admin portlet.
- `clientSecret`: The client secret of the same OAuth 2 application in the portal.
- `scopes`: The portal permissions to request. You can define a set of permissions associated with an OAuth 2 application in the portal's OAuth 2 Admin portlet. Use this property to request a subset of those permissions.
- `callback`: A function called with the authentication's result. If authentication succeeds, you receive a non-null session containing the authentication; otherwise you receive an error.

Note: You can call the `LROAuth2SignIn.signIn` method without a callback by passing `nil` as the `callback` argument. This causes the request to execute synchronously. If you provide a callback, the request is executed asynchronously in another thread and the callback receives the response.

Follow these steps to call the `LROAuth2SignIn.signIn` method for the Resource Owner Password grant type:

1. If you want to provide a callback, define it in the view controller in which you'll call `LROAuth2SignIn.signIn`. This callback can perform any action you need. In this example, if the authentication succeeds the callback prints a success message and calls a sample method that tests the session's user credentials; otherwise it prints an error message:

```
let oauth2Callback: (LRSession?, Error?) -> Void = { session, error in
    if let session = session {
        print("Login successful")
        testCredentials(session: session)
    }
    else {
        print(error!)
    }
}
```

2. In the same view controller, call the `LROAuth2SignIn.signIn` method with the above parameters. This example does this in a `loginWithUsernameAndPassword()` method:

```
func loginWithUsernameAndPassword() {
    if password.isEmpty {
        fatalError("you have to enter the password")
    }

    let session = LRSession(server: "http://my-server.com")
    let clientId = "12345"
    let clientSecret = "12345"

    _ = try? LROAuth2SignIn.signIn(withUsername: "test@example.com", password: password,
                session: session, clientId: clientId, clientSecret: clientSecret, scopes: [],
                callback: oauth2Callback)
}
```

Client Credentials

The OAuth 2 Client Credentials grant type authenticates without requiring user interaction. This is useful when the app needs to access its own resources, not those of a specific user.

Warning: The Client Credentials grant type poses a security risk to the portal. To authenticate without user credentials, the mobile app must contain the OAuth 2 application's client ID and client secret. Anyone who can access those values via the mobile app can also authenticate without user credentials.

To authenticate with the Client Credentials grant type, you must call the `LROAuth2SignIn.signIn` method that lacks arguments for user credentials or redirect URLs:

```
LROAuth2SignIn.clientCredentialsSignIn(with: LRSession, clientId: String,
    clientSecret: String, scopes: [String], callback: (LRSession?, Error?) -> Void)
```

Here are descriptions of this method's parameters:

- `with`: The session that you want to authenticate. Its `server` property must be set.
- `clientId`: The ID of the portal's OAuth 2 application that you want to use. To find this value, navigate to that application in the portal's OAuth 2 Admin portlet.
- `clientSecret`: The client secret of the same OAuth 2 application in the portal.
- `scopes`: The portal permissions to request. You can define a set of permissions associated with an OAuth 2 application in the portal's OAuth 2 Admin portlet. Use this property to request a subset of those permissions.
- `callback`: A function called with the authentication's result. If authentication succeeds, you receive a non-null session containing the authentication; otherwise you receive an error.

Note: You can call the `LROAuth2SignIn.signIn` method without a callback by passing `nil` as the `callback` argument. This causes the request to execute synchronously. If you provide a callback, the request is executed asynchronously in another thread and the callback receives the response.

Follow these steps to call the `LROAuth2SignIn.signIn` method for the Client Credentials grant type:

1. If you want to provide a callback, define it in the view controller in which you'll call `LROAuth2SignIn.signIn`. This callback can perform any action you need. In this example, if the authentication succeeds the callback prints a success message and calls a sample method that tests the session's user credentials; otherwise it prints an error message:

   ```
   let oauth2Callback: (LRSession?, Error?) -> Void = { session, error in
       if let session = session {
           print("Login successful")
           testCredentials(session: session)
       }
       else {
           print(error!)
       }
   }
   ```

2. In the same view controller, call the `LROAuth2SignIn.signIn` method with the above parameters. This example does this in a `loginWithClientCredentials()` method:

```
func loginWithClientCredentials() {

    let session = LRSession(server: "http://my-server.com")
    let clientId = "12345"
    let clientSecret = "12345"

    _ = try? LROAuth2SignIn.clientCredentialsSignIn(with: session, clientId: clientId,
                clientSecret: clientSecret, scopes: [], callback: oauth2Callback)
}
```

Related Topics

Using OAuth 2 in Liferay Screens for iOS

OAuth 2.0

111.7 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:

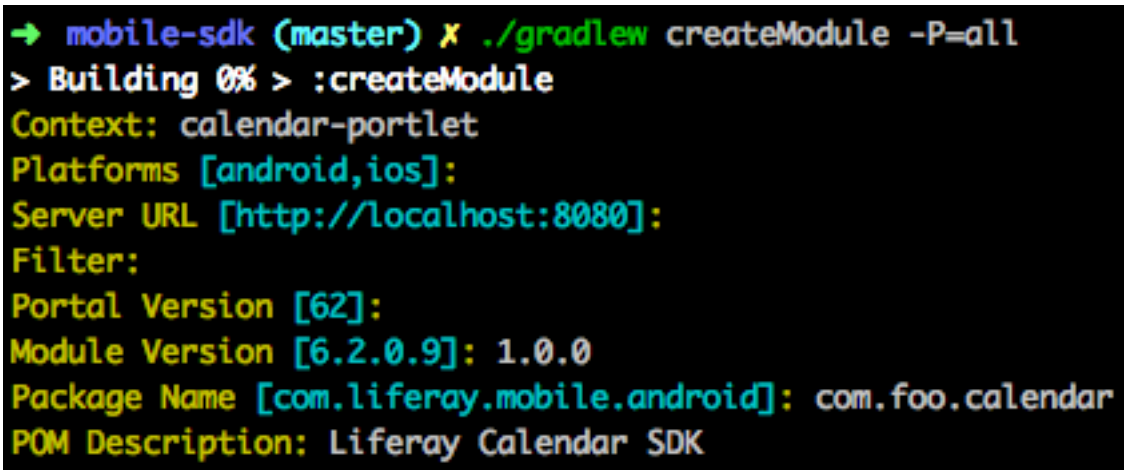


Figure 111.4: The Mobile SDK Builder's wizard lets you specify property values for building your module.

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.
- `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 112

TRACKING CUSTOM ASSETS

Liferay Analytics Cloud can detect and analyze built-in Liferay DXP assets like Forms, Blogs, Documents and Media, and Web Content. To analyze assets in your custom app, however, you must tag your app's HTML so the Analytics Cloud JavaScript plugin can detect and track user interaction with your assets.

112.1 Asset Events

The Analytics Cloud JavaScript plugin contains the following events that you can track:

`AssetClicked`: User clicks the asset area. Also carries information about the tag clicked.

`AssetDepthReached`: Scroll event in the asset area. Also carries information about the content depth the user reached (e.g., how far down a blog post the user scrolled).

`AssetViewed`: User views the asset.

`AssetDownloaded`: User clicks a link that downloads the asset.

`AssetSubmitted`: Form submission in the asset area. This requires an input type of `submit` to be placed under an HTML `form` element.

112.2 Required Metadata

You must have the following information to enable tracking for a custom entity. You must specify this information in HTML via the attributes listed:

Asset Type (String): The asset type to track. The HTML attribute for this is `data-analytics-asset-type`. Note that the value for this attribute **is not** your entity's exact type. For custom entities, this value is always `custom`.

Asset ID (String): The asset's unique identifier. The HTML attribute for this is `data-analytics-asset-id`.

Asset Category (String, Optional): The category of the custom app that contains the asset. The HTML attribute for this is `data-analytics-asset-category`. You can use this to identify the custom app by name (e.g., "polls"). Note that within a category, all asset IDs must be unique.

Asset Title (String, Optional): The asset's title. The HTML attribute for this is `data-analytics-asset-title`.

112.3 Tracking Asset Events

For example, if you want to track a poll in a custom Polls app, you might use HTML like this:

```
<div>
<h1> What's your favorite food? </h1>
<form action="/submit.php">
<div>
  <input type="radio" id="sushi" name="food" value="Sushi"
     checked>
  <label for="sushi">Sushi</label>
</div>
<div>
  <input type="radio" id="pizza" name="food" value="Pizza">
  <label for="pizza">Pizza</label>
</div>
<input type="submit" value="Submit" />
</form>
</div>
```

To track this poll's events, add the above attributes to the `div`:

```
<div data-analytics-asset-type="custom"
  data-analytics-asset-id="favorite-food-poll"
  data-analytics-asset-category="polls"
  data-analytics-asset-title="What is your favorite food Poll">

<h1> What's your favorite food? </h1>
...
</div>
```

Note that you must add these attributes to each individual asset that you want Analytics Cloud to track. However, you can populate the attributes' values via a script, therefore automating this process for each asset.

Tracking Downloads

To track downloads, you must tag the element that triggers the action with this attribute:

```
data-analytics-asset-action="download"
```

For example, here's the above poll with a download link for a PDF file that contains the poll's instructions:

```
<div data-analytics-asset-type="custom"
  data-analytics-asset-id="favorite-food-poll"
  data-analytics-asset-category="polls"
  data-analytics-asset-title="What is your favorite food Poll">

<a href="/poll-instructions.pdf" data-analytics-asset-action="download">Download the Poll Instructions </a>

<h1> What's your favorite food? </h1>
...
</div>
```

112.4 Related Topics

Asset Framework
    Liferay Analytics Cloud Admin Guide

CHAPTER 113

WEB EXPERIENCE MANAGEMENT

Web Experience Management encompasses Liferay's features and tools for building Sites and creating content. Many of these, like Web Content Management and Page Templates, are graphical tools used by administrators and marketers. Others, like Page Fragments, let web developers to flex their muscles in content creation. These tutorials cover where web development intersects with user experience and how to use Liferay's Web Experience frameworks to integrate custom applications into Liferay DXP.

Specifically, you'll learn about

- Developing Fragments
- Screen Navigation

CHAPTER 114

DEVELOPING PAGE FRAGMENTS

The goal of Page Fragments is to take a mock-up design and realize it on a web page as accurately as possible. To do this, developers are given a space where they have a "blank slate." You have three tools at your disposal to accomplish this:

HTML: The markup of the fragment. Fragments use standard HTML with special tags to add dynamic behavior.

CSS: CSS styles and positions the fragment's markup.

JavaScript: JavaScript provides dynamic behavior to the fragment.

The HTML, CSS, and JavaScript are all completely standard, just like anywhere else on the web, but are also enhanced with Liferay-specific features. You can specify text and images as editable, as well as providing for "rich text": that is, text with additional formatting. Liferay portlets can also be embedded in Fragments as "widgets," making pages with Fragments more dynamic than regular web content.

114.1 Creating a Fragment

The first example is simple. If you aren't sure about the basics of Fragments and Collections, you should read Creating Page Fragments first so you know what you're getting yourself into.

1. Go to *Site Administration* and for your selected site click *Build → Fragments*.
2. Create a new Collection named *Developing Fragments*.
3. Inside of the new Collection, create a new Fragment named *Basic Fragment*.

You're now on the Fragment editing page. There are four panes on this screen. You enter HTML in the top left pane, CSS in the top right, JavaScript in the bottom left, and preview the results in the bottom right.

You can look at the three editing panes as if each were writing to a separate file. Everything in the HTML pane goes to `index.html`, the CSS pane goes to `index.css`, and the JavaScript one goes to `index.js`. The preview pane renders everything as it will look on the page. Here's how this works:

1. Add the following code inside the `div` in the HTML pane:

```
<h1>
    This text is styled, so it will look pretty!
</h1>
```

2. Add the following code to the CSS pane inside the fragment block:

```
width: 100%;
background-color: white;
text-align: center;
padding: 2em 0;
font-size: 28px;
```

3. Add this code below it and replace the ##### with the auto-generated number provided for the fragment

```
.fragment_##### h1 {
  font-weight: 100;
  width: calc(100% - 1em);
  max-width: 60ch;
  margin: 0 auto;
  font-size: 28px;
 }
```

4. Click *Publish* to save your work and make it available to add to a content page.

As you work, you can observe the preview in the preview pane.

From here, the Fragment can be used to create a Content Page. To see this process in action, see the Building Content Pages from Fragments tutorial. Next, you'll explore some Fragment-specific tags.

114.2 Fragment Specific Tags

While HTML, CSS, and JavaScript are universal tools for building websites, Liferay DXP includes some additional tools to make fragments more powerful. One tag can make text editable not just in the HTML editor, but also at the time of publishing. The other enables you to embed Liferay portlets into your fragment in the form of "Widgets."

Making Text Editable

Making text editable allows the marketer or web admin to modify the text before publishing it. This way, you can reuse a single fragment with different headings or different text for different pages. Fragments make creating content easy and this feature can save you the work of duplicating work just to change the text.

You can make any text of a fragment editable by enclosing it in an `<lfr-editable>` tag like this:

```
<lfr-editable id="unique-id" type="text">
   This is editable text!
</lfr-editable>
```

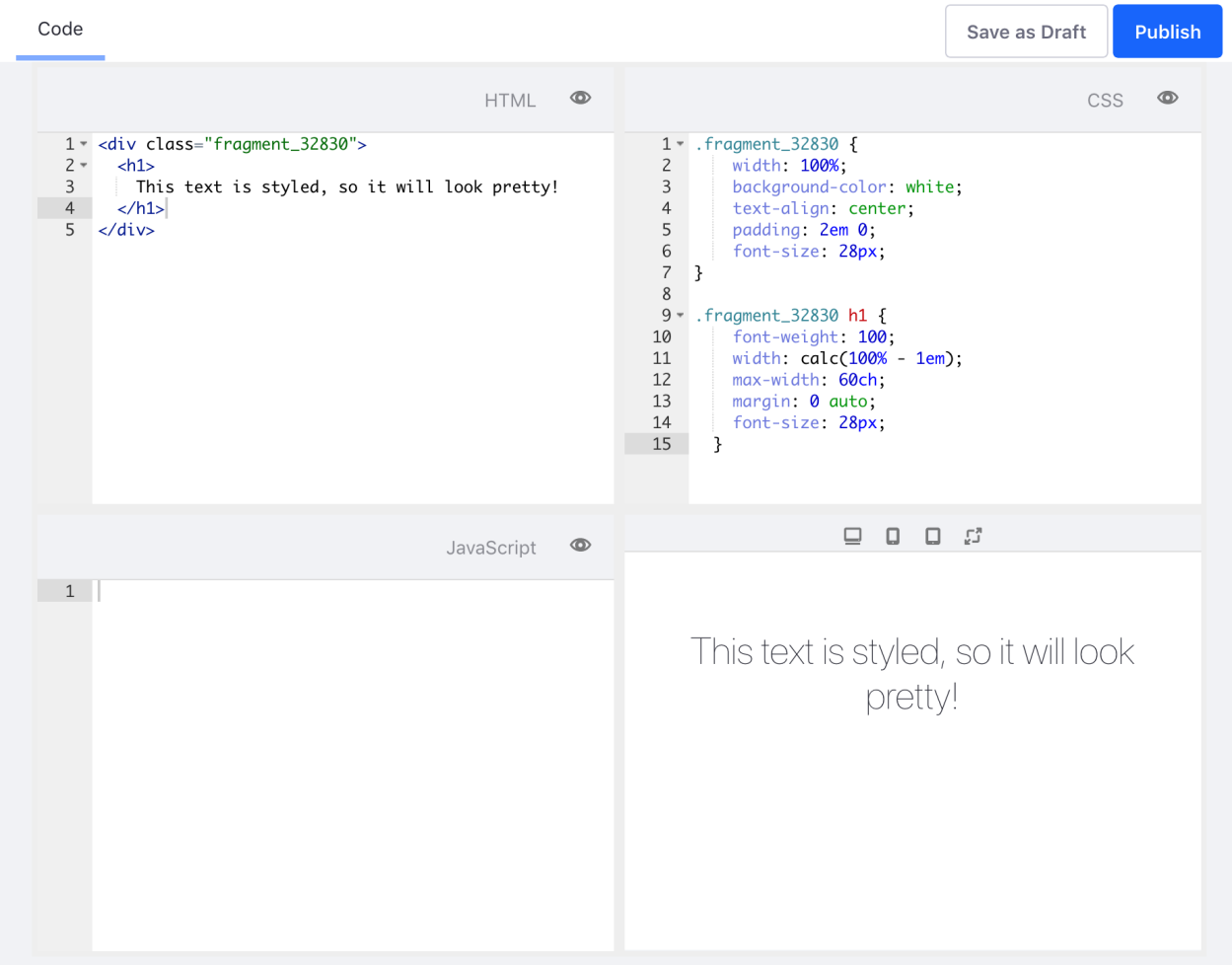


Figure 114.1: The Fragment editor with HTML and CSS code and a live preview.

The `lfr-editable` tag doesn't render without a unique `id`.

The following three `type` options are available inside of of `lfr-editable` tag:

`text`: Creates a space for plain text that can be edited before publishing.

`image`: Must contain a valid `<img>` tag which can then be replaced with any image before publishing—including those from Documents and Media.

`rich-text`: Provides rich text formatting, such as bold, italics, underline, links, and predefined styles.

Note: If you want to make text inside an HTML element editable, you must use the `rich-text` type. The `text` type strips HTML formatting out of the text before rendering.

Now add editable text to a Fragment, add the Fragment to a Content Page, and then edit the text before publishing:

1. Go back into the Fragment you were working on before.

2. Inside the <h1> in the HTML pane, surround the text with <lfr-editable> tags so that it looks like this.

```
<h1>
  <lfr-editable id="heading" type="text">
      This text is styled, so it will look pretty!
  </lfr-editable>
</h1>
```

3. Click *Publish*.

Now your fragment contains editable text. Add it to a Content Page to be published:

1. Go to *Navigation → Site Pages*.
2. Select the *Page Templates* tab at the top.
3. Create a new Collection named *Templates for Developing Fragments* and a New Page Template inside of it named *Editable Page Template*.
4. From the menu on the right, add your fragment to the page.
5. Click on the text that you defined as editable and change it.

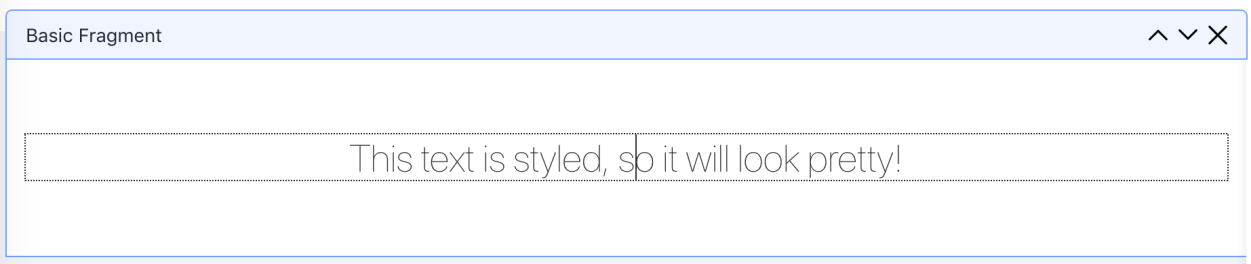


Figure 114.2: You can edit text in the Page Template editor.

Similarly, the `rich-text` type provides the same functionality with additional features in the editor.

The template saves automatically, and when it is turned into a page, the new text is displayed.

Making Images Editable

Like text, you can set images images as editable. An editable image can be selected from the user's local files or from the Documents and Media Library. You must be careful with defining styles for editable images since an image that without the proper dimensions or that is forced into a poorly sized space can have major negative effects on your layout.

Images use the same `<lfr-editable>` tag as text, but with the `image` type, like this:

```
<lfr-editable id="unique-id" type="image">
   <img src="...">
</lfr-editable>
```

After you add the `lfr-editable` tag with the type `image` to a Fragment, when you add that Fragment to a page, you can then click on the editable image and select a replacement.

Including Widgets Within A Fragment

You can add more dynamic behavior to a Fragment by including a widget. Currently, you can only embed a portlet as a widget, but other types of widgets will be available in the future.

To include a widget you need to know its registered name. For example, the Site Navigation Menu portlet is registered as `nav`. Each portlet which is registered has an `lfr-widget-[name]` tag that's used to embed it. For example: the Navigation Menu tag is `<lfr-widget-nav />`. You could embed it in a block like this:

```
<div class="nav-widget">
    <lfr-widget-nav>
    </lfr-widget-nav>
</div>
```

Implement this in your Fragment:

1. Go to the *Fragments* page.
2. Go to the *Developing Fragments* collection and add a new Fragment inside of it named *Widget Fragment*.
3. Insert the following code in the main `<div>` in the HTML pane:

    ```
    <div class="container-fluid">
      <div class="row">
        <div class="col-md-10">
           <lfr-widget-nav>
           </lfr-widget-nav>
        </div>
      </div>
    </div>
    ```

4. Click *Publish*.

Now you need to add it to a Content Page to display it.

1. Go to *Navigation → Site Pages*.
2. Select the *Page Templates* tab at the top.
3. Go to the *Templates for Developing Fragments Collection*.
4. Create a new Page Template named *Widget Page Template*.
5. In the new Page Template, from the menu on the right, add your fragment to the page.
6. Exit the template editor and it saves automatically.
7. Go to the *Pages* tab.
8. Click the + icon to create a new page.
9. Select the *Widget Page Template* and save.
10. Now go back to your site, and select your new page.

For the full list of widgets that can be imbedded, see the Embedding Widgets in Page Fragments reference article.

Embedding Your Widget in a Fragment

If you have a custom widget that you want to embed in a fragment, you can configure that widget to be embeddable. In order to embed your widget, it must be an OSGi Component. Inside the `@Component` annotation for the portlet class that you want to embed, add this property:

```
"com.liferay.fragment.entry.processor.portlet.alias=app-name"
```

When you deploy your widget, it's available to add. The name you specify in the property must be appended to the `lfr-widget` tag like this:

```
<lfr-widget-app-name>
</lfr-widget-app-name>
```

NOTE: According to the W3C HTML standards, custom elements cannot be self closing. Therefore, even though you cannot add anything between the opening and closing `<lfr-widget...>` tags, you cannot use the self closing notation for the tag.

Embedding widgets in Fragments opens a world of options. Now that you've explored some of the power of Fragments, next you'll learn about best practices for development.

114.3 Recommendations and Best Practices

In general all your code should be semantic and highly reusable. A main concern is making sure that everything is namespaced properly so it won't interfere with other elements on the page outside of the Fragment.

CSS

While you can write any CSS in a fragment, it's recommended to prefix it with a class specific to the fragment to avoid impacting other fragments. To facilitate this, when creating a new fragment, the HTML includes a `div` with an automatically generated class name and the CSS shows a sample selector using that class. Use it as the basis for all selectors you add.

JavaScript

Avoid adding a lot of JavaScript code, since it isn't easily reusable. Instead, reference external JS libraries.

Developing A Fragment Using Desktop Tools

You can develop a fragment using any preferred desktop tools. Since the Fragment is HTML, CSS, and JavaScript, you could use a text editor or a specialized tool with its own built in previews.

To import a Collection into Liferay, it must be archived in a `.zip` with the contents in the following format:

- `collection.json`: a text file which describes your collection with the format `{"name":"<collection-name>","description":"<collection-description>"}`

    - `[fragment-name]/`: a folder containing all of the files for a single Page Fragment.

        * `fragment.json`: a text file that describes a Page Fragment with the format `{"jsPath":"src/index.js","htmlPath":"src/index.html","cssPath":"src/index.css","name":"<frag name>"}`

        * `src/`: a folder containing the source code files for the fragment.

            - `index.html`: the HTML source for the fragment
            - `index.css`: the CSS source for the fragment
            - `index.js`: the JavaScript source for the fragment

A collection can contain any number of fragments, so you can have lots of subfolders in the collection. This format is the same as what's exported from within Liferay.

Developers can also create fragments to be imported into an existing collection. Put them in a similarly formatted `.zip`, but without the Collection information at the top level:

- `[fragment-name]/`: a folder containing all the files for a single Page Fragment.
- `fragment.json`: a text file that describes a Page Fragment with the format `{"jsPath":"src/index.js","htmlPath":"src/index.html","cssPath":"src/index.css","name":"<frag name>"}`
- `src/`: a folder containing the source code files for the fragment.

    - `index.html`: the HTML source for the fragment
    - `index.css`: the CSS source for the fragment
    - `index.js`: the JavaScript source for the fragment

Multiple fragments can be included in a single `.zip` file with each having its own folder at the top level.

CHAPTER 115

SCREEN NAVIGATION FRAMEWORK

The Screen Navigation Framework is for customizing and extending application UIs. You can use it to make Liferay's applications your own and to make your applications customizable by others.

To make this work, the framework assumes a specific structure for screens and supports one or two levels of navigation. Each item in the top level navigation is a `ScreenNavigationCategory`. Each item in the second level is a `ScreenNavigationEntry`. Categories are usually represented by tabs, while entries use a second level of navigation. You need not have any Entries in your application, but you must have at least one Category.

The Screen structure normally renders Navigation Categories as horizontal tabs at the top of the page and Navigation Entries as a vertical list of items along the left side of the page. The screen box containing the content uses the rest of the screen. You can customize this default layout for your needs.

Secondary navigation is optional; you may only need category tabs with no additional navigation.

115.1 Using the Framework for Your Application

The Screen Navigation Framework comprises two parts: Java classes for your screens and a tag library for your front-end. First you'll create the necessary Java classes and then add the front-end support through JSPs.

Adding Screens to Your Application's Back-end

You must create at least one Navigation Category. To add screens to your application, first you must add at least one Navigation Category for the top level navigation. Then you can add additional Navigation Entries for each page that you need.

First, add a Navigation Category

1. Create a component that implements the `ScreenNavigationCategory` interface.
2. Implement the following methods in your component:

   `getCategoryKey()`: returns the category's primary key.

   `getLabel(Locale locale)`: returns the label of the key.

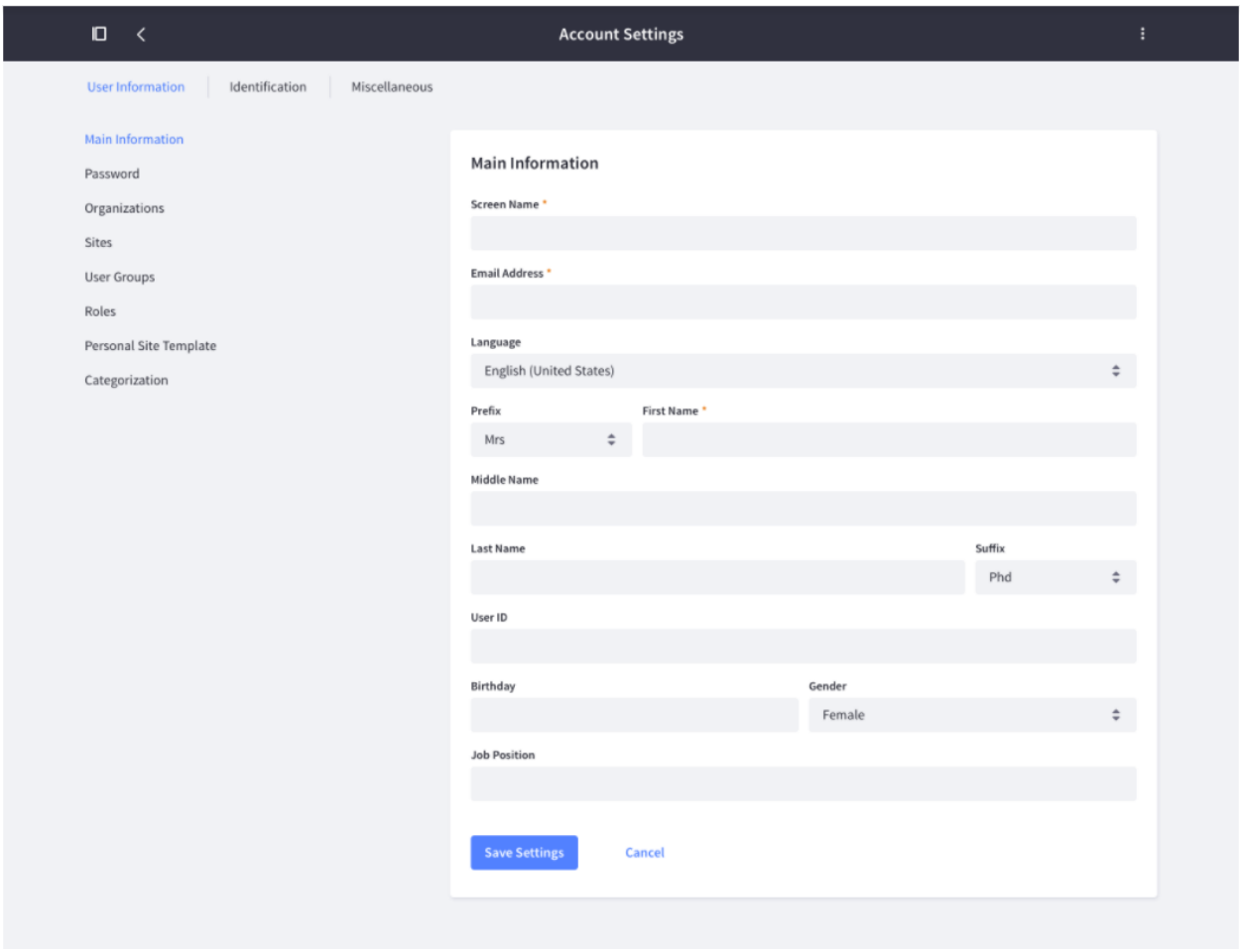


Figure 115.1: A typical application using screen navigation has three categories and numerous entries.

`getScreenNavigationKey()`: returns the navigation key that the category belongs in, as defined in your application.

Next, add a Navigation Entry.

1. Create a component which implements `ScreenNavigationEntry`.
2. Implement the following methods in your component:

   `getCategoryKey()`: returns the category's primary key.

   `getEntryKey()`: returns the entry's primary key.

   `getLabel()`: returns the entries label.

   `getScreenNavigationKey()`: returns the navigation key for the category of the current entry.

   `isVisible(User user, T screenModelBean)`: boolean for whether or not the entry should be visible for the current user.

   `render(HttpServletRequest request, HttpServletResponse response)`: renders the entry.

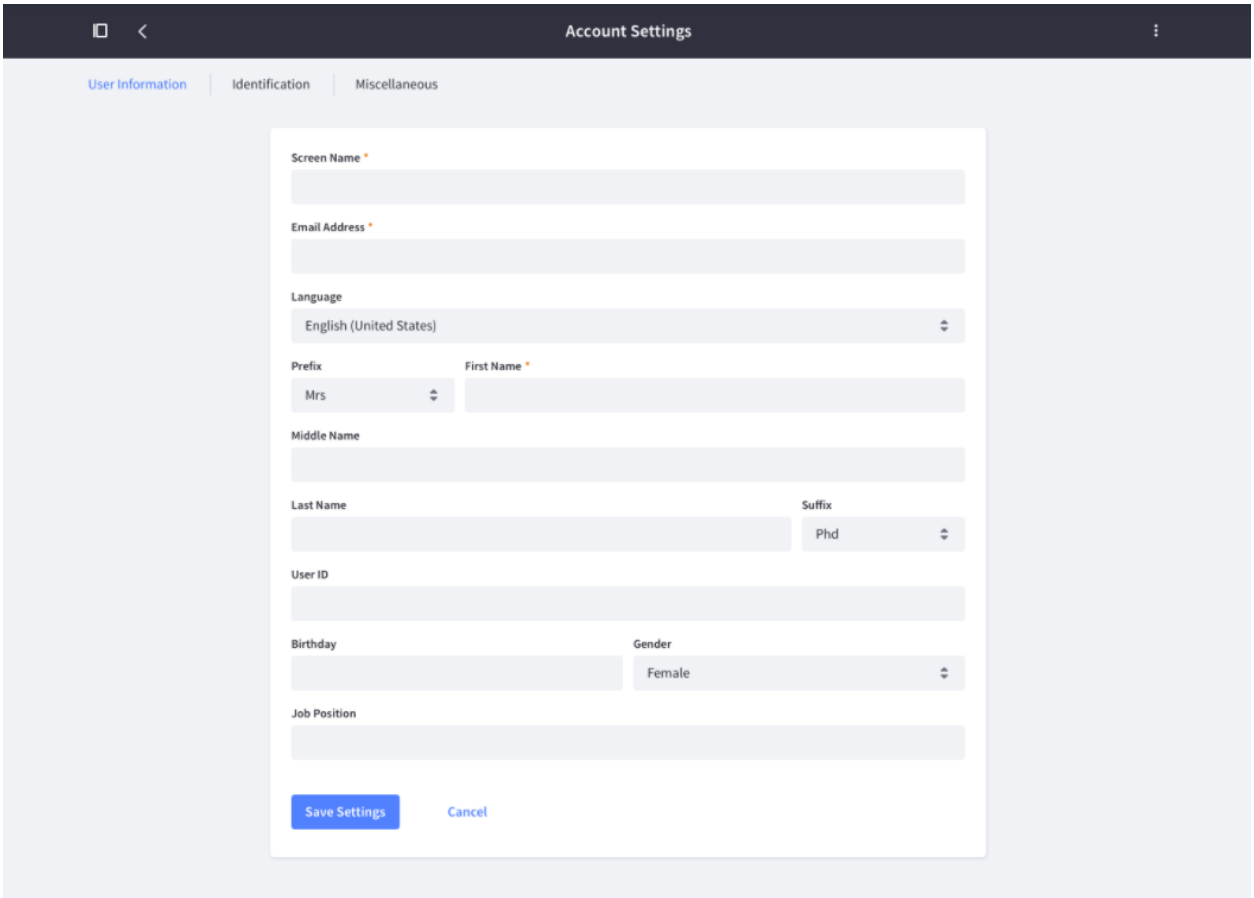


Figure 115.2: Secondary navigation is optional; you may opt to have only tabs.

You can implement your render method any way that you want as long as it provides a way to render HTML. Liferay developers typically use JSPs, shown below.

Adding Screens to Your Application's Front-end

To use JSPs to render your screens, you must invoke the `JSPRenderer` component in your `render` method and create the JSP that renders the HTML.

1. Create a `render` method which uses `JSPRenderer` like this:

```
@Override
public void render(HttpServletRequest request, HttpServletResponse response)
throws IOException {

    _jspRenderer.renderJSP(request, response, "/my-category/view-category.jsp");
}
```

2. Add the following code at the bottom of your class to use the reference annotation to access the `JSPRenderer`:

```
@Reference
private JSPRenderer _jspRenderer;
```

3. Create a JSP that includes the `liferay-frontend:screen-navigation` taglib and the necessary parameters like this:

```
<liferay-frontend:screen-navigation key=
"<%= AssetCategoriesConstants.CATEGORY_KEY_GENERAL %>"
modelBean="<%= category %>"
portletURL="<%= portletURL %>"
/>
```

The parameters it needs are `key`, `modelBean`, and `portletURL`.

- **Key**: a unique name for the navigation in this application.
- **modelBean**: the model that is being rendered
- **portletURL**: the portlet URL used to build the titles for each link.

In the next section, you'll see how to extend an existing Liferay class with more screens.

115.2 Adding Custom Screens to Liferay Applications

You can extend certain Liferay Applications with custom screens. Custom screens can add configuration for features you've developed, integrating them seamlessly with the original application.

Categories Administration

The Categories Administration application supports adding Custom Screens to provide additional options for editing a category. To demonstrate adding a new Screen Navigation Entry and Category, you'll add one to Categories Administration.

1. Create a new Java class in the `asset-categories-admin-web` module named `CategoryCustomScreenNavigationEntry` that implements `ScreenNavigationCategory` and `ScreenNavigationEntry`.
2. Add the following Component annotation above the class declaration:

```
@Component(
    property = {
      "screen.navigation.category.order:Integer=1",
      "screen.navigation.entry.order:Integer=1"
    },
    service = {ScreenNavigationCategory.class, ScreenNavigationEntry.class}
)
```

   The `screen.navigation.category.order` and `screen.navigation.entry.order` determine where in the navigation the items appear. Higher is first in the navigation.

   In the `service` declaration, declare it as defining a `ScreenNavigationCategory`, `ScreenNavigationEntry`, or both.

3. For the class body, insert this code:

```
@Override
public String getCategoryKey() {
    return "custom-screen";
}

@Override
public String getEntryKey() {
    return "custom-screen";
}

@Override
public String getLabel(Locale locale) {
    return LanguageUtil.get(locale, "custom-screen");
}

@Override
public String getScreenNavigationKey() {
    return AssetCategoriesConstants.CATEGORY_KEY_GENERAL;
}

@Override
public void render(HttpServletRequest request, HttpServletResponse response)
    throws IOException {

   _jspRenderer.renderJSP(request, response, "/category/custom-screen.jsp");
}

@Reference
private JSPRenderer _jspRenderer;
```

4. Create a `custom-screen.jsp` in the `/resources/META-INF/resources/category/` folder.
5. At the top of your JSP class, insert the following scriplet at the top to use the Screen Navigation UI:

```
<%
String redirect = ParamUtil.getString(request, "redirect", assetCategoriesDisplayContext.getEditCategoryRedirect());

long categoryId = ParamUtil.getLong(request, "categoryId");

AssetCategory category = AssetCategoryLocalServiceUtil.fetchCategory(categoryId);

long parentCategoryId = BeanParamUtil.getLong(category, request, "parentCategoryId");

long vocabularyId = ParamUtil.getLong(request, "vocabularyId");

portletDisplay.setShowBackIcon(true);
portletDisplay.setURLBack(redirect);

renderResponse.setTitle(((category == null) ? LanguageUtil.get(request, "add-new-category") : category.getTitle(locale)));
%>
```

6. Below that insert the following tag:

```
<liferay-frontend:screen-navigation key=
"<%= AssetCategoriesConstants.CATEGORY_KEY_GENERAL %>"
modelBean="<%= category %>"
portletURL="<%= portletURL %>"
/>
```

7. For the rest of the JSP, create your custom screen.

Now you can use that pattern to create additional screens for whatever you need.

CHAPTER 116

PRODUCT NAVIGATION

Liferay's product navigation consists of the main menus you use to customize, configure, and navigate the system. When you edit a page, switch to a different Site scope, access a User's credentials, etc., you're using the default navigation menus. Providing a customization to a default menu can help give your Liferay instance a unique touch. You can extend and customize the default product navigation to fit your need.

There are four product navigation sections that you can extend: Product Menu, Control Menu, Simulation Menu, and User Personal Menu.

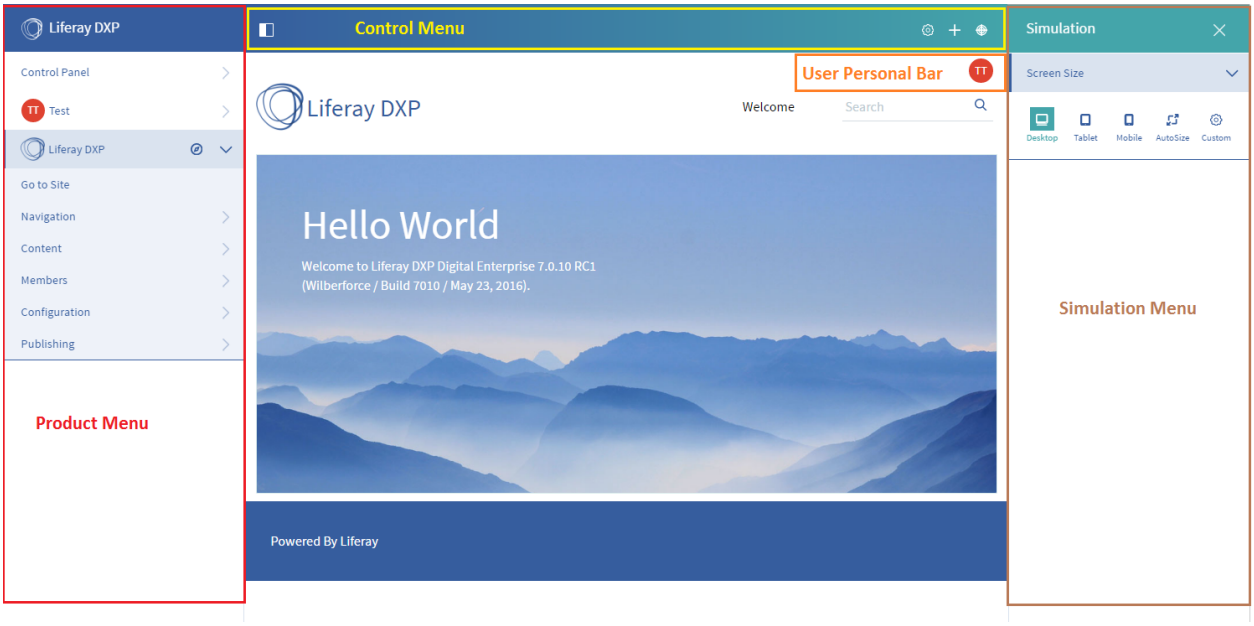


Figure 116.1: The main product navigation menus include the Product Menu, Control Menu, and Simulation Menu.

The Product Menu is on the left, and displays the Control Panel, User account settings, and Site Administration functionality. The Control Menu is on top, offering navigation to the Product Menu, Simulation Menu (the right menu), and the *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.). Finally, the User Personal Menu holds selectable items containing 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.

CHAPTER 117

CUSTOMIZING THE PRODUCT MENU

By default, Liferay's Product Menu consists of three main sections: Control Panel, User Menu, and Site Administration. These sections are called panel categories. For instance, the Control Panel is a single panel category, and when clicking on it, you see four child panel categories: *Users*, *Sites*, *Apps*, and *Configuration*. Clicking a child panel category shows 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.

The Product Menu is intuitive and easy to use—but you can still change it any way you want. You can reorganize the panel categories and apps, or add completely new categories and populate them with custom panel apps. Here you'll learn how to provide new or modified panel categories and panel apps for the Product Menu.

117.1 Adding Custom Panel Categories

As you navigate the Product Menu, you can see that Panel Apps like *Web Content* and *Site Settings* are organized into Panel Categories such as *Content* and *Configuration*. This tutorial explains how to add new Panel Categories to the menu. Adding new Panel Apps is covered in the next section.

There are three steps to creating a new category:

1. Create the OSGi structure and metadata.
2. Implement Liferay's Frameworks.
3. Define the Control Menu Category.

Creating the OSGi Module

First you must create the project.

1. Create an OSGi module using your favorite third party tool, or use Blade CLI. Blade CLI offers a Panel App template, which is for creating a 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`).

Implementing Liferay's Frameworks

Next, you must connect your OSGi module to Liferay's frameworks and use those to define information about your entry. This takes only two steps:

1. Insert the `@Component` annotation declaring the panel category keys.
2. Implement the `PanelCategory` interface.

Both of these steps are described below.

Insert the @Component Annotation

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.

Finally, 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
)
```

Note: To insert a panel category between existing categories in the default menu, you must know the `panel.category.order:Integer` property for the existing categories. Default categories with a given panel.category.key are numbered in increments of 100, starting with 100.

For example, the Product Menu's three main sections—Control Panel, User Menu, and Site Administration—have `panel.category.order:Integer` properties of 100, 200, and 300, respectively.

A new panel inserted between Control Panel and User Menu would need a `panel.category.key` of ROOT and a `panel.category.order:Integer` of 150.

Implement the PanelCategory Interface

The `PanelCategory` interface requires you to implement the following methods:

- `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 saves the state of the panel category when navigating away from the menu.

You can reduce the number of methods you must implement if you extend a base class that already implements the `PanelCategory` interface. The recommended 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. To add more complex functionality, you can then provide a custom UI for your panel using any front-end technology by implementing the `include()` or `includeHeader()` from the `PanelCategory` interface.

If you plan to use JSPs as the front-end technology, extend a base class called `BaseJSPPanelCategory` that already implements the methods `include()` and `includeHeader()` for you. This is covered in more detail below.

Note: In this tutorial, example JSPs describe how to provide functionality to panel categories and apps. JSPs, however, are not the only way to provide front-end functionality to your categories/apps. You can create your own class implementing `PanelCategory` to use other technologies such as FreeMarker.

Defining the Control Menu Category

After establishing the framework you're using to create the category, you must 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`.

BasePanelCategory

If you need 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");
}
```

BaseJSPPanelCategory

If you need more complex functionality, extend `BaseJSPPanelCategory` and use JSPs to render the panel category. For example, the `SiteAdministrationPanelCategory` specifies the `getHeaderJspPath` and `getJspPath` methods. You could create a JSP with the UI you want 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 renders the panel category's header (displayed when panel is collapsed) and the other its body (displayed when panel is expanded).

You must also 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
```

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.

117.2 Adding Custom Panel Apps

After you have created a Category, create a Panel app to go in it:

1. Create an OSGi module using your favorite third party tool, or use Blade CLI. Blade CLI offers a Panel App template to help 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 similar to those discussed in the previous tutorial. The `panel.category.key` assigns your panel app to a panel category. The `panel.app.order:Integer` property specifies the order your panel app appears among other panel apps in the same category. For example, if you want to add a panel app to Site Administration → *Content*, 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.

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 by extending the BasePanelApp abstract class. As you learned in the previous tutorial on panel categories, if you must create a more complex UI to render in the panel, you can.

   If you want to use JSPs to render that UI, extend 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 front-end functionality to your panel apps. You can create your own class implementing `PanelCategory` to use other technologies such as FreeMarker.

5. If you are implementing the PanelApp interface without extending a base class, you must implement its methods. The `BlogsPanelApp` is a simple example of how to specify your portlet as a panel app. This class extends `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 portlet must be created. By default, the panel app only appears 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);
}
```

You can also 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.

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.

CHAPTER 118

CUSTOMIZING THE CONTROL MENU

The Control Menu is the most visible and accessible menu in Liferay. 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 118.1: The Control Menu has three configurable areas: left, right, and middle. It also displays the title and type of page that you are currently viewing.

If you navigate away from the home page, the Control Menu adapts and provides helpful functionality for whatever option you're using. For example, if you navigate to Site Administration → *Content* → *Web Content*, you see a Control Menu with different functionality tailored for that option.

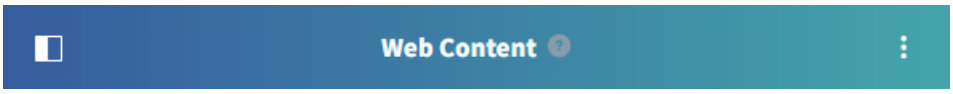


Figure 118.2: When switching your context to web content, the Control Menu adapts to provide helpful options for that area.

The default Control Menu contains three categories representing the left, middle, and right portions of the menu. You can create navigation entries for each category.

Note: You can add the Control Menu to a 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 embeds 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.

Next you'll learn how to customize the Control Menu.

118.1 Creating Control Menu Entries

Now you'll create entries to customize the Control Menu. Make sure to read Adding Custom Panel Categories before beginning this tutorial. This tutorial assumes you know how to create a panel category. Creating a Control Menu Entry follows the same pattern as creating a category:

1. Create the OSGi structure and metadata.
2. Implement Liferay's Frameworks.
3. Define the Control Menu Entry.

Creating the OSGi Module

First you must create the project.

1. Create a generic OSGi module. 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. Give your class a unique name followed by *ProductNavigationControlMenuEntry* (e.g.,`StagingProductNavigationControlMenuEntry`).

Implementing Liferay's Frameworks

Next, you need to connect your OSGi module to Liferay's frameworks and use those to define information about your entry.

1. Directly above the class's declaration, insert this code:

```
@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 your entry's category. The default Control Menu provides three categories: Sites (left portion), Tools (middle portion), and User (right portion).

Figure 118.3: 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, this property places your entry in the middle portion of the Control Menu:

```
"product.navigation.control.menu.category.key=" + ProductNavigationControlMenuCategoryKeys.TOOLS
```

Like panel categories, you must specify an integer to place your entry in the category. Entries are ordered from left to right: an entry with order `1` appears 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. Finally, your `service` element should specify the `ProductNavigationControlMenuEntry.class` service.

2. 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 want to provide a more complex UI with buttons or a sub-menu, you can override the `include()` and `includeBody()` methods. If you use JSPs for generating the UI, you can extend `BaseJSPProductNavigationControlMenuEntry` to save time. This is covered in more detail below.

Defining the Control Menu Entry

Now you must define your Control Menu Entry. Here are some examples for defining your entry.

Control Menu Examples

The `IndexingProductNavigationControlMenuEntry` is a simple example for providing text and an icon. It extends the `BaseProductNavigationControlMenuEntry` class and is used when Liferay is indexing. 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 `ProductMenuProductNavigationControlMenuEntry` is more sophisticated. This entry appears 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 use JSPs to define your entry's UI. There are two methods to notice:

```
@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 test this by opening and closing the Product Menu on the home page.

Finally, if you provide functionality that is exclusively inside the Control Menu, the `StagingProductNavigationControlMenuEntry` class calls its JSP like this:

```
@Override
public String getIconJspPath() {
    return "/control_menu/entry.jsp";
}
```

The `entry.jsp` is returned, which embeds the Staging Bar portlet into the Control Menu.

You must also specify the servlet context for the JSP files. If this is inside an OSGi module, make sure your `bnd.bnd` file defines 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:

```
@Override
@Reference(
    target = "(osgi.web.symbolicname=com.sample.my.module.web)",
    unbind = "-"
)
public void setServletContext(ServletContext servletContext) {
    super.setServletContext(servletContext);
}
```

Displaying Your Control Menu Entry

Part of creating the entry is defining when it appears. The Control Menu shows different entries depending on the displayed page. You can specify when your entry appears with the `isShow(HttpServletRequest)` method.

For example, the `IndexingProductNavigationControlMenuEntry` class queries the number of indexing jobs when calling `isShow`. If the query count is `0`, the indexing entry doesn't appear 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 selects the pages to appear. The staging entry never appears 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;
}
```

Defining Dependencies

Define dependencies for your Control Menu Entry 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.

118.2 Defining Icons and Tooltips

When creating a Control Menu entry, you can use an icon in addition to or in place of text. You can also use tooltips to provide a more in depth explanation.

Control Menu Entry Icons

You can provide a Lexicon or CSS icon in your `*ControlMenuEntry`. To use a Lexicon icon, you should override the methods in `ProductMenuProductNavigationControlMenuEntry` like this one:

```
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 this one:

```
public String getIconCssClass(HttpServletRequest request) {
    return "icon-css";
}

public String getIcon(HttpServletRequest request) {
    return "";
}

public String getMarkupView(HttpServletRequest request) {
    return "";
}
```

You can find these icons in the icons-lexicon and icons-font-awesome components, respectively.

Control Menu Entry Tooltips

To provide a tooltip for the Control Menu entry, create a `getLabel` method like this:

```
@Override
public String getLabel(Locale locale) {
    ResourceBundle resourceBundle = ResourceBundleUtil.getBundle(
        "content.Language", locale, getClass());

    return LanguageUtil.get(
        resourceBundle, "the-portal-is-currently-reindexing");
}
```

You need to create a `Language.properties` to store your labels. You can learn more about resource bundles in the Internationalization tutorials.

118.3 Extending the Simulation Menu

When testing how pages and apps appear for users, it's important to simulate their views in as many ways as possible. The Simulation Menu on the right-side of the main page allows this, and you can extend the menu if you need to simulate something that it does not provide.

First, you must get accustomed to using panel categories/apps. This is covered in detail in the Customizing The Product Menu tutorial. Once you know how to create panel categories and panel apps, continue with 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, a hidden category needed to hold the `DevicePreviewPanelApp`. This is the app and functionality you see in the Simulation Menu by default.

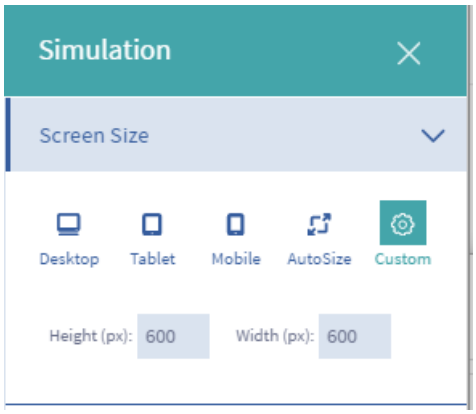


Figure 118.4: The Simulation Menu offers a device preview application.

To provide your own functionality in the Simulation Menu, you must create a panel app in `SimulationPanelCategory`. If you want to add extensive functionality, you can even create additional panel categories in the menu to divide up your panel apps. This tutorial covers the simpler case of creating a panel app for the already present hidden category.

1. Follow the steps documented in Adding Custom Panel Apps 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 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 must add a dependency on `com.liferay.product.navigation.simulation`.

   Be sure to also specify the order to display your new panel app, which was explained in Adding Custom Panel Apps.

3. This tutorial assumes you're using JSPs. Therefore, you should extend the `BaseJSPPanelApp` abstract class, which 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 front-end technology you want, if you want 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 JavaScript resources in your simulation module.

   The right servlet context is also provided by 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 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 similar to `DevicePreviewPanelApp`, except it points to a different portlet and JSP.

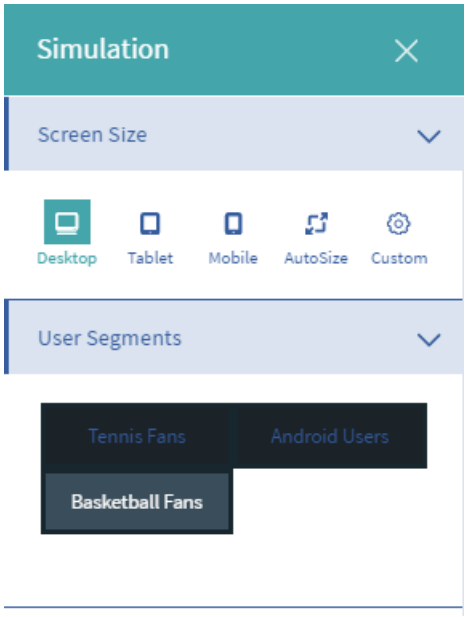


Figure 118.5: The Audience Targeting app extends the Simulation Menu to help simulate different users and campaign views.

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 this code:

   ```
   var iframe = A.one('#simulationDeviceIframe');
   ```

   Then you can modify the device preview frame URL like this:

   ```
   iframe.setAttribute('src', newUrlWithCustomParameters);
   ```

Now that you know how to extend the necessary panel categories and panel apps to modify the Simulation Menu, create a module of your own and customize the Simulation Menu so it's most helpful for your needs.

118.4 Providing the User Personal Bar

The User Personal Bar displays options unique to the current user. By default, this menu appears as an avatar button that expands the User Settings sub-menu in the Product Menu. In a custom theme, the User Personal Bar could appear anywhere in the interface.

Although Liferay's default User Personal Bar is bare-bones, you can add more functionality 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.

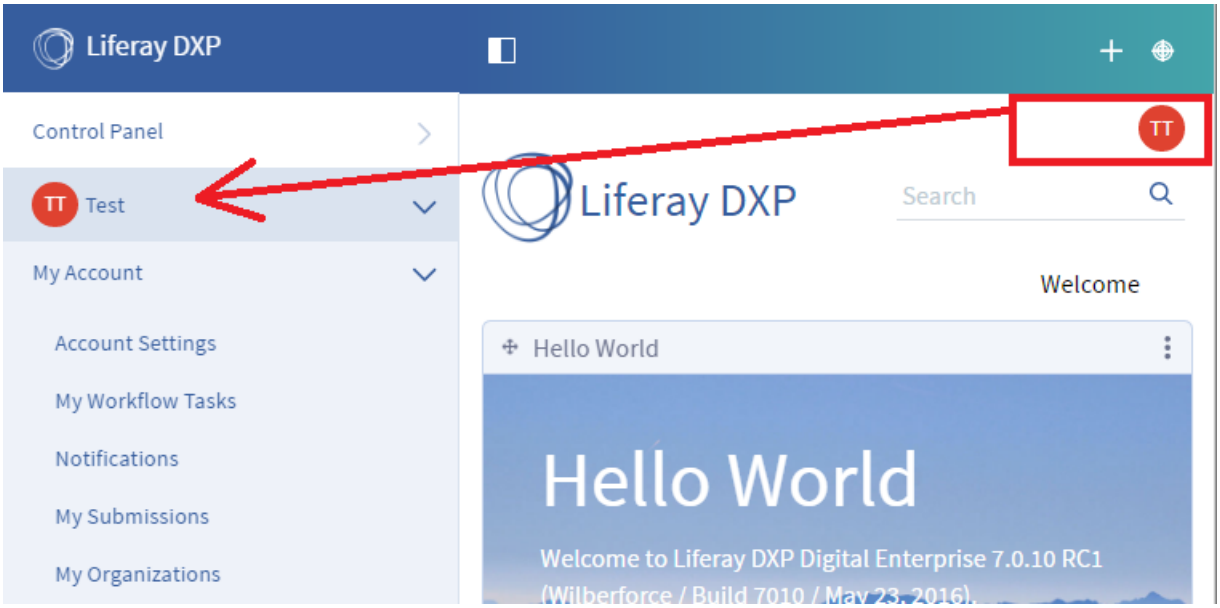


Figure 118.6: By default, the User Personal Menu contains the signed-in user's avatar, which navigates to the Product Menu when selected.

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 replaced by other portlets.

Note: You can add the User Personal Bar to your 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.

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. Above the class 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 it can be displayed in the User Personal Bar.

You should also specify the service rank for your new portlet so it overrides the default. 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 only want the User Personal Bar to display your portlet, the `service` element should 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 want in the User Personal Bar by declaring the following method in your class:

```
@Override
public String getPortletName() {
    return PORTLET_NAME;
}
```

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_bar_web_portlet_ProductNavigationPersonalBarPortlet` for its default User Personal Bar portlet.

You've successfully provided a portlet to be displayed in the User Personal Bar. If you want 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 119

COLLABORATION

The collaboration suite helps users interact and create content together. This can be as simple as a quick conversation in the Message Boards app or as complex as joint file management via the Documents and Media Library. Users can Blog their experiences and share knowledge with the Wiki. Comments can be posted on all content. Integrated applications mean the whole is greater than the sum of the parts: together, users create something of value that couldn't exist if they were working in isolation.

Underlying the collaboration suite is a set of powerful APIs so you can leverage these features in your own apps. For example, if your app lets users create a custom content type, you can tie into the collaboration suite's social API to let users comment on and rate that content. You can also customize how your app's users select items from the Documents and Media Library, send custom alerts and announcements, and much, much more. The tutorials in this section show you how.

CHAPTER 120

ITEM SELECTOR

An *Item Selector* is a UI component for selecting entities in a user-friendly manner. Many Liferay apps use Item Selectors to let users select items such as images, videos, audio files, documents, and pages. For example, the Documents and Media Library lets users select files.

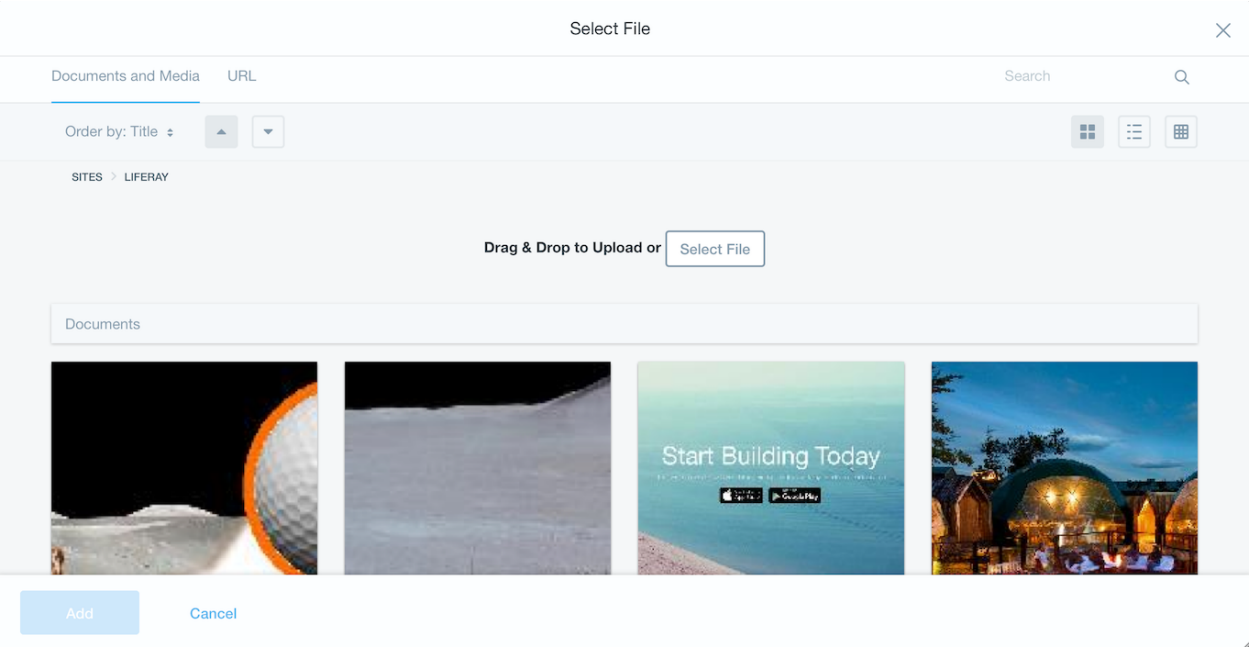


Figure 120.1: Item Selectors let users browse and select different kinds of entities.

The Item Selector API provides a framework for you to use, extend, and create Item Selectors in your own apps. The tutorials in this section show you how to use this framework.

Here are some use cases for the Item Selector API:

1. Letting your app's users select entities via an Item Selector.
2. Configuring an Item Selector to select your app's custom entity.

3. Customizing the selection experience by adding a new *selection view* for an entity.

You'll learn what a selection view is, along with the Item Selector API's other components, in the next tutorial.

120.1 Understanding the Item Selector API's Components

Before working with the Item Selector API, you should learn about its components. You'll work with these components as you leverage the API in your apps:

- **Selection views:** These are the framework's key components. They show entities of particular types from different sources. For example, an Item Selector configured to show images might show selection views from Documents and Media, a third-party image provider, or a drag-and-drop UI.

- **Markup:** A markup file that renders the selection view. You have a great deal of flexibility in the markup language you choose. For example, you can use a JSP, FreeMarker, or even pure HTML and JavaScript.

- **Return Type:** A class that represents the type of information returned from the entities selected by the users. For example, if users select images and you want to return the selected image's URL, then you need a URL return type class. Each return type class must implement the `ItemSelectorReturnType` interface. Such classes are named after the data they return and suffixed with `ItemSelectorReturnType`. For example, the URL return type class is `URLItemSelectorReturnType`.

- **Criterion:** A class that represents the entity selected by the users. For example, if users select images then you need an image criterion class. Each criterion class must implement the `ItemSelectorCriterion` interface. Criterion classes are named for the entity they represent and suffixed with `ItemSelectorCriterion`. For example, the criterion class for images is `ImageItemSelectorCriterion`.

- **Criterion Handler:** A class that gets the appropriate selection view. Each criterion requires a criterion handler. Criterion handler classes extend the `BaseItemSelectorCriterionHandler` class with the criterion's entity as a type argument. Criterion handler classes are named after the criterion's entity and suffixed by `ItemSelectorCriterionHandler`. For example, the image criterion handler class is `ImageItemSelectorCriterionHandler`. It extends `BaseItemSelectorCriterionHandler<ImageItemSelectorCriterion>`.

This diagram shows how these components interact to form a working API.

Related Topics

Selecting Entities Using an Item Selector
    Creating Custom Item Selector Entities
    Creating Custom Item Selector Views

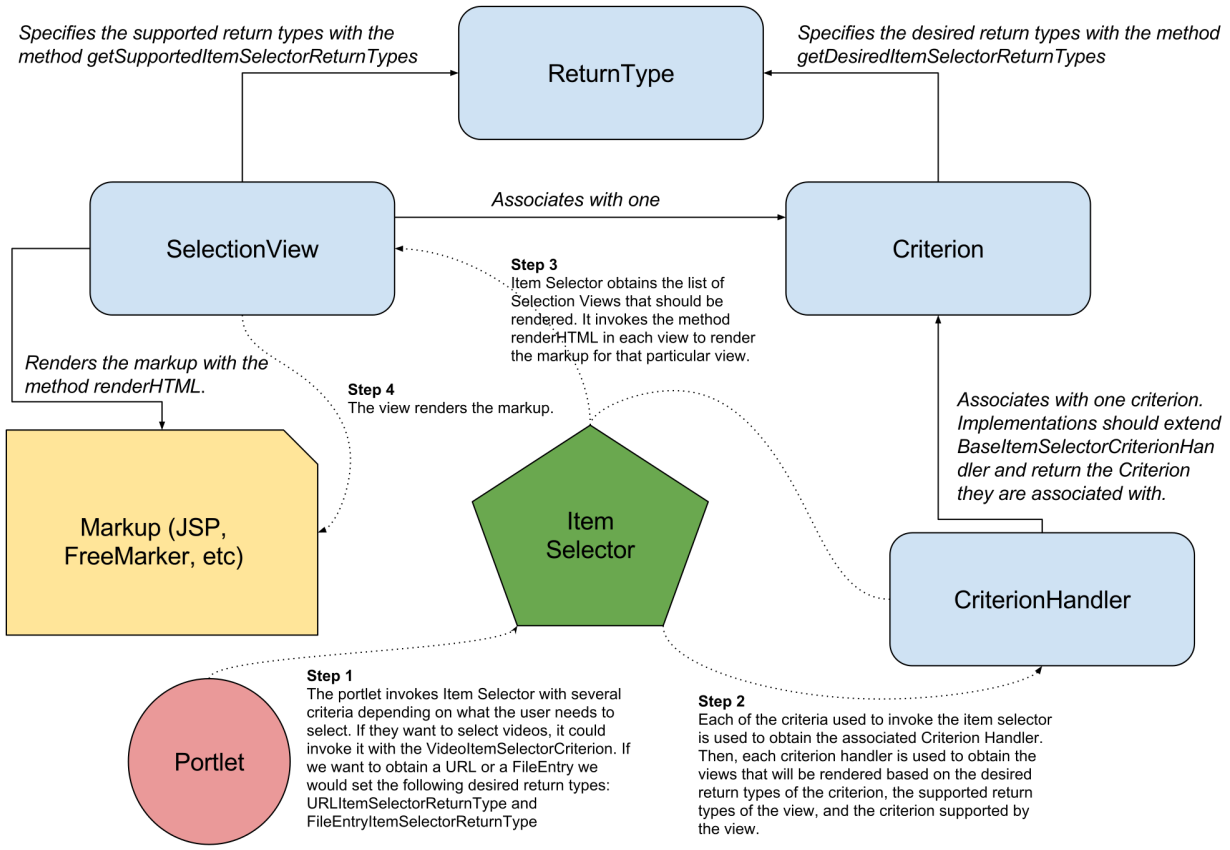


Figure 120.2: Item Selector views (selection views) are determined by the return type and criterion, and rendered by the markup.

120.2 Selecting Entities Using an Item Selector

An Item Selector lets users select entities such as images, videos, documents, and sites. You can use an Item Selector in your app to let users select such entities. This tutorial shows you how to do this via these steps:

1. **Determine the Criteria for an Item Selector:** You must first define which entities an Item Selector can let users select.

2. **Get an Item Selector for Your Criteria:** If your criteria is for images, for example, then in this step you'll get an Item Selector capable of selecting images.

3. **Use an Item Selector Dialog:** Display the Item Selector in your UI.

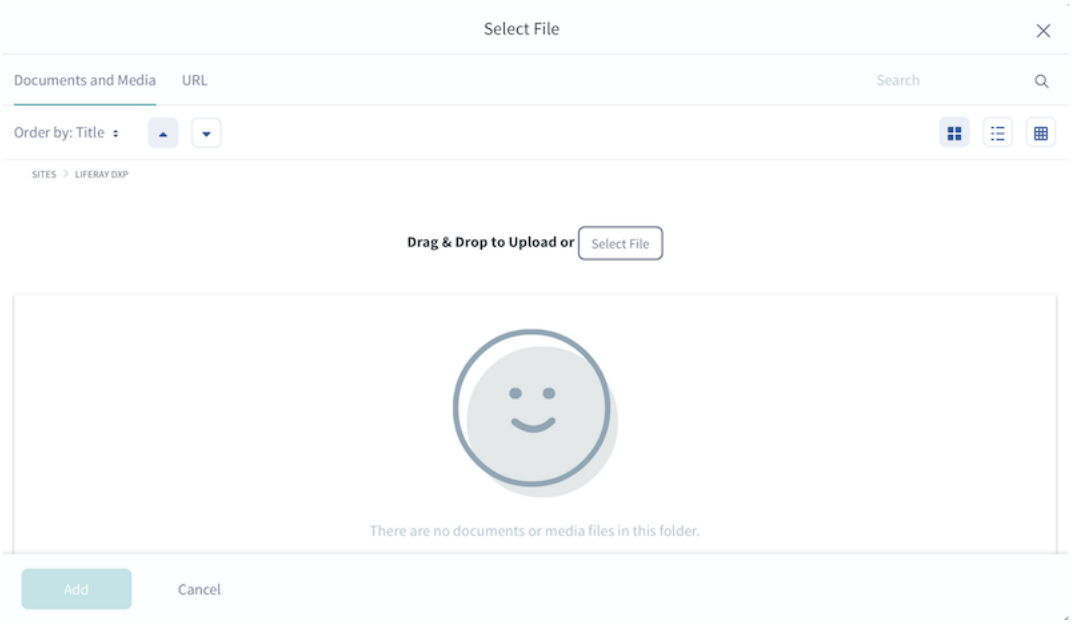


Figure 120.3: An Item Selector makes selecting entities a breeze.

Determining Item 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.

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 DXP's apps and app suites provide.

If there's no return type class that meets your needs, you can implement your own `ItemSelectorReturnType` class.

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

To use an Item Selector with your criteria, you must get that Item Selector's URL. The URL is needed to open the Item Selector dialog in your UI. To get this URL, you must get an `ItemSelector` reference and call its `getItemSelectorURL` method with the following parameters:

- `RequestBackedPortletURLFactory`: A factory that creates portlet URLs.
- `ItemSelectedEventName`: A unique, arbitrary JavaScript event name that the Item Selector triggers when the element is selected.
- `ItemSelectorCriterion`: The 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:

1. Use Declarative Services to get an `ItemSelector` OSGi Service Component:

   ```
   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.

2. Create the factory you'll use to create the URL. To do this, invoke the `RequestBackedPortletURLFactoryUtil.creat` method with the current request object. The request can be an `HttpServletRequest` or `PortletRequest`:

   ```
   RequestBackedPortletURLFactory requestBackedPortletURLFactory =
       RequestBackedPortletURLFactoryUtil.create(request);
   ```

3. Create a list of return types expected for the image entity. The return types list consists of a URL return type `URLItemSelectorReturnType`:

   ```
   List<ItemSelectorReturnType> desiredItemSelectorReturnTypes =
       new ArrayList<>();
   desiredItemSelectorReturnTypes.add(new URLItemSelectorReturnType());
   ```

4. Create a criterion object for images (`ImageItemSelectorCriterion`):

   ```
   ImageItemSelectorCriterion imageItemSelectorCriterion =
       new ImageItemSelectorCriterion();
   ```

5. Use the criterion's `setDesiredItemSelectorReturnTypes` method to set the return types list from step 3 to the criterion:

   ```
   imageItemSelectorCriterion.setDesiredItemSelectorReturnTypes(
       desiredItemSelectorReturnTypes);
   ```

6. Call the Item Selector's `getItemSelectorURL` method to get an Item Selector URL based on the criterion. The method requires the URL factory, an arbitrary event name, and a series of criterion (one, in this case):

```
PortletURL itemSelectorURL = _itemSelector.getItemSelectorURL(
    requestBackedPortletURLFactory, "sampleTestSelectItem",
    imageItemSelectorCriterion);
```

There are a few things to keep in mind when getting an Item Selector's URL:

- You can invoke the URL object's `toString` method to get its value.
- You can configure an Item Selector to use any number of criterion. The criterion can use any number of return types.
- The order of the Item Selector's criteria determines the selection view order. For example, if you pass the Item Selector an `ImageItemSelectorCriterion` followed by a `VideoItemSelectorCriterion`, the Item Selector displays the image selection views first.
- The 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 an Item Selector, you can start using that Item Selector in your UI.

Using the Item Selector Dialog

To open the Item Selector in your UI, you must use the JavaScript component `LiferayItemSelectorDialog` 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. Define the UI element that you'll use to open the Item Selector dialog. For example, this creates a *Choose* button with the ID `chooseImage`:

```
<aui:button name="chooseImage" value="Choose" />
```

3. Get the Item Selector's URL:

```
<%
String itemSelectorURL = GetterUtil.getString(request.getAttribute("itemSelectorURL"));
%>
```

4. Add the `<aui:script>` tag and set it to use the `liferay-item-selector-dialog` module:

```
<aui:script use="liferay-item-selector-dialog">

</aui:script>
```

5. Inside the `<aui:script>` tag, attach an event handler to the UI element you created in step 2. For example, this attaches a click event and a function to the *Choose* button:

```
<aui:script use="liferay-item-selector-dialog">

    $('#<portlet:namespace />chooseImage').on(
    'click',
      function(event) {
        <!-- function logic goes here -->
      }
    );

</aui:script>
```

   Inside the function, you must create a new instance of the `LiferayItemSelectorDialog` AlloyUI component and configure it to use the Item Selector. The next steps walk you through this.

6. Now you must create the function logic. First, create a new instance of the Liferay Item Selector dialog:

```
var itemSelectorDialog = new A.LiferayItemSelectorDialog(
    {
        ...
    }
);
```

7. Inside the braces of the `LiferayItemSelectorDialog` constructor, first set set the `eventName` attribute. This makes the dialog listen for the item selected event. The event name is the the Item Selector's event name that you specified in your Java code (the code that gets the Item Selector URL):

```
eventName: 'ItemSelectedEventName',
```

8. Immediately after the `eventName` setting, set the `on` attribute to implement a function that operates on the selected item change. For example, this function sets its variables for the newly selected item. The information available to parse depends on the return type(s) that were set. As the comment indicates, you must add the logic for using the selected element:

```
on: {
        selectedItemChange: function(event) {
            var selectedItem = event.newVal;

            if (selectedItem) {
                var itemValue = JSON.parse(
                selectedItem.value
                );
                itemSrc = itemValue.url;

                <!-- use item as needed -->
            }
        }
},
```

9. Immediately after the `on` setting, set the `title` attribute to the dialog's title:

```
title: '<liferay-ui:message key="select-image" />',
```

10. Immediately after the `title` setting, set the `url` attribute to the previously retrieved Item Selector URL. This concludes the attribute settings inside the `LiferayItemSelectorDialog` constructor:

```
url: '<%= itemSelectorURL.toString() %>'
```

11. To conclude the logic of the function from step 4, open the Item Selector dialog by calling its `open` method:

```
itemSelectorDialog.open();
```

Here's the complete example code for these steps:

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

<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>
```

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.

Great! Now you know how to select entities using an Item Selector. Using the Item Selector API, you can give your app's users the power of choice!

Related Articles

Understanding the Item Selector API's Components
    Creating Custom Item Selector Views
    Creating Custom Item Selector Entities
    Front-End Taglibs

120.3 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:

1. Create a class that extends the `BaseItemSelectorCriterion` class. This class specifies what kind of entity the user is selecting and what information the Item Selector should return. The methods inherited from `BaseItemSelectorCriterion` provide the logic for obtaining this information.

   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.

   This example 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 your app's needs, 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 reference document Item Selector Criterion and Return Types. 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` interface. You should name such classes after their entity, and suffix them with `ItemSelectorReturnType`. For example, if you were to create a return type for a task item, its return type class would be `TaskItemSelectorReturnType`. Such a `*ItemSelectorReturnType` class is used as an identifier by the Item Selector and does not return any information itself. 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. For example, here's the example return type class `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{

}
```

Nice work! Your new entity's criterion and return type classes can be used by your app to create the Item Selector's URL. To learn how to obtain the Item Selector URL's, see the tutorial Selecting Entities Using the Item Selector.

Once you have the Item Selector's URL, a selection view is responsible for returning the proper entity information specified by the return type. To create such a selection view for your entity, see the tutorial Creating Custom Item Selector Views.

Related Topics

Understanding the Item Selector API's Components
    Selecting Entities using the Item Selector
    Creating Custom Item Selector Views

120.4 Creating Custom Item Selector Views

Item Selector's default selection views may provide everything you need for your app. Custom selection views are required, however, for certain situations. For example, if you want your users to be able to select images from an external image provider, then you must create a custom selection view. You can create a custom selection view by following the steps in this tutorial. Before getting started, you'll learn a bit more about selection views.

Note that the view the Item Selector presents is determined by the type of entity the user is selecting. The Item Selector can also render multiple views for the same entity type. For example, several selection views are available when a user selects an image. Each selection view is a tab in the UI that corresponds to the image's location.

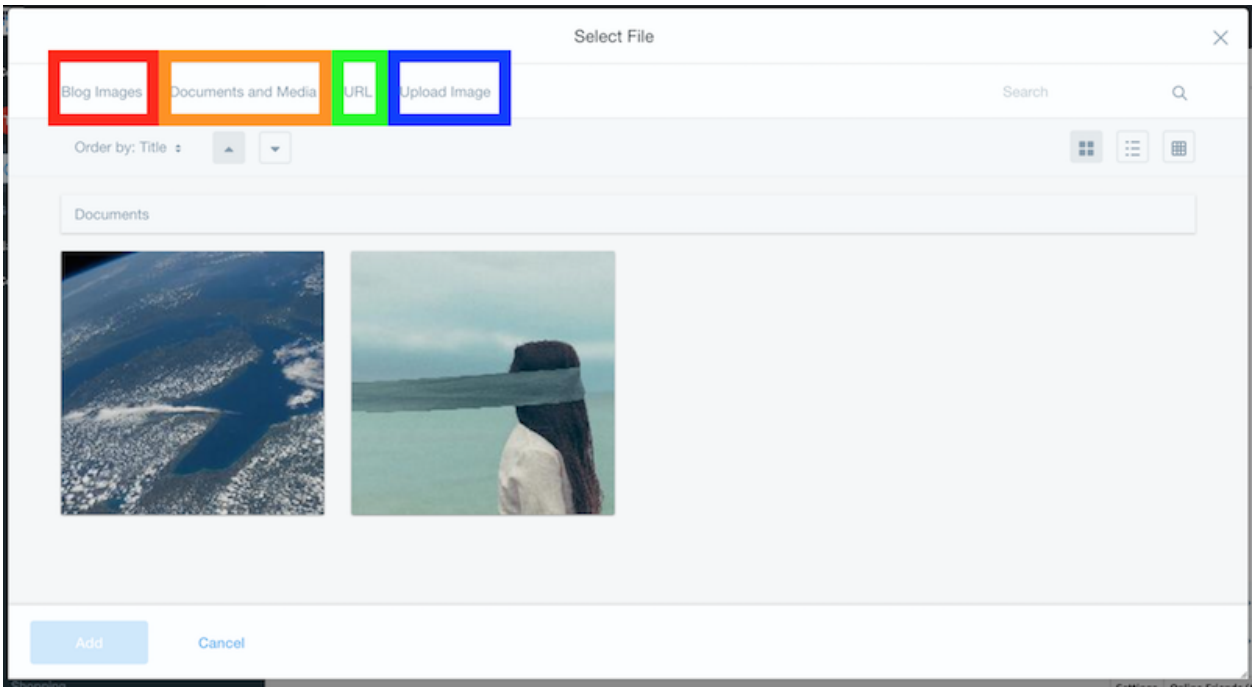


Figure 120.4: An entity type can have multiple selection views.

Each selection view is 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

You'll create a custom selection view by following these steps:

1. Configure your selection view's OSGi module.
2. Implement the selection view's class.
3. Write your selection view's markup.

Configuring Your Selection View's OSGi Module

Follow these steps to configure your selection view's 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, this configuration 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 your module's resources:

```
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 selection view's class.

Implementing Your Selection View's Class

To create a new selection view, you must first know what kind of entities you want it to present (images, videos, users, etc.). This determines the specific `ItemSelectorCriterion` you need to use. For example, a selection view for images must use `ImageItemSelectorCriterion`.

You must also know the entity's return type (the information type you expect from entities when users select them). For example, if a selected entity returns its URL, you would use `URLItemSelectorReturnType` for the return type.

For a full list of the criterion and returns types available in Liferay DXP's apps, see the reference document Item Selector Criterion and Return Types.

Once you've determined these things, follow these steps to create your selection view's class:

1. Create an `ItemSelectorView` component class that implements the `ItemSelectorView` interface. Use the criterion the view requires as a type argument to this interface. In the `@Component` annotation, set the `item.selector.view.order` property to the order you want it to appear in when displayed alongside other selector views of the same criterion. The lower this value is, the higher the view's priority is and the sooner it appears in the order.

   For example, this example selector view class is for images, so it implements `ItemSelectorView` with the `ImageItemSelectorCriterion` class as a type argument. The `@Component` annotation sets the `item.selector.view.order` property to `200` and registers the class as an `ItemSelectorView` service:

   ```
   @Component(
       property = {"item.selector.view.order:Integer=200"},
       service = ItemSelectorView.class
   )
   public class SampleItemSelectorView
       implements ItemSelectorView<ImageItemSelectorCriterion> {...
   ```

   Note that the criteria order can also be specified in the app's `getItemSelectorURL` method.

2. Create getter methods for the criterion class, servlet context, and return types. You'll use these in the steps that follow:

   ```
   @Override
   public Class<ImageItemSelectorCriterion> getItemSelectorCriterionClass()
   {
       return ImageItemSelectorCriterion.class;
   }

   @Override
   public ServletContext getServletContext() {
       return _servletContext;
   }

   @Override
   public List<ItemSelectorReturnType> getSupportedItemSelectorReturnTypes() {
       return _supportedItemSelectorReturnTypes;
   }
   ```

   Note that the `getSupportedItemSelectorReturnTypes` method returns a list of `ItemSelectorReturnTypes`. You'll populate this list in a later step to specify the return types that the selection view supports.

3. Configure the title, search options, and visibility settings for the selection view. You'll do this via these methods:

   - `getTitle`: returns the localized title of the tab to display in the Item Selector dialog.
   - `isShowSearch()`: 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()`](https://docs.liferay.com/dxp/apps/collaboration/latest/javadocs/com/liferay/item/selector/ItemSelectorView.html#isVisible-com.liferay.portal.kernel.theme.ThemeDisplay-):
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.

Here's an example configuration for the `Sample Selector` selection view:

    @Override
    public String getTitle(Locale locale) {
        return "Sample Selector";
    }

    @Override
    public boolean isShowSearch() {
        return false;
    }

    @Override
    public boolean isVisible(ThemeDisplay themeDisplay) {
        return true;
    }
```

4. Use the `renderHTML` method to set the render settings for your view. In addition to the servlet request and response, this method takes the following arguments:

    - `itemSelectorCriterion`: the `*ItemSelectorCriterion` required to display the selection view.
    - `portletURL`: the portlet URL used to invoke the Item Selector.
    - `itemSelectedEventName`: the event name that the caller listens for. When an element is selected, the view fires a JavaScript event with this name.
    - `search`: a search boolean that specifies when the selection view should render search results. When the user performs a search, this boolean should be set to `true`.

    Here's an example implementation of a `renderHTML` method that points to a JSP file (`sample.jsp`) to render the view. Note that the `itemSelectedEventName` is passed as a request attribute so it can be used in the view markup. The view markup is specified via the `ServletContext` method `getRequestDispatcher`. Although this example uses JSPs, you can use another language such as FreeMarker to render the markup:

    ```
    @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);
    }
    ```

5. Use the `@Reference` annotation to reference your module's class for the `setServletContext` method. In the annotation, use the `target` parameter to specify the available services for the servlet context. This example uses the `osgi.web.symbolicname` property to specify the `com.liferay.selector.sample.web` class as the default value. You should also use the `unbind = -` parameter to specify that there's no unbind method for this module. In the method body, simply set the servlet context variable:

```
@Reference(
    target =
    "(osgi.web.symbolicname=com.liferay.item.selector.sample.web)",
    unbind = "-"
)
public void setServletContext(ServletContext servletContext) {
    _servletContext = servletContext;
}
```

6. Define the `_supportedItemSelectorReturnTypes` list that you referenced in step 2 with the return types that this view supports. This example adds the `URLItemSelectorReturnType` class and `FileEntryItemSelectorReturnType` class to the list of supported return types (you can use more return types if needed). More return types means that the view is more reusable. Also note that this example defines its servlet context variable at the bottom of the file:

```
private static final List<ItemSelectorReturnType>
    _supportedItemSelectorReturnTypes =
    Collections.unmodifiableList(
        ListUtil.fromArray(
            new ItemSelectorReturnType[] {
                new FileEntryItemSelectorReturnType(),
                new URLItemSelectorReturnType()
            }));

 private ServletContext _servletContext;
```

For a real-world example of a view class, see the `SiteNavigationMenuItemItemSelectorView` class.

Writing Your View Markup

Now that you've implemented your selection view's class, you must write the markup that renders the view. The exact markup you write depends on your app's needs. It also depends on your personal preferences, as you can write it with taglibs, AUI components, or even pure HTML and JavaScript. Therefore, there's no standard or typical view markup, even for simple applications. Regardless, the markup must do two key things:

- Render the entities for the user to select.
- When an entity is selected, pass the information specified by the Item Selector return type via a JavaScript event.

For example, the example view class in the previous section passes the JavaScript event name as a request attribute in the `renderHTML` method. You can therefore use this event name in the markup:

```
Liferay.fire(
       `<%= {ITEM_SELECTED_EVENT_NAME} %>',

       {
           data:{
```

```
            the-data-your-client-needs-according-to-the-return-type
        }
    }
);
```

For a complete, real-world example, see the `layouts.jsp` view markup for the `com.liferay.layout.item.selector.w` module. Even though this example is for the previous version of Liferay DXP, it still applies to 7.0. Here's a walkthrough of this `layouts.jsp` file:

1. This `layouts.jsp` file first defines some variables. Note that `LayoutItemSelectorViewDisplayContext` is an optional class that contains additional information about the criteria and view:

    ```
    <%
    LayoutItemSelectorViewDisplayContext layoutItemSelectorViewDisplayContext =
        (LayoutItemSelectorViewDisplayContext)request.getAttribute(
        BaseLayoutsItemSelectorView.LAYOUT_ITEM_SELECTOR_VIEW_DISPLAY_CONTEXT);

    LayoutItemSelectorCriterion layoutItemSelectorCriterion =
        layoutItemSelectorViewDisplayContext.getLayoutItemSelectorCriterion();

    Portlet portlet = PortletLocalServiceUtil.getPortletById(company.getCompanyId(),
        portletDisplay.getId());
    %>
    ```

2. This snippet 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 tutorial Using the Liferay Util Taglib.

3. 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>
    ```

    This renders the following UI:

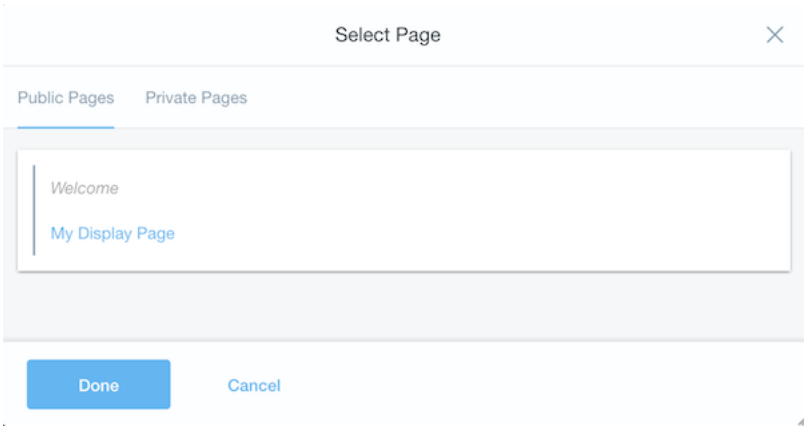


Figure 120.5: The Layouts Item Selector view uses Lexicon and Liferay Layout taglibs to create the UI.

4. 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) {
                                    var labelEl = treeNode.get('labelEl');

                                    if (labelEl) {
                                            labelText = LString.escapeHTML(labelEl.text());

                                            buffer.unshift(labelText);
                                    }
                            }
                    );
            }

            return buffer.join(' > ');
    };
```

5. The following snippet passes the return type data when the layout (entity) is selected. Note the `url` and `uuid` variables 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);
```

6. This checks if the return type information is a URL or a UUID. It then sets the value for the JSON object's `data` attribute accordingly. The last line adds the `CKEditorFuncNum` for the editor to the JSON object's `data` attribute:

```
	<c:choose>
		<c:when test="<%= Objects.equals(layoutItemSelectorViewDisplayContext.getItemSelectorReturnTypeName(), URLItemSel
			data.value = url;
		</c:when>
		<c:when test="<%= Objects.equals(layoutItemSelectorViewDisplayContext.getItemSelectorReturnTypeName(), UUIDItemSe
			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 here:

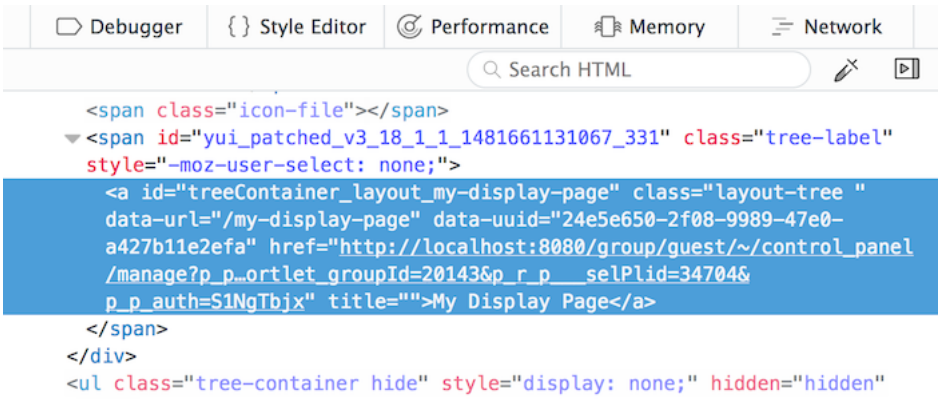


Figure 120.6: The URL and UUID can be seen in the `data-url` and `data-uuid` attributes of the Layout Item Selector's HTML markup.

7. 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
		}
	);
};
```

8. 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.

Great! Now you know how to create custom views for the Item Selector.

Related Topics

Understanding the Item Selector API's Components
Selecting Entities Using the Item Selector
Creating Custom Item Selector Entities

CHAPTER 121

DOCUMENTS AND MEDIA API

Liferay DXP's Documents and Media library stores uploaded files so users can use, manage, and share them. For example, users can embed files in content, organize them in folders, edit and collaborate on them with other users, and more. See the user guide for more information on the Documents and Media library's features.

A powerful API underlies the Documents and Media library's functionality. You can leverage this API in your own apps. For example, you could create an app that lets users upload files to the Documents and Media library. Your app could even let users update, delete, and copy files.

The tutorials in this section show you how to use the Documents and Media library's API. Note that this is a large API and it may seem daunting at first. To keep backwards compatibility, the API has different entry points and multiple methods or classes with similar functionality. Fortunately, you don't need to learn all of them. These tutorials therefore focus on the API's most useful classes and methods.

Also note that the Documents and Media app is itself a consumer of this API—Liferay's developers used the API to implement the app's functionality. Therefore, these tutorials use code from this app as an example of how to use the API.

121.1 Getting Started with the Documents and Media API

Before you start using the Documents and Media API, you must learn these things:

- **Key Interfaces:** The interfaces you'll use most while using the API.
- **Getting a Service Reference:** How to get a service reference that lets you call the API's services.
- **Specifying Repositories:** How to specify which Documents and Media repository you want.
- **Specifying Folders:** How to specify which folder you want.

Key Interfaces

The Documents and Media API contains several key interfaces:

- **Documents and Media Services:** These interfaces expose all the available Documents and Media functionality:

  - `DLAppLocalService`: The local service.
  - `DLAppService`: The remote service. This service wraps the local service methods in permission checks.

  Note that Liferay used Service Builder to create these services. Because the remote service contains permission checks, it's a best practice to call it instead of the local service. See the section below for instructions on getting a service reference.

- **Entity Interfaces:** These interfaces represent entities in the Documents and Media library. Here are the primary ones you'll use:

  - `FileEntry`: Represents a file.
  - `Folder`: Represents a folder.
  - `FileShortcut`: Represents a shortcut to a file.

Getting a Service Reference

Before you can do anything with the Documents and Media API, you must get a service reference. If you're using OSGi modules, use the `@Reference` annotation to get a service reference in an OSGi component via Declarative Services. For example, this code gets a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

If you're using a standard web module (WAR file), use a Service Tracker to get a reference to the service instead.

Getting the reference this way ensures that you leverage OSGi's dependency management features. If you must use the Documents and Media services outside of an OSGi component (e.g., in a JSP), then you can use the services' static `*Util` classes:

- `DLAppServiceUtil`
- `DLAppLocalServiceUtil`

Specifying Repositories

Many methods in the Documents and Media API contain a `repositoryId` parameter that identifies the Documents and Media repository where the operation is performed. A Site (group) can have multiple repositories, but only one can be accessed via the portal UI. This is called the Site (group) repository, which is effectively a Site's default repository. To access this repository via the API, provide the group ID as the `repositoryId`.

You can also get the `repositoryId` via file (`FileEntry`), folder (`Folder`), and file shortcut (`FileShortcut`) entities. Each of these entities has a `getRepositoryId` method that gets the ID of its repository. For example, this code gets the repository ID of the `FileEntry` object `fileEntry`:

```
long repositoryId = fileEntry.getRepositoryId();
```

There may also be cases that require a `Repository` object. You can get one by creating a `RepositoryProvider` reference and passing the repository ID to its `getRepository` method:

```
@Reference
private RepositoryProvider repositoryProvider;

Repository repository = repositoryProvider.getRepository(repositoryId);
```

Even if you only have an entity ID (e.g., a file ID or folder ID), you can still use `RepositoryProvider` to get a `Repository` object. To do so, call the `RepositoryProvider` method for the entity type with the entity ID as its argument. For example, this code gets a folder's `Repository` by calling the `RepositoryProvider` method `getFolderRepository` with the folder's ID:

```
Repository repository = repositoryProvider.getFolderRepository(folderId);
```

See the `RepositoryProvider` Javadoc for a list of the methods for other entity types.

Note that there are ways to create repositories programmatically, including repositories private to specific apps. For simplicity, however, the tutorials here access the default site repository.

Specifying Folders

Many API methods require the ID of a folder that they perform operations in or on. For example, such methods may contain parameters like `folderId` or `parentFolderId`. Also note that you can use the constant `DLFolderConstants.DEFAULT_PARENT_FOLDER_ID` to specify the root folder of your current repository.

Related Topics

Service Builder
- OSGi Services and Dependency Injection with Declarative Services
- Leveraging Dependencies
- Service Trackers

CHAPTER 122

CREATING FILES, FOLDERS, AND SHORTCUTS

The primary use case for the API is to create files, folders, and file shortcuts in the Documents and Media library.

If you've used other Liferay APIs, the Docs & Media API follows the same conventions. In general, methods that do similar things tend to have similar names. When you must create an entity (whatever it is), look for methods that follow the pattern `add[ModelName]`, where `[ModelName]` is the name of the entity's data model object. As the getting started tutorial explains, you'll use `DLAppService` to access the API. This service object contains the following methods for adding entities:

- `addFileEntry`: Adds a file.
- `addFolder`: Adds a folder.
- `addFileShortcut`: Adds a shortcut to a file.

The tutorials that follow show you how to use these methods.

122.1 Creating Files

To create files (`FileEntry` entities) in the Documents and Media library, you must use the `DLAppService` interface's `addFileEntry` methods. There are three such methods, and they differ by the data type used to create the file. Click each method to see a full description of the method and its parameters:

- `addFileEntry(..., byte[] bytes, ...)`
- `addFileEntry(..., File file, ...)`
- `addFileEntry(..., InputStream is, long size, ...)`

Note that the following arguments are optional:

- `sourceFileName`: This keeps track of the uploaded file. It infers the content type if that file has an extension.

- `mimeType`: Defaults to a binary stream. If omitted, Documents and Media tries to infer the type from the file extension.
- `description`: The file's description to display in the portal.
- `changeLog`: Descriptions for file versions.
- `is` and `size`: In the method that takes an `InputStream`, you can use `null` for the `is` parameter. If you do this, however, you must use `0` for the `size` parameter.

Follow these steps to create a file via the `DLAppService` method `addFileEntry`. Note that these steps use the method that contains `InputStream`:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `addFileEntry` method's arguments. Since it's common to create a file with data submitted by the end user, you can extract the data from the request. This example does so via `UploadPortletRequest` and `ParamUtil`, but you can get the data any way you wish:

   ```
   long repositoryId = ParamUtil.getLong(uploadPortletRequest, "repositoryId");
   long folderId = ParamUtil.getLong(uploadPortletRequest, "folderId");
   String sourceFileName = uploadPortletRequest.getFileName("file");
   String title = ParamUtil.getString(uploadPortletRequest, "title");
   String description = ParamUtil.getString(uploadPortletRequest, "description");
   String changeLog = ParamUtil.getString(uploadPortletRequest, "changeLog");
   boolean majorVersion = ParamUtil.getBoolean(uploadPortletRequest, "majorVersion");

   try (InputStream inputStream = uploadPortletRequest.getFileAsStream("file")) {

       String contentType = uploadPortletRequest.getContentType("file");
       long size = uploadPortletRequest.getSize("file");

       ServiceContext serviceContext = ServiceContextFactory.getInstance(
               DLFileEntry.class.getName(), uploadPortletRequest);
   }
   ```

   For more information on getting repository and folder IDs, see the getting started tutorial's sections on specifying repositories and folders. For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `addFileEntry` method with the data from the previous step. Note that this example does so inside the previous step's try-with-resources statement:

   ```
   try (InputStream inputStream = uploadPortletRequest.getFileAsStream("file")) {

       ...

       FileEntry fileEntry = _dlAppService.addFileEntry(
                           repositoryId, folderId, sourceFileName, contentType, title,
                           description, changeLog, inputStream, size, serviceContext);
   }
   ```

The method returns a `FileEntry` object, which this example sets to a variable for later use. Note, however, that you don't have to do this.

You can find the full code for this example in the `updateFileEntry` method of Liferay DXP's `EditFileEntryMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileEntry` actions that the Documents and Media app supports. Also note that this `updateFileEntry` method, as well as the rest of `EditFileEntryMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

Updating Files
- Deleting Files
- Moving Folders and Files
- Creating Folders
- Creating File Shortcuts

122.2 Creating Folders

To create folders (`Folder` entities) in the Documents and Media library, you must use the `DLAppService` interface's `addFolder` method:

```
addFolder(long repositoryId,
        long parentFolderId,
        String name,
        String description,
        ServiceContext serviceContext)
```

See this method's Javadoc for a description of the parameters. Note that the `description` parameter is optional.

Follow these steps to create a folder with the `DLAppService` method `addFolder`:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `addFolder` method's arguments. Since it's common to create a folder with data submitted by the end user, you can extract the data from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`:

   ```
   long repositoryId = ParamUtil.getLong(actionRequest, "repositoryId");
   long parentFolderId = ParamUtil.getLong(actionRequest, "parentFolderId");
   String name = ParamUtil.getString(actionRequest, "name");
   String description = ParamUtil.getString(actionRequest, "description");

   ServiceContext serviceContext = ServiceContextFactory.getInstance(
              DLFolder.class.getName(), actionRequest);
   ```

For more information on getting repository and folder IDs, see the getting started tutorial's sections on specifying repositories and folders. For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `addFolder` method with the data from the previous step:

```
Folder folder = _dlAppService.addFolder(
                        repositoryId, parentFolderId, name, description,
                        serviceContext);
```

The method returns a `Folder` object, which this example sets to a variable for later use. Note, however, that you don't have to do this.

You can find the full code for this example in the `updateFolder` method of Liferay DXP's `EditFolderMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `Folder` actions that the Documents and Media app supports. Also note that this `updateFolder` method, as well as the rest of `EditFolderMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Folders and External Repositories

By creating a folder that acts as a proxy for an external repository (e.g., SharePoint), you can effectively mount that repository inside a Site's default repository. When users enter this special folder, they see the external repository. These folders are called *mount points*. You can create one via the API by setting the Service Context's `mountPoint` attribute to `true`, and then using that Service Context in the `addFolder` method:

```
serviceContext.setAttribute("mountPoint", true);
```

Note that the `repositoryId` of such a folder indicates the external repository the folder points to—not the repository the folder exists. Also, mount point folders can only exist in the default Site repository.

Related Topics

Updating Folders
Deleting Folders
Copying Folders
Moving Folders and Files

122.3 Creating File Shortcuts

To create file shortcuts (`FileShortcut` entities) in the Documents and Media library, you must use the `DLAppService` interface's `addFileShortcut` method:

```
addFileShortcut(long repositoryId,
                long folderId,
                long toFileEntryId,
                ServiceContext serviceContext)
```

See this method's Javadoc for a description of the parameters. Note that all this method's parameters are mandatory.

Keep these things in mind when creating shortcuts:

- You can create a shortcut to a file in a different Site, if that file and its resulting shortcut are in the same portal instance.
- You can't create folder shortcuts.
- Shortcuts can only exist in the default Site repository. If you try to invoke `addFileShortcut` with an external repository's ID (e.g., a SharePoint repository), the operation fails. Because not all repositories have the same features, the Documents and Media API only supports the common denominators for all repositories: files and folders.

Follow these steps to create a file shortcut with the `DLAppService` method `addFileShortcut`:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `addFileShortcut` method's arguments. Since it's common to create a file shortcut with data submitted by the end user, you can extract the data from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get the data any way you wish:

   ```
   long repositoryId = ParamUtil.getLong(actionRequest, "repositoryId");
   long folderId = ParamUtil.getLong(actionRequest, "folderId");
   long toFileEntryId = ParamUtil.getLong(actionRequest, "toFileEntryId");

   ServiceContext serviceContext = ServiceContextFactory.getInstance(
                       DLFileShortcutConstants.getClassName(), actionRequest);
   ```

   For more information on getting repository and folder IDs, see the getting started tutorial's sections on specifying repositories and folders. For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `addFileShortcut` method with the data from the previous step:

   ```
   FileShortcut fileShortcut = _dlAppService.addFileShortcut(
                                       repositoryId, folderId, toFileEntryId,
                                       serviceContext);
   ```

   The method returns a `FileShortcut` object, which this example sets to a variable for later use. Note, however, that you don't have to do this.

You can find the full code for this example in the `updateFileShortcut` method of Liferay DXP's `EditFileShortcutMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileShortcut` actions that the Documents and Media app supports. Also note that this `updateFileShortcut` method, as well as the rest of `EditFileShortcutMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

Deleting File Shortcuts
    Updating File Shortcuts

CHAPTER 123

DELETING ENTITIES

Now that you know how to create Documents and Media entities, you should learn how to delete them. Note that the exact meaning of *delete* depends on the portal configuration and the delete operation you choose. This is because the Recycle Bin, which is enabled by default, can be used to recover deleted items. Deletions via `DLAppService`, however, are permanent. To send items to the Recycle Bin, you must use the Capabilities API.

This section of tutorials shows you how to use `DLAppService` to delete entities from the Documents and Media library. The last tutorial in this section shows you how to move entities to the Recycle Bin via the Capabilities API.

123.1 Deleting Files

There are two methods you can use to delete files:

- `deleteFileEntry(long fileEntryId)`
- `deleteFileEntryByTitle(long repositoryId, long folderId, String title)`

These methods differ only in how they identify a file for deletion. The combination of the `folderId` and `title` parameters in `deleteFileEntryByTitle` uniquely identify a file because it's impossible for two files in the same folder to share a name.

Follow these steps to delete a file:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the arguments of the `deleteFileEntry*` method you wish to use. Since it's common to delete a file specified by the end user, you can extract the data you

need from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get the data any way you wish. Also note that this example gets only the file entry ID because it uses `deleteFileEntry`:

```
long fileEntryId = ParamUtil.getLong(actionRequest, "fileEntryId");
```

If you want to use `deleteFileEntryByTitle` instead, you can also get the repository ID, folder ID, and title from the request. For more information on getting repository and folder IDs, see the getting started tutorial's sections on specifying repositories and folders.

3. Call the service reference's `deleteFileEntry*` method you wish to use with the data from the previous step. This example calls `deleteFileEntry` with the file entry's ID:

```
_dlAppService.deleteFileEntry(fileEntryId);
```

You can find the full code for this example in the `deleteFileEntry` method of Liferay DXP's `EditFileEntryMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileEntry` actions that the Documents and Media app supports. Also note that this `deleteFileEntry` method, as well as the rest of `EditFileEntryMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

Moving Entities to the Recycle Bin
Creating Files
Updating Files
Moving Folders and Files

123.2 Deleting File Versions

When a file is modified, Documents and Media creates a new file version and leaves the previous version intact. Over time, old file versions can accumulate and consume precious storage space. Fortunately, you can use the Documents and Media API to delete them. Note, however, that there's no way to send file versions to the Recycle Bin—once you delete them, they're gone forever.

You can delete file versions with the `DLAppService` method `deleteFileVersion`:

```
deleteFileVersion(long fileEntryId, String version)
```

See this method's Javadoc for a description of the parameters.
Follow these steps to use `deleteFileVersion` to delete a file version:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the file entry ID and version for the file you want to delete. Since it's common to delete a file version specified by the end user, you can extract these parameters from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can do this any way you wish:

```
long fileEntryId = ParamUtil.getLong(actionRequest, "fileEntryId");
String version = ParamUtil.getString(actionRequest, "version");
```

3. Use the service reference to call the `deleteFileVersion` method with the file entry ID and version from the previous step:

```
_dlAppService.deleteFileVersion(fileEntryId, version);
```

You can find the full code for this example in the `deleteFileEntry` method of Liferay DXP's `EditFileEntryMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileEntry` actions that the Documents and Media app supports. Also note that this `deleteFileEntry` method, as well as the rest of `EditFileEntryMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Identifying File Versions

Since there may be many versions of a file, it's useful to identify programmatically old versions for deletion. You can do this with `FileVersionVersionComparator`.

The following example creates such a comparator and uses its `compare` method to identify old versions of a file. The code does so by iterating through each approved version of the file (`fileVersion`). Each iteration uses the `compare` method to test that file version (`fileVersion.getVersion()`) against the same file's current version (`fileEntry.getVersion()`). If this comparison is greater than `0`, then the iteration's file version (`fileVersion`) is old and is deleted by `deleteFileVersion`:

```
FileVersionVersionComparator comparator = new FileVersionVersionComparator();

for (FileVersion fileVersion: fileEntry.getVersions(WorkflowConstants.STATUS_APPROVED)) {

    if (comparator.compare(fileEntry.getVersion(), fileVersion.getVersion()) > 0) {

        dlAppService.deleteFileVersion(fileVersion.getFileEntryId(), fileVersion.getVersion());
    }
}
```

Related Topics

Deleting Files
- Deleting File Shortcuts
- Deleting Folders
- Moving Entities to the Recycle Bin

123.3 Deleting File Shortcuts

The Documents and Media API also lets you delete file shortcuts. To do so, use the `DLAppService` method `deleteFileShortcut` with the ID of the shortcut you want to delete:

`deleteFileShortcut(long fileShortcutId)`

Follow these steps to use this method to delete a file shortcut:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the file shortcut's ID. Since it's common to delete a file shortcut specified by the end user, you can extract its ID from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can do this any way you wish:

   ```
   long fileShortcutId = ParamUtil.getLong(actionRequest, "fileShortcutId");
   ```

3. Use the service reference to call the `deleteFileShortcut` method with the file shortcut ID from the previous step:

   ```
   _dlAppService.deleteFileShortcut(fileShortcutId);
   ```

You can find the full code for this example in the `deleteFileShortcut` method of Liferay DXP's `EditFileShortcutMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileShortcut` actions that the Documents and Media app supports. Also note that this `deleteFileShortcut` method, as well as the rest of `EditFileShortcutMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

Moving Entities to the Recycle Bin
Creating File Shortcuts
Updating File Shortcuts

123.4 Deleting Folders

Deleting folders is similar to deleting files. There are two methods you can use to delete a folder. Click each method to see its Javadoc:

- `deleteFolder(long folderId)`
- `deleteFolder(long repositoryId, long parentFolderId, String name)`

Which method you use is up to you—they both delete a folder. Follow these steps to use one of these methods to delete a folder:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the arguments of the `deleteFolder*` method you wish to use. Since it's common to delete a folder specified by the end user, you can extract the data you need from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get the data any way you wish. Also note that this example gets only the folder ID because the next step deletes the folder with `deleteFolder(folderId)`:

```
long folderId = ParamUtil.getLong(actionRequest, "folderId");
```

If you want to use the other `deleteFolder` method, you can also get the repository ID, parent folder ID, and folder name from the request. For more information on getting repository and folder IDs, see the getting started tutorial's sections on specifying repositories and folders.

3. Call the service reference's `deleteFolder*` method you wish to use with the data from the previous step. This example calls `deleteFolder` with the folder's ID:

```
_dlAppService.deleteFolder(folderId);
```

You can find the full code for this example in the `deleteFolders` method of Liferay DXP's `EditFolderMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `Folder` actions that the Documents and Media app supports. Also note that this `deleteFolders` method, as well as the rest of `EditFolderMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

Moving Entities to the Recycle Bin
- Creating Folders
- Updating Folders
- Copying Folders
- Moving Folders and Files
- Deleting Files

123.5 Moving Entities to the Recycle Bin

Instead of deleting entities, you can move them to the Recycle Bin. Note that the Recycle Bin isn't part of the Documents and Media API. Although you can use the Recycle Bin API directly, in the case of Documents and Media it's better to use the Capabilities API. This is because some third-party repositories (e.g., SharePoint) don't support Recycle Bin functionality. The Capabilities API lets you verify that the repository you're working in supports the Recycle Bin. It's therefore a best practice to always use the Capabilities API when moving entities to the Recycle Bin.

Follow these steps to use the Capabilities API to move an entity to the Recycle Bin:

1. Verify that the repository supports the Recycle Bin. Do this by calling the repository object's `isCapabilityProvided` method with `TrashCapability.class` as its argument. This example does so in `if` statement's condition:

```
if (repository.isCapabilityProvided(TrashCapability.class)) {
    // The code to move the entity to the Recycle Bin
    // You'll write this in the next step
}
```

2. Move the entity to the Recycle Bin if the repository supports it. To do this, first get a `TrashCapability` reference by calling the repository object's `getCapability` method with `TrashCapability.class` as its argument. Then call the `TrashCapability` method that moves the entity to the Recycle Bin. For example, this code calls `moveFileEntryToTrash` to move a file to the Recycle Bin:

```
if (repository.isCapabilityProvided(TrashCapability.class)) {

    TrashCapability trashCapability = repository.getCapability(TrashCapability.class);
    trashCapability.moveFileEntryToTrash(user.getUserId(), fileEntry);
}
```

   See the `TrashCapability` Javadoc for information on the methods you can use to move other types of entities to the Recycle Bin.

Related Topics

Deleting Files
- Deleting Folders
- Deleting File Shortcuts
- Moving Folders and Files

CHAPTER 124

UPDATING ENTITIES

Like creating and deleting entities, updating entities is a key task when working with Documents and Media. The methods in the Documents and Media API for creating and updating entities are similar. There are, however, a few important differences. The tutorials in this section show you how to update entities and highlight these differences.

124.1 Updating Files

Updating a file is a bit more complicated than creating one. This is due to the way the update operation handles a file's metadata and content. To update only a file's content, you must also supply the file's existing metadata. Otherwise, the update operation could lose the metadata. The opposite, however, isn't true. You can modify a file's metadata without re-supplying the content. In such an update, the file's content is automatically copied to the new version of the file. To make this easier to remember, follow these rules when updating files:

- Always provide all metadata.
- Only provide the file's content when you want to change it.

`DLAppService` has three `updateFileEntry` methods that you can use to update a file. These methods differ only in the file content's type. Click each method to see its Javadoc, which contains a full description of its parameters:

- `updateFileEntry(..., byte[] bytes, ...)`
- `updateFileEntry(..., File file, ...)`
- `updateFileEntry(..., InputStream is, long size, ...)`

Keep these things in mind when using these methods:

- To retain the original file's title and description, you must provide those parameters to `updateFileEntry`. Omitting them deletes any existing title and description.

- If you supply `null` in place of the file's content (e.g., `bytes`, `file`, or `is`), the update automatically uses the file's existing content. Do this only if you want to update the file's metadata.
- If you use `false` for the `majorVersion` parameter, the update increments the file version by `0.1` (e.g., from `1.0` to `1.1`). If you use `true` for this parameter, the update increments the file version to the next `.0` value (e.g., from `1.0` to `2.0`, `1.1` to `2.0`, etc.).

Follow these steps to update a file. Note that the example in these steps uses the `updateFileEntry` method that contains `InputStream`, but you can adapt the example to the other methods if you wish:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `updateFileEntry` method's arguments. Since it's common to update a file with data submitted by the end user, you can extract the data from the request. This example does so via `UploadPortletRequest` and `ParamUtil`, but you can get the data any way you wish:

   ```
   long repositoryId = ParamUtil.getLong(uploadPortletRequest, "repositoryId");
   long folderId = ParamUtil.getLong(uploadPortletRequest, "folderId");
   String sourceFileName = uploadPortletRequest.getFileName("file");
   String title = ParamUtil.getString(uploadPortletRequest, "title");
   String description = ParamUtil.getString(uploadPortletRequest, "description");
   String changeLog = ParamUtil.getString(uploadPortletRequest, "changeLog");
   boolean majorVersion = ParamUtil.getBoolean(uploadPortletRequest, "majorVersion");

   try (InputStream inputStream = uploadPortletRequest.getFileAsStream("file")) {

       String contentType = uploadPortletRequest.getContentType("file");
       long size = uploadPortletRequest.getSize("file");

       ServiceContext serviceContext = ServiceContextFactory.getInstance(
               DLFileEntry.class.getName(), uploadPortletRequest);
   }
   ```

   For more information on getting repository and folder IDs, see the getting started tutorial's sections on specifying repositories and folders. For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `updateFileEntry` method with the data from the previous step. Note that this example does so inside the previous step's try-with-resources statement:

   ```
   try (InputStream inputStream = uploadPortletRequest.getFileAsStream("file")) {

       ...

       FileEntry fileEntry = _dlAppService.updateFileEntry(
                               fileEntryId, sourceFileName, contentType, title,
                               description, changeLog, majorVersion, inputStream, size,
                               serviceContext);
   }
   ```

The method returns a `FileEntry` object, which this example sets to a variable for later use. Note, however, that you don't have to do this.

You can find the full code for this example in the `updateFileEntry` method of Liferay DXP's `EditFileEntryMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileEntry` actions that the Documents and Media app supports. Also note that this `updateFileEntry` method, as well as the rest of `EditFileEntryMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

Creating Files
Deleting Files
Moving Folders and Files

124.2 Updating Folders

The Documents and Media API lets you copy or move folders to a different location. Options for in-place folder updates, however, are limited. You can only update a folder's name and description. You can do this with the `DLAppService` method `updateFolder`:

```
updateFolder(long folderId, String name, String description, ServiceContext serviceContext)
```

All parameters except the description are mandatory. For a full description of this method and its parameters, see its Javadoc.

Follow these steps to use this method to update a folder:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `updateFolder` method's arguments. Since it's common to update a folder with data submitted by the end user, you can extract the data from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get the data any way you wish:

   ```
   long folderId = ParamUtil.getLong(actionRequest, "folderId");
   String name = ParamUtil.getString(actionRequest, "name");
   String description = ParamUtil.getString(actionRequest, "description");

   ServiceContext serviceContext = ServiceContextFactory.getInstance(
              DLFolder.class.getName(), actionRequest);
   ```

   For more information on getting folder IDs, see the getting started tutorial's section on specifying folders. For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `updateFolder` method with the data from the previous step:

```
_dlAppService.updateFolder(folderId, name, description, serviceContext);
```

You can find the full code for this example in the `updateFolder` method of Liferay DXP's `EditFolderMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `Folder` actions that the Documents and Media app supports. Also note that this `updateFolder` method, as well as the rest of `EditFolderMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

Creating Folders
Deleting Folders
Copying Folders
Moving Folders and Files

124.3 Updating File Shortcuts

The Documents and Media API lets you update file shortcuts (`FileShortcut` entities). You can update a shortcut to change the file it points to or the folder it resides in. You can do this via the `DLAppService` method `updateFileShortcut`:

```
updateFileShortcut(long fileShortcutId, long folderId, long toFileEntryId, ServiceContext serviceContext)
```

All of this method's parameters are mandatory. To retain any of the shortcut's original values, you must provide them to this method. For a full description of the parameters, see the method's Javadoc.

Follow these steps to use this method to update a file shortcut:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `updateFileShortcut` method's arguments. Since it's common to update a file shortcut with data submitted by the end user, you can extract the data from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get the data any way you wish:

   ```
   long fileShortcutId = ParamUtil.getLong(actionRequest, "fileShortcutId");
   long folderId = ParamUtil.getLong(actionRequest, "folderId");
   long toFileEntryId = ParamUtil.getLong(actionRequest, "toFileEntryId");

   ServiceContext serviceContext = ServiceContextFactory.getInstance(
              DLFileShortcutConstants.getClassName(), actionRequest);
   ```

For more information on getting folder IDs, see the getting started tutorial's section on specifying folders. For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `updateFileShortcut` method with the data from the previous step:

```
_dlAppService.updateFileShortcut(
        fileShortcutId, folderId, toFileEntryId, serviceContext);
```

You can find the full code for this example in the `updateFileShortcut` method of Liferay DXP's `EditFileShortcutMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileShortcut` actions that the Documents and Media app supports. Also note that this `updateFileShortcut` method, as well as the rest of `EditFileShortcutMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

Creating File Shortcuts

Deleting File Shortcuts

CHAPTER 125

FILE CHECK-OUT AND CHECK-IN

The Document Library lets users check out files for editing. Only the user who checked out the file can edit it. This prevents conflicting edits on the same file from multiple users. The Documents and Media API allows these check-in/check-out operations:

- File Check-out
- File Check-in
- Cancelling a Check-out

The tutorials in this section show you how to do these operations.

125.1 File Check-out

Here's what happens when you check out a file:

- A private working copy of the file is created that only you and administrators can access. Until you check the file back in or cancel your changes, any edits you make are stored in the private working copy.
- Other users can't change or edit any version of the file. This state remains until you cancel or check in your changes.

The main `DLAppService` method for checking out a file is this `checkOutFileEntry` method:

```
checkOutFileEntry(long fileEntryId, ServiceContext serviceContext)
```

If this method throws an exception, then you should assume the checkout failed and repeat the operation. For a full description of the method and its parameters, see its Javadoc.

Follow these steps to use this method to check out a file:

1. Get a reference to `DLAppService`:

```
@Reference
private DLAppService _dlAppService;
```

For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `checkOutFileEntry` method's arguments. Since it's common to check out a file in response to an action by the end user, you can extract the data from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get the data any way you wish:

```
long fileEntryId = ParamUtil.getLong(actionRequest, "fileEntryId");

ServiceContext serviceContext = ServiceContextFactory.getInstance(actionRequest);
```

   For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `checkOutFileEntry` method with the data from the previous step:

```
_dlAppService.checkOutFileEntry(fileEntryId, serviceContext);
```

You can find the full code for this example in the `checkOutFileEntries` method of Liferay DXP's `EditFileEntryMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileEntry` actions that the Documents and Media app supports. Also note that this `checkOutFileEntries` method, as well as the rest of `EditFileEntryMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Fine-tuning Checkout

You can control how the checkout is performed by setting the following attributes in the `ServiceContext` parameter:

- `manualCheckInRequired`: By default, the system automatically checks out/in a file when a user edits it. Setting this attribute to `true` prevents this, therefore requiring manual check-out and check-in.
- `existingDLFileVersionId`: The system typically reuses the private working copy across different check-out/check-in sequences. There's little chance for conflicting edits because only one user at a time can access the private working copy. To force the system to create a new private working copy each time, omit this attribute or set it to `0`.
- `fileVersionUuid`: This is used by staging, but can be ignored for normal use. Setting this attribute causes the system to create the new private working copy version with the given UUID.

To set these attributes, use the `ServiceContext` method `setAttribute(String name, Serializable value)`. Here's an example of setting the `manualCheckInRequired` attribute to `true`:

```
serviceContext.setAttribute("manualCheckInRequired", Boolean.TRUE)
```

Related Topics

File Check-in
    Cancelling a Check-out
    Updating Files

125.2 File Check-in

After checking out and editing a file, you must check it back in for other users to see the new version. Once you do so, you can't access the private working copy. The next time the file is checked out, the private working copy's contents are overwritten.

The `DLAppService` method for checking in a file is `checkInFileEntry`:

```
checkInFileEntry(long fileEntryId, boolean majorVersion, String changeLog,
                ServiceContext serviceContext)
```

For a full description of the method and its parameters, see its Javadoc. This method uses the private working copy to create a new version of the file. As the Updating Files tutorial explains, the `majorVersion` parameter's setting determines how the file's version number is incremented.

Follow these steps to use `checkInFileEntry` to check in a file:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `checkInFileEntry` method's arguments. Since it's common to check in a file in response to an action by the end user, you can extract the data from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get the data any way you wish:

   ```
   long fileEntryId = ParamUtil.getLong(actionRequest, "fileEntryId");
   boolean majorVersion = ParamUtil.getBoolean(actionRequest, "majorVersion");
   String changeLog = ParamUtil.getString(actionRequest, "changeLog");

   ServiceContext serviceContext = ServiceContextFactory.getInstance(actionRequest);
   ```

   For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `checkInFileEntry` method with the data from the previous step:

   ```
   _dlAppService.checkInFileEntry(
           fileEntryId, majorVersion, changeLog, serviceContext);
   ```

You can find the full code for this example in the `checkInFileEntries` method of Liferay DXP's `EditFileEntryMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileEntry` actions that the Documents and Media app supports. Also note that this `checkInFileEntries` method, as well as the rest of `EditFileEntryMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

File Check-out
    Cancelling a Check-out
    Updating Files

125.3 Canceling a Check-out

The Documents and Media API also lets you cancel a check-out. Use caution with this operation—it discards any edits made since check-out. If you're sure you want to cancel a check-out, do so with the `DLAppService` method `cancelCheckOut`:

```
cancelCheckOut(long fileEntryId)
```

For a full description of this method and its parameter, see its Javadoc. If you invoke this method without error, you can safely assume that it discarded the private working copy and unlocked the file. Other users should now be able to check out and edit the file.

Follow these steps to cancel a check-out:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the ID of the file whose check-out you want to cancel. Since it's common to cancel a check-out in response to a user action, you can extract the file ID from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get it any way you wish:

   ```
   long fileEntryId = ParamUtil.getLong(actionRequest, "fileEntryId");
   ```

3. Call the service reference's `cancelCheckOut` method with the file's ID:

   ```
   _dlAppService.cancelCheckOut(fileEntryId);
   ```

You can find the full code for this example in the `cancelFileEntriesCheckOut` method of Liferay DXP's `EditFileEntryMVCActionCommand` class. This class uses the Documents and Media API to implement almost all the `FileEntry` actions that the Documents and Media app supports. Also note that this `cancelFileEntriesCheckOut` method, as well as the rest of `EditFileEntryMVCActionCommand`, contains additional logic to suit the specific needs of the Documents and Media app.

Related Topics

File Check-out
File Check-in
Updating Files

CHAPTER 126

COPYING AND MOVING ENTITIES

Although the Documents and Media API lets you copy and move entities, these operations have some important caveats and limitations. Keep these things in mind when copying entities:

- There's no way to copy files—you can only copy folders. However, copying a folder also copies its contents, which can include files.
- Folders can only be copied within their current repository.

The move operation doesn't have these restrictions. It's possible to move files and folders between different repositories. In general, however, the move operation is a bit more complicated than the copy operation. For example, the API's behavior changes depending on whether you move entities to a different repository or within the same one.

The tutorials in this section cover these differences, and more.

126.1 Copying Folders

The Documents and Media API lets you copy folders within a repository. You can't, however, copy a folder between different repositories. Also note that copying a folder also copies its contents.

To copy a folder, use the `DLAppService` method `copyFolder`:

```
copyFolder(long repositoryId, long sourceFolderId, long parentFolderId, String name,
        String description, ServiceContext serviceContext)
```

For a full description of the method and its parameters, see its Javadoc.

Follow these steps to use this method to copy a folder:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the `copyFolder` method's arguments. How you do this depends on your use case. The copy operation in this example takes place in the default Site repository and retains the folder's existing name and description. It therefore needs the folder's group ID (to specify the default site repository), name, and description. Also note that because the destination folder in this example is the repository's root folder, the parent folder ID isn't needed—Liferay DXP supplies a constant for specifying a repository's root folder.

   In the following code, `ParamUtil` gets the folder's ID from the request (`javax.portlet.ActionRequest`), and the service reference's `getFolder` method gets the corresponding folder object. The folder's `getGroupId()`, `getName()`, and `getDescription()` methods then get the folder's group ID, name, and description, respectively:

   ```
   long folderId = ParamUtil.getLong(actionRequest, "folderId");

   Folder folder = _dlAppService.getFolder(folderId);
   long groupId = folder.getGroupId();
   String folderName = folder.getName();
   String folderDescription = folder.getDescription();

   ServiceContext serviceContext = ServiceContextFactory.getInstance(
               DLFolder.class.getName(), actionRequest);
   ```

   For more information on getting repository and folder IDs, see the getting started tutorial's sections on specifying repositories and folders. For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's `copyFolder` method with the data from the previous step. Note that this example uses the `DLFolderConstants` constant `DEFAULT_PARENT_FOLDER_ID` to specify the repository's root folder as the destination folder:

   ```
   _dlAppService.copyFolder(
           groupId, folderId, DLFolderConstants.DEFAULT_PARENT_FOLDER_ID,
           folderName, folderDescription, serviceContext);
   ```

Keep in mind that you can change any of these values to suit your copy operation. For example, if your copy takes place in a repository other than the default Site repository, you would specify that repository's ID in place of the group ID. You could also specify a different destination folder, and/or change the new folder's name and/or description.

Related Topics

Getting Started with the Documents and Media API
- Understanding Service Context
- Creating Folders
- Updating Folders
- Deleting Folders
- Moving Folders and Files

126.2 Moving Folders and Files

The move operation is more flexible than the copy operation. Copying only works with folders, and you can't copy between repositories. The move operation, however, works with files and folders within or between repositories.

Note: Depending on the repository implementation, you may get unexpected behavior when moving folders between repositories. Moving a folder also moves its contents via separate move operations for each item in the folder. In some repository implementations, if any move sub-operation fails, the parent move operation also fails. In other repository implementations, the results of successful sub-operations remain even if others fail, which leaves a partially complete move of the whole folder.

To move a folder, use the `DLAppService` method `moveFolder`:

```
moveFolder(long folderId, long parentFolderId, ServiceContext serviceContext)
```

For a full description of this method and its parameters, see its Javadoc. This method is similar to `copyFolder`, but it doesn't let you change the folder's name or description, and it can move folders between repositories. Folder contents are moved with the folder.

The operation for moving a file is almost identical to moving a folder. To move a file, use the `DLAppService` method `moveFileEntry`:

```
moveFileEntry(long fileEntryId, long newFolderId, ServiceContext serviceContext)
```

For a full description of this method and its parameters, see its Javadoc.

Follow these steps to use `moveFolder` and `moveFileEntry` to move a folder and a file, respectively. Although this example does both just to demonstrate the procedure:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the method arguments. Since moving folders and files is typically done in response to a user action, you can get the data from the request. This example does so via `javax.portlet.ActionRequest` and `ParamUtil`, but you can get the data any way you wish:

   ```
   // Get the folder IDs
   long folderId = ParamUtil.getLong(actionRequest, "folderId");
   long newFolderId = ParamUtil.getLong(actionRequest, "newFolderId");

   // Get the file ID
   long fileEntryId = ParamUtil.getLong(actionRequest, "fileEntryId");

   ServiceContext serviceContext = ServiceContextFactory.getInstance(
           DLFileEntry.class.getName(), actionRequest);
   ```

For more information on getting folder IDs, see the getting started tutorial's section on specifying folders. For more information on `ServiceContext`, see the tutorial Understanding ServiceContext.

3. Call the service reference's method(s). This example calls `moveFolder` to move a folder (`folderId`) to a different folder (`newFolderId`). It then calls `moveFileEntry` to move a file (`fileEntryId`) to the same destination folder:

```
_dlAppService.moveFolder(folderId, newFolderId, serviceContext);

_dlAppService.moveFileEntry(fileEntryId, newFolderId, serviceContext);
```

Related Topics

Copying Folders

CHAPTER 127

GETTING ENTITIES

The Documents and Media API contains many methods for getting entities from a repository. Most methods in `DLAppService` are for getting single entities (e.g., a file or folder), a collection of entities that match certain characteristics, or the number of such entities. Because there are so many similar methods for getting entities, these tutorials don't describe them all in detail. All of them are covered in the reference documentation.

127.1 Getting Files

Getting files is one of the most common tasks you'll perform with the Documents and Media API. There are two main method families for getting files:

- `getFileEntries`: Gets files from a specific repository.
- `getGroupFileEntries`: Gets files from a Site (group), regardless of repository.

Since these method families are common, their methods share many parameters:

- `repositoryId`: The ID of the repository to get files from. To specify the default Site repository, use the `groupId` (Site ID).
- `folderId`: The ID of the folder to get files from. Note that these methods don't traverse the folder structure—they only get files directly from the specified folder. To specify the repository's root folder, use the constant `DLFolderConstants.DEFAULT_PARENT_FOLDER_ID`.
- `start` and `end`: Integers that specify the lower and upper bounds, respectively, of collection items to include in a page of results. If you don't want to use pagination, use `QueryUtil.ALL_POS` for these parameters.
- `obc`: The comparator to use to order collection items. Comparators are `OrderByComparator` implementations that sort collection items.
- `fileEntryTypeId`: The ID of the file type to retrieve. Use this to retrieve files of a specific type.
- `mimeTypes`: The MIME types of the files to retrieve. Use this to retrieve files of the specified MIME types. You can specify MIME types via the constants in `ContentTypes`.

Note that the `obc` parameter must be an implementation of `OrderByComparator`. Although you can implement your own comparators, Liferay DXP already contains a few useful implementations in the package `com.liferay.document.library.kernel.util.comparator`:

- `RepositoryModelCreateDateComparator`: Sorts by creation date.
- `RepositoryModelModifiedDateComparator`: Sorts by modification date.
- `RepositoryModelReadCountComparator`: Sorts by number of views.
- `RepositoryModelSizeComparator`: Sorts by file size.
- `RepositoryModelTitleComparator`: Sorts by title.

As an example, this `getFileEntries` method contains all the above parameters except `fileEntryTypeId` (it contains `mimeTypes` instead):

```
List<FileEntry> getFileEntries(
        long repositoryId,
        long folderId,
        String[] mimeTypes,
        int start,
        int end,
        OrderByComparator<FileEntry> obc
)
```

Follow these steps to use this method to get a list of files. Note that the example in these steps gets all the PNG images from the root folder of a Site's default repository, sorted by title:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the method's arguments. You can do this any way you wish. As the next step describes, Liferay DXP provides constants and a comparator for all the arguments this example needs besides the group ID. This example gets the group ID by using `ParamUtil` with the request (`javax.portlet.ActionRequest`):

   ```
   long groupId = ParamUtil.getLong(actionRequest, "groupId");
   ```

   It's also possible to get the group ID via the `ThemeDisplay`. Calling the `ThemeDisplay` method `getScopeGroupId()` gets the ID of your app's current site (group):

   ```
   ThemeDisplay themeDisplay = (ThemeDisplay) request.getAttribute(WebKeys.THEME_DISPLAY);
   long groupId = themeDisplay.getScopeGroupId();
   ```

   For more information, see the Data Scopes tutorial.

3. Use the data from the previous step to call the service reference method you want to use to get the files. This example calls the above `getFileEntries` method with the group ID from the previous step, and constants and a comparator for the remaining arguments:

   ```
   List<FileEntry> fileEntries =
           _dlAppService.getFileEntries(
                   groupId,
                   DLFolderConstants.DEFAULT_PARENT_FOLDER_ID,
                   new String[] {ContentTypes.IMAGE_PNG},
                   QueryUtil.ALL_POS,
                   QueryUtil.ALL_POS,
                   new RepositoryModelTitleComparator<>()
           );
   ```

Here's a description of the arguments used in this example:

- `groupId`: Using the group ID as the repository ID specifies that the operation takes place in the default site repository.
- `DLFolderConstants.DEFAULT_PARENT_FOLDER_ID`: Uses the `DLFolderConstants` constant `DEFAULT_PARENT_FOLDER_ID` to specify the repository's root folder.
- `new String[] {ContentTypes.IMAGE_PNG}`: Uses the `ContentTypes` constant `IMAGE_PNG` to specify PNG images.
- `QueryUtil.ALL_POS`: Uses the `QueryUtil` constant `ALL_POS` for the start and end positions in the results. This specifies all results, bypassing pagination.
- `new RepositoryModelTitleComparator<>()`: Creates a new `RepositoryModelTitleComparator`, which sorts the results by title.

Remember, this is just one of many `getFileEntries` and `getGroupFileEntries` methods. To see all such methods, see the `DLAppService` Javadoc.

Related Topics

Getting Started with the Documents and Media API
Getting Folders
Getting Multiple Entity Types

127.2 Getting Folders

The Documents and Media API can get folders in a similar way to getting files. The main difference is that folder retrieval methods may have an additional argument to tell the system whether to include *mount folders*. Mount folders are mount points for external repositories (e.g. Alfresco or SharePoint) that appear as regular folders in a Site's default repository. They let users navigate seamlessly between repositories. To account for this, some folder retrieval methods include the boolean parameter `includeMountFolders`. Setting this parameter to `true` includes mount folders in the results, while omitting it or setting it to `false` excludes them.

For example, to get a list of a parent folder's subfolders from a repository, including any mount folders, use the `getFolders` method:

```
getFolders(long repositoryId, long parentFolderId, boolean includeMountFolders)
```

Follow these steps to use this method to get the folders from a parent folder. Note that the example in these steps gets the folders in the default Site repository's root folder:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the method's arguments any way you wish. This `getFolders` method needs a repository ID, a parent folder ID, and a boolean value that indicates whether to include mount folders in the results. To specify the default site repository, you can use the group ID as the repository ID. This example gets the group ID from the request (`javax.portlet.ActionRequest`) via `ParamUtil`:

   ```
   long groupId = ParamUtil.getLong(actionRequest, "groupId");
   ```

   It's also possible to get the group ID via the `ThemeDisplay`. Calling the `ThemeDisplay` method `getScopeGroupId()` gets the ID of your app's current site (group). For more information, see the Data Scopes tutorial.

   ```
   ThemeDisplay themeDisplay = (ThemeDisplay) request.getAttribute(WebKeys.THEME_DISPLAY);
   long groupId = themeDisplay.getScopeGroupId();
   ```

   Note that getting the parent folder ID isn't necessary because this example uses the root folder, for which Liferay DXP provides a constant. Also, the boolean value can be provided directly—it doesn't need to be retrieved from somewhere. For more information on getting repository and folder IDs, see the getting started tutorial's sections on specifying repositories and folders.

3. Call the service reference's `getFolders` method with the data from the previous step and any other values you want to provide. Note that this example uses `DLFolderConstants.DEFAULT_PARENT_FOLDER_ID` to specify the repository's root folder as the parent folder. It also uses `true` to include any mount folders in the results:

   ```
   _dlAppService.getFolders(groupId, DLFolderConstants.DEFAULT_PARENT_FOLDER_ID, true)
   ```

Note that this is one of many methods you can use to get folders. The rest are listed in the `DLAppService` Javadoc.

Related Topics

Getting Started with the Documents and Media API
    Getting Files
    Getting Multiple Entity Types

127.3 Getting Multiple Entity Types

There are also methods in the Documents and Media API that retrieve lists containing several entity types. These methods use many of the same parameters as those already described for retrieving files and folders. For example, this method gets files and shortcuts from a given repository and folder. The `status` parameter specifies a workflow status. As before, the `start` and `end` parameters control pagination of the entities:

```
getFileEntriesAndFileShortcuts(long repositoryId, long folderId, int status, int start, int end)
```

To see all such methods, see the `DLAppService` Javadoc.

Follow these steps to use the above `getFileEntriesAndFileShortcuts` method. Note that the example in these steps gets all the files and shortcuts in the default Site repository's root folder:

1. Get a reference to `DLAppService`:

   ```
   @Reference
   private DLAppService _dlAppService;
   ```

   For more information on this, see the section on getting a service reference in the getting started tutorial.

2. Get the data needed to populate the method's arguments any way you wish. To specify the default Site repository, you can use the group ID as the repository ID. This example gets the group ID from the request (`javax.portlet.ActionRequest`) via `ParamUtil`:

   ```
   long groupId = ParamUtil.getLong(actionRequest, "groupId");
   ```

   Getting the parent folder ID, workflow status, and start and end parameters isn't necessary because Liferay DXP provides constants for them. The next step shows this in detail.

3. Call the service reference method with the data from the previous step and any other values you want to provide. This example calls `getFileEntriesAndFileShortcuts` with the group ID from the previous step and constants for the remaining arguments:

   ```
   _dlAppService.getFileEntriesAndFileShortcuts(
           groupId,
           DLFolderConstants.DEFAULT_PARENT_FOLDER_ID,
           WorkflowConstants.STATUS_APPROVED,
           QueryUtil.ALL_POS,
           QueryUtil.ALL_POS
   )
   ```

   Here's a description of the arguments used in this example:

   - `groupId`: Using the group ID as the repository ID specifies that the operation takes place in the default site repository.
   - `DLFolderConstants.DEFAULT_PARENT_FOLDER_ID`: Uses the `DLFolderConstants` constant `DEFAULT_PARENT_FOLDER_ID` to specify the repository's root folder.
   - `WorkflowConstants.STATUS_APPROVED`: Uses the `WorkflowConstants` constant `STATUS_APPROVED` to specify only files/folders that have been approved via workflow.
   - `QueryUtil.ALL_POS`: Uses the `QueryUtil` constant `ALL_POS` for the start and end positions in the results. This specifies all results, bypassing pagination.

Related Topics

Getting Started with the Documents and Media API
    Getting Files
    Getting Folders

CHAPTER 128

ADAPTIVE MEDIA

The Adaptive Media app lets administrators tailor the quality of images to the device viewing those images. For information on using this app, see the Adaptive Media user guide.

If you want to leverage Adaptive Media in your own app, you're in the right place! The tutorials here explain how to use adapted images in your app. You'll also learn how to change Adaptive Media's image processing.

Onwards!

128.1 Displaying Adapted Images in Your App

To display adapted images in your apps, Adaptive Media offers a convenient tag library 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.

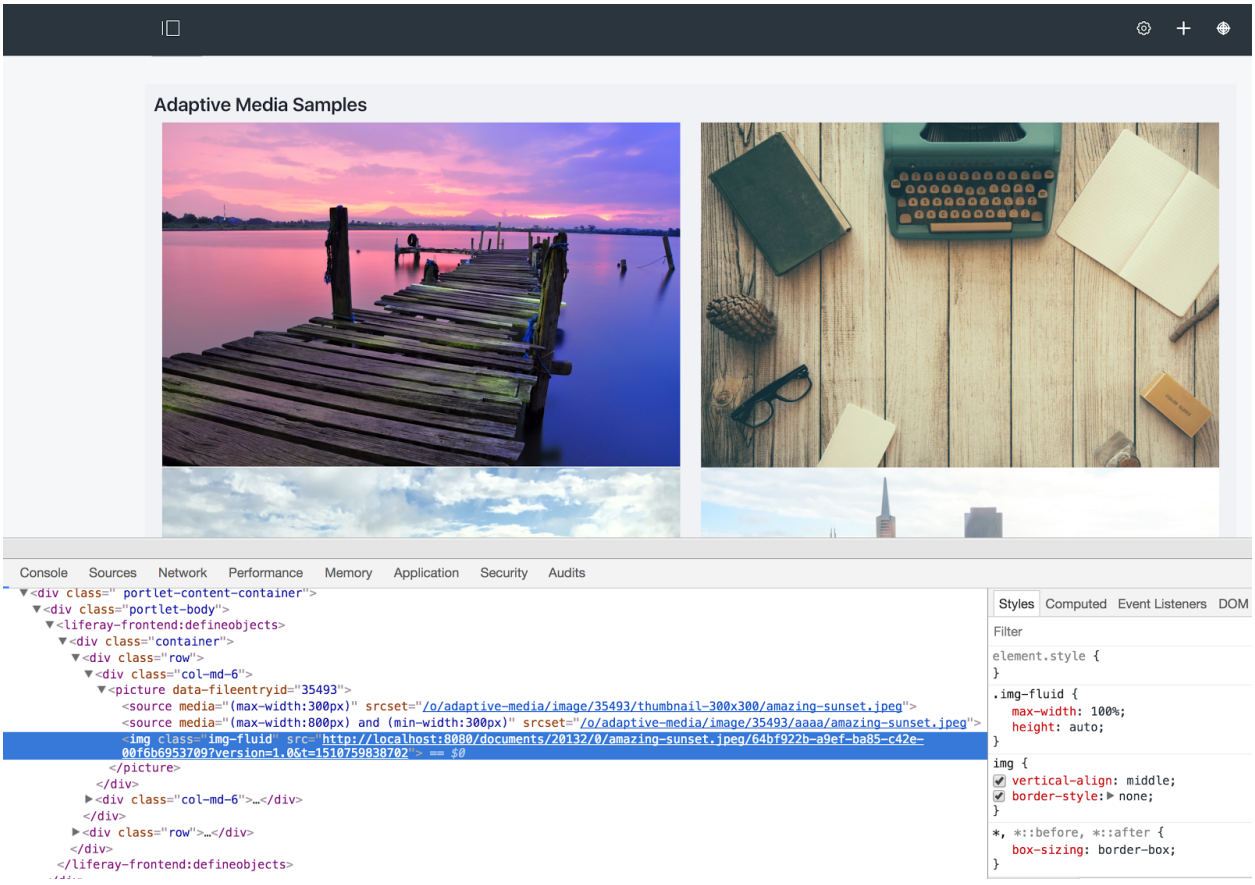


Figure 128.1: The Adaptive Media Samples app shows all the site's adapted images.

Well done! Now you know how to display adapted images in your app.

Related Topics

Finding Adapted Images
    Changing Adaptive Media's Image Scaling
    Adapting Your Media Across Multiple Devices

128.2 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_HEIGHT` 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 Scaling
Adapting Your Media Across Multiple Devices

128.3 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 129

SOCIAL API

The social API lets users interact with content throughout the portal, including content in your applications. For example, users can provide feedback on content, share that content with others, subscribe to receive notifications, and more. These features let users 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 this social API to enable these features in your own app.

129.1 Applying Social Bookmarks

When you enable social bookmarks, icons for sharing on Twitter, Facebook, and Google Plus appear below your application's content. Taglibs provide the markup you need to add this feature to your app.

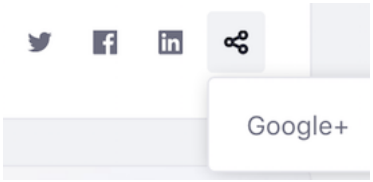


Figure 129.1: Social bookmarks are enabled in the built-in Blogs portlet

Follow these steps to add social bookmarks to your app:

1. Make sure your entity is asset enabled.
2. In your project's `build.gradle` file, add a dependency to the module `com.liferay.social.bookmarks.taglib`:

   ```
   compileOnly group: "com.liferay", name: "com.liferay.social.bookmarks.taglib", version: "1.0.0"
   ```

3. Choose a view in which to show the social bookmarks. For example, you can display them in one of your portlet's views.

```
**Note:** You don't need to implement social bookmarks in your portlet's
[asset renderers](/docs/7-1/tutorials/-/knowledge_base/t/rendering-an-asset).
The Asset Publisher displays social bookmarks in asset renderers by default.
```

---

4. In your view's JSP, include the `liferay-social-bookmarks` taglib declaration:

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

5. Get an instance of your entity. You can do this however you wish. This example uses `ParamUtil` to get the entity's ID from the render request, then uses the entity's `-LocalServiceUtil` class to create an entity object:

```
<%
long entryId = ParamUtil.getLong(renderRequest, "entryId");
entry = EntryLocalServiceUtil.getEntry(entryId);
%>
```

6. Use the `liferay-social-bookmarks:bookmarks` tag to add the social bookmarks component. Its attributes are described below:

`className`: The entity's class name.

`classPK`: The Java class primary key of the entity.

`displayStyle`: The display style of the social bookmarks. Possible values are `inline`, which displays them in a row, and `menu`, which hides them in a menu.

`title`: A title for the content being shared.

`types`: A comma-delimited list of the social media services to use (e.g., `facebook,twitter`). To use every social media service available in the portal, omit this attribute or use `<%= null %>` for its value.

`url`: A URL to the portal content being shared.

Here's an example of using the `liferay-social-bookmarks:bookmarks` tag to add social bookmarks for a blog entry in the Blogs app:

```
<liferay-social-bookmarks:bookmarks
        className="<%= BlogsEntry.class.getName() %>"
        classPK="<%= entry.getEntryId() %>"
        displayStyle="inline"
        title="<%= entry.getTitle() %>"
        types="facebook,twitter"
        url="<%= PortalUtil.getCanonicalURL(bookmarkURL.toString(), themeDisplay, layout) %>"
/>
```

The `displayStyle` in this example is set to `inline`. The screenshot at the beginning of this tutorial shows what this looks like. The first three social bookmarks appear in a row, and the rest appear in the *Share* menu. If you use `menu` instead, all the social bookmarks appear in the menu.

The `title` is retrieved by the entry's `getTitle()` method. Keep in mind that you should retrieve your entity's title with the appropriate method for that entity.

Note that the `PortalUtil` method `getCanonicalURL` is called for the `url`. This method constructs an SEO-friendly URL from the page's full URL. For more information, see the method's Javadoc.

Related Topics

Asset Framework

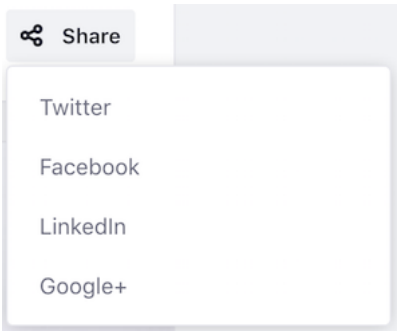


Figure 129.2: With `displayStyle` set to `menu`, the social bookmarks all appear in the *Share* menu.

129.2 Creating Social Bookmarks

Applying social bookmarks lets you link your app's content to the social networks of your choice. By default, Liferay DXP supports social bookmarks for Twitter, Facebook, LinkedIn, and Google Plus. You can also create your own social bookmark by registering a component that implements the `SocialBookmark` interface from the module `com.liferay.social.bookmarks.api`. This tutorial shows you how to do this.

Implementing the SocialBookmark Interface

Follow these steps to implement the `SocialBookmark` interface:

1. Create your `*SocialBookmark` class and register a component that defines the `social.bookmarks.type` property. This property's value is what you enter for the `liferay-social-bookmarks:bookmarks` tag's `type` attribute when you use your social bookmark.

   For example, here's the definition for a Twitter social bookmark class:

   ```
   @Component(immediate = true, property = "social.bookmarks.type=twitter")
   public class TwitterSocialBookmark implements SocialBookmark {...
   ```

2. Create a `ResourceBundleLoader` reference to help localize the social bookmark's name.

   ```
   @Reference(
           target = "(bundle.symbolic.name=com.liferay.social.bookmark.twitter)"
   )
   private ResourceBundleLoader _resourceBundleLoader;
   ```

3. Implement the `getName` method to return the social bookmark's name as a `String`. This method takes a `Locale` object that you can use for localization via `LanguageUtil` and `ResourceBundle`:

   ```
   @Override
   public String getName(Locale locale) {
       ResourceBundle resourceBundle = _resourceBundleLoader.loadResourceBundle(locale);

       return LanguageUtil.get(resourceBundle, "twitter");
   }
   ```

4. Implement the `getPostURL` method to return the share URL. This method constructs the share URL from a title and URL, and uses `URLCodec` to encode the title in the URL:

```
@Override
public String getPostURL(String title, String url) {
    return String.format(
        "https://twitter.com/intent/tweet?text=%s&tw_p=tweetbutton&url=%s",
        URLCodec.encodeURL(title), url);
}
```

5. Create a `ServletContext` reference:

```
@Reference(
        target = "(osgi.web.symbolicname=com.liferay.social.bookmark.twitter)"
)
private ServletContext _servletContext;
```

6. Implement the `render` method, which is called when the inline display style is selected. Typically, this method renders a link to the share URL (e.g., a share button), but you can use it for whatever you need. To keep a consistent look and feel with the default social bookmarks, you can use a Clay icon.

   This example gets a `RequestDispatcher` for the JSP that contains a Clay icon (`page.jsp`), and then includes that JSP in the response:

```
@Override
public void render(
                String target, String title, String url, HttpServletRequest request,
                HttpServletResponse response)
        throws IOException, ServletException {

        RequestDispatcher requestDispatcher =
                _servletContext.getRequestDispatcher("/page.jsp");

        requestDispatcher.include(request, response);
}
```

Next, you'll see an example of how to create a `page.jsp` file.

Creating Your JSP

The `page.jsp` file referenced in the above `SocialBookmark` implementation uses a Clay link (`clay:link`) to specify and style the Twitter icon included with Clay. Follow these steps to create a JSP for your own social bookmark:

1. Add the `clay` and `liferay-theme` taglib declarations:

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

2. Import `GetterUtil` and `SocialBookmark`:

```
<%@ page import="com.liferay.portal.kernel.util.GetterUtil" %>
<%@ page import="com.liferay.social.bookmarks.SocialBookmark" %>
```

3. From the request, get a `SocialBookmark` instance and the social bookmark's title and URL:

```
<%
SocialBookmark socialBookmark = (SocialBookmark)request.getAttribute("liferay-social-bookmarks:bookmark:socialBookmark");
String title = GetterUtil.getString((String)request.getAttribute("liferay-social-bookmarks:bookmark:title"));
String url = GetterUtil.getString((String)request.getAttribute("liferay-social-bookmarks:bookmark:url"));
%>
```

   The title and URL are set via the `liferay-social-bookmarks` tag library when applying the social bookmark.

4. Add the Clay link. This example sets the following `clay:link` attributes:

   - `buttonStyle`: This example renders The button's type as a secondary button.
   - `elementClasses`: The custom CSS to use for styling the button (optional).
   - `href`: The button's URL. You should specify this by calling your `SocialBookmark` instance's `getPostURL` method.
   - `icon`: The button's icon. This example specifies the Twitter icon included in Clay (`twitter`).
   - `title`: The button's title. This example uses the `SocialBookmark` instance's `getName` method.

   ```
   <clay:link
           buttonStyle="secondary"
           elementClasses="btn-outline-borderless btn-sm lfr-portal-tooltip"
           href="<%= socialBookmark.getPostURL(title, url) %>"
           icon="twitter"
           title="<%= socialBookmark.getName(locale) %>"
   />
   ```

To see a complete, real-world example of a social bookmark implementation, see Liferay's Twitter social bookmark code.

Related Topics

Applying Social Bookmarks

Using the Clay Taglib in Your Portlets

129.3 Adding Comments to Your App

Letting users comment on content makes your app come alive. Taglibs provide the markup you need to add this feature. This tutorial shows you how to use these taglibs to enable comments.

These steps use a sample Guestbook app as an example:

1. Make sure your entity is asset enabled.
2. Choose a read-only view of the entity you want to enable comments on. You can display the comments component in your app's view, or if you've implemented asset rendering you can display it in the full content view in the Asset Publisher app.

3. Include the liferay-ui, liferay-comment, and portlet taglib declarations in your JSP:

```
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>
<%@ taglib prefix="liferay-comment" uri="http://liferay.com/tld/comment" %>
<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet_2_0" %>
```

4. Use ParamUtil to get the entity's ID from the render request. Then create an entity object using the -LocalServiceUtil class. Here's an example that does this for a guestbook entry in the example Guestbook app:

```
<%
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 lets users hide the discussion area:

```
<liferay-ui:panel-container extended="<%=false%>"
  id="guestbookCollaborationPanelContainer" persistState="<%=true%>">
  <liferay-ui:panel collapsible="<%=true%>" extended="<%=true%>"
    id="guestbookCollaborationPanel" persistState="<%=true%>"
    title="Collaboration">
```

6. Create a URL for the discussion using the portlet:actionURL tag:

```
<portlet:actionURL name="invokeTaglibDiscussion" var="discussionURL" />
```

7. Use the liferay-comment:discussion tag to add the discussion. To let the user return to the JSP after making a comment, set the tag's redirect attribute to the current URL. You can use PortalUtil.getCurrentURL((renderRequest)) to get the current URL from the request object. In this example, the current URL was earlier set to the currentURL variable:

```
    <liferay-comment: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, see the tutorial on Configuring JSP Templates.

Great! Now you know how to let users comment on content in your asset enabled portlets.

Related Topics

Asset Framework

Rating Assets

129.4 Rating Assets

The asset framework supports a system that lets users rate content in apps. This feature appears in many of Liferay DXP's built-in apps. For example, users can rate articles published in the Blogs app. Using taglibs, you can enable ratings for your app's content in only a few lines of code. This tutorial shows you how.

Figure 129.3: Ratings let users quickly provide feedback on content.

Follow these steps to enable ratings in your app. Note that these steps use the Guestbook app as an example. As its name implies, this app lets users leave simple messages in a guestbook.

1. Make sure your entity is asset enabled.
2. Choose a read-only view of the entity for which you want to enable ratings. You can display ratings 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 app.
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 create an entity object using the `-LocalServiceUtil` class. Here's an example that does this for a guestbook entry in the example Guestbook app:

   ```
   <%
   long entryId = ParamUtil.getLong(renderRequest, "entryId");
   entry = EntryLocalServiceUtil.getEntry(entryId);
   %>
   ```

5. Use the `liferay-ui:ratings` tag to add the ratings component for the entity:

   ```
   <liferay-ui:ratings className="<%=Entry.class.getName()%>"
       classPK="<%=entry.getEntryId()%>" type="stars" />
   ```

   The `type` attribute specifies the rating system to use:

   - `like`: Likes
   - `stars`: Stars (five, by default)
   - `thumbs`: Thumbs up/down (as shown in the above screenshot)

You can also make the rating type configurable by administrators. To do this, see the tutorial Implementing Ratings Type Selection and Value Type Transformation.

Great! Now you know how to let users rate content in your asset-enabled apps.

Related Topics

Asset Framework
Implementing Ratings Type Selection and Value Transformation

129.5 Implementing Ratings Type Selection and Value Transformation

Liferay DXP has three different mechanisms for rating content:

- Likes
- Stars
- Thumbs (up/down)

Prior to 7.0, there was no way for portal or site administrators to select a rating type–it was hard-coded in each app. In 7.0 and later, admins can select the rating type for an app's entities via the Control Panel and Site Administration:

- **Portal admins:** can set the default rating type for the portal
- **Site admins:** can override the default rating type for their site

All Liferay apps leverage this feature. Your apps can too: this tutorial shows you how.

Specifying an Entity's Rating Type

A custom app that uses ratings must define its rating type in an OSGi component that implements the `PortletRatingsDefinition` interface. This class declares the usage of ratings (specifying the portlet and the entity) and the default rating type (that can be overridden by portal and site admins).

Follow these steps to implement `PortletRatingsDefinition` to define your app's rating type:

1. Register the class as an OSGi component and set the `model.class.name` property to the fully qualified class name of the class that will use this rating definition. For example, this example rating definition is for a blog entry, so the `model.class.name` property is set to `com.liferay.portlet.blogs.model.BlogsEntry`:

```
@Component(
    property = {
        "model.class.name=com.liferay.portlet.blogs.model.BlogsEntry"
    }
)
public class BlogsPortletRatingsDefinition implements PortletRatingsDefinition {...
```

2. The `PortletRatingsDefinition` interface has two methods that you must implement:

- `getDefaultRatingsType`: returns the entity's default rating type, which portal and site admins can override. You can do this via the `RatingsType` enum, which lets you use `LIKE`, `STARS`, or `THUMBS` to set the rating type:

```
@Override
public RatingsType getDefaultRatingsType() {
    return RatingsType.THUMBS;
}
```

- `getPortletId`: returns the portlet ID of the main portlet that uses the entity. You can do this via the `PortletKeys` enum, which defines many constants that correspond to the portlet IDs of the built-in portlets. This example specifies the Blogs portlet:

```
@Override
public String getPortletId() {
    return PortletKeys.BLOGS;
}
```

Next, you'll learn how to transform values between rating types.

Transforming Ratings Values Between Rating Types

The rating values are stored in the database as normalized values. This permits switching among different rating types without modifying the underlying data. When administrators change an entity's rating type, its best match is computed. Here's a list of the default transformations between rating types:

1. When changing from stars to:

    - **Like:** A value of 3, 4, or 5 stars is considered a like; a value of 1 or 2 stars is omitted.
    - **Thumbs up/down:** A value of 3, 4, or 5 stars is considered a thumbs up; a value of 1 or 2 stars is considered a thumbs down.

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 some cases, however, where you want to apply different criteria to determine the new rating values. A mechanism exists to let you do this, but it modifies the stored rating values. To define such transformations, create an OSGi component that implements the `RatingsDataTransformer` interface.

Note: The portal doesn't provide a default RatingsDataTransformer implementation. Unless you provide such an implementation, the stored rating values always remain the same while the portal interprets existing values for the selected rating type.

When implementing RatingsDataTransformer, implement the transformRatingsData method to transform the data. This method's arguments include the RatingsType variables fromRatingsType and toRatingsType, which contain the rating type to transform from and to, respectively. These values let you write your custom transformation's logic. You can write this logic by implementing the interface ActionableDynamicQuery.PerformActionMethod as an anonymous inner class in the transformRatingsData method, implementing the performAction method with your transformation's logic.

For example, follow these steps to implement a RatingsDataTransformer:

1. Create an OSGi component class that implements RatingsDataTransformer:

```
@Component
public class DummyRatingsDataTransformer implements RatingsDataTransformer {...
```

2. In this class, implement the transformRatingsData method. Note that it contains the RatingsType variables fromRatingsType and toRatingsType:

```
@Override
public ActionableDynamicQuery.PerformActionMethod transformRatingsData(
        final RatingsType fromRatingsType, final RatingsType toRatingsType)
    throws PortalException {

}
```

3. In the transformRatingsData method, implement the interface ActionableDynamicQuery.PerformActionMethod as an anonymous inner class:

```
return new ActionableDynamicQuery.PerformActionMethod() {

};
```

4. In the anonymous ActionableDynamicQuery.PerformActionMethod implementation, implement the performAction method to perform your transformation. This example irreversibly transforms the rating type from like to stars, resetting the value to 0. The if statement uses the fromRatingsType and toRatingsType values to specify that the transformation only occurs when going from likes to stars. The transformation is performed via RatingsEntry and its -LocalServiceUtil. After getting a RatingsEntry object, its setScore method sets the rating score to 0. The RatingsEntryLocalServiceUtil method updateRatingsEntry then updates the RatingsEntry in the database:

```
@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);
            }
        }
```

Here's the complete class for this example:

```
@Component
public class DummyRatingsDataTransformer 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.

Nice work! Now you know how to set an entity's rating type. You also know how to implement a rating data transformer. We salute you with a thumbs up!

Related Topics

Asset Framework
    Rating Assets

129.6 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 each other. Unfortunately, sometimes people have a bad day and decide to take their frustrations out in inappropriate posts. No worries though, the asset framework supports a system for flagging content in apps. Letting users flag inappropriate content takes much of the work off site administrators.

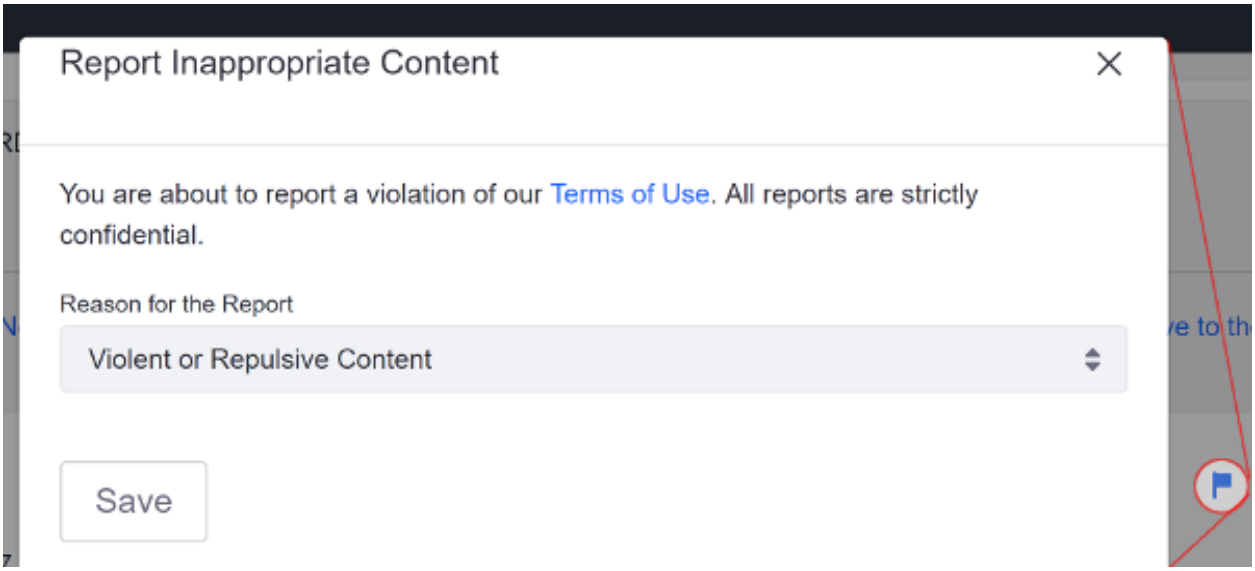


Figure 129.4: Flags for letting users mark objectionable content are enabled in the Message Boards portlet.

This tutorial shows you how to enable content flagging in a portlet.
Follow these steps to enable content flagging in your app:

1. Make sure your entity is asset enabled.
2. Choose a read-only view of the entity you want to enable flags on. You can display flags in one of your app's views, or if you've implemented asset rendering you can display it in the full content view in the Asset Publisher app.
3. In your JSP, include the `liferay-flags` taglib declaration:

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

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 tag `liferay-flags:flags` 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 apps.

Related Topics

Asset Framework
    Rating Assets

Chapter 130

Export/Import and Staging

The 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 gives 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. 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 to frequently store a complete record of all data given during a course. The institution offering the course must 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 let you 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 User Guide section.

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. 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. 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, you can also access APIs that let you 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: 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.

130.1 Decision to Implement Staging

Staging 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, Bookmarks, etc.) 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, 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 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.

130.2 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 `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.

To step through the process for leveraging the Model Adapter Builder for an existing app, visit the Creating Staged Models Manually tutorial.

130.3 Generating Staged Models Using Service Builder

This document has been updated and ported to Liferay Learn and is no longer maintained here.

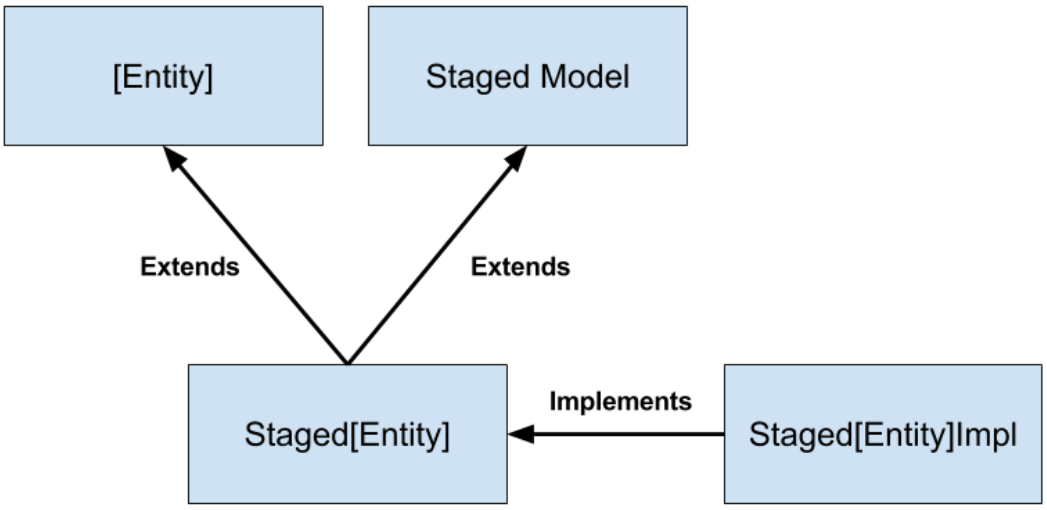


Figure 130.1: The Staged Model Adapter class extends your entity and staged model interfaces.

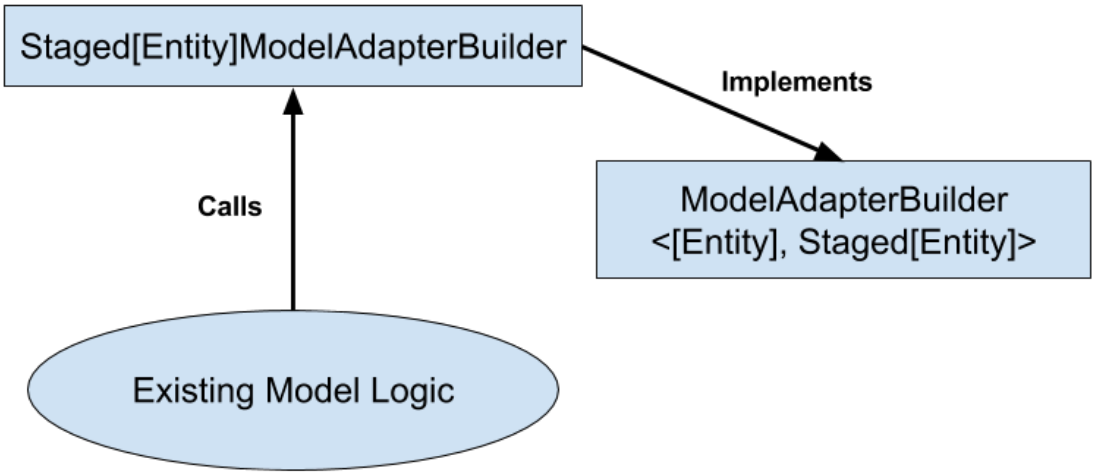


Figure 130.2: The Model Adapter Builder gets an instance of the model and outputs a staged model.

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.

130.4 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!

130.5 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 is 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.

The data handler framework is available 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.

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.

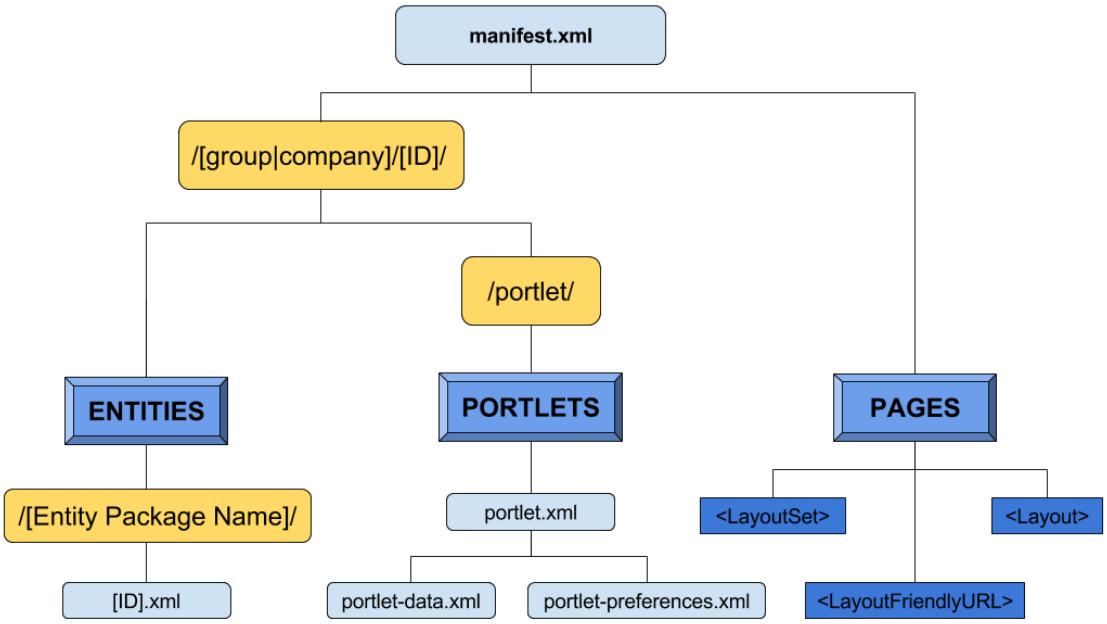


Figure 130.3: Entities, Portlets, and Pages are defined in a LAR in different places.

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 more information on staged models, see the Understanding Staged Models tutorial.

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.

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

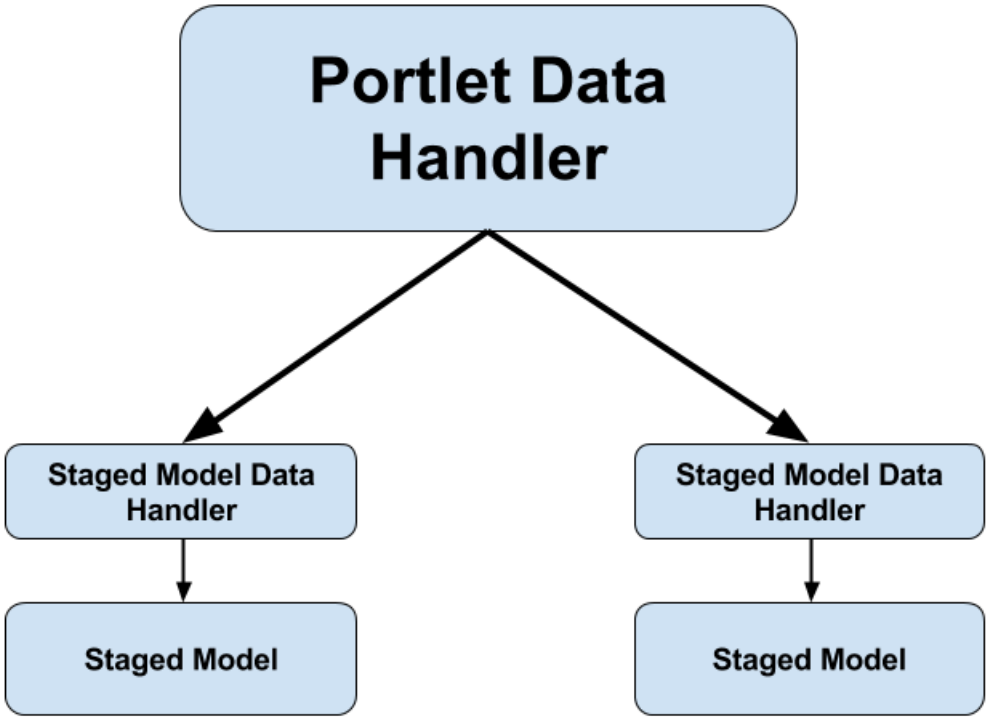


Figure 130.4: The Data Handler framework uses portlet data handlers and staged model data handlers to track and export/import portlet and staged model information, respectively.

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 Portlet Data Handlers and Developing Staged Model Data Handlers tutorials.

130.6 Developing Portlet Data Handlers

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. In this tutorial, you'll create a Portlet Data Handler for a Bookmarks application.

Note: You must ensure your application is properly configured to use data handlers. For more information on how to do this, see the Data Handler Fundamentals section.

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.

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 `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. For 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;
```

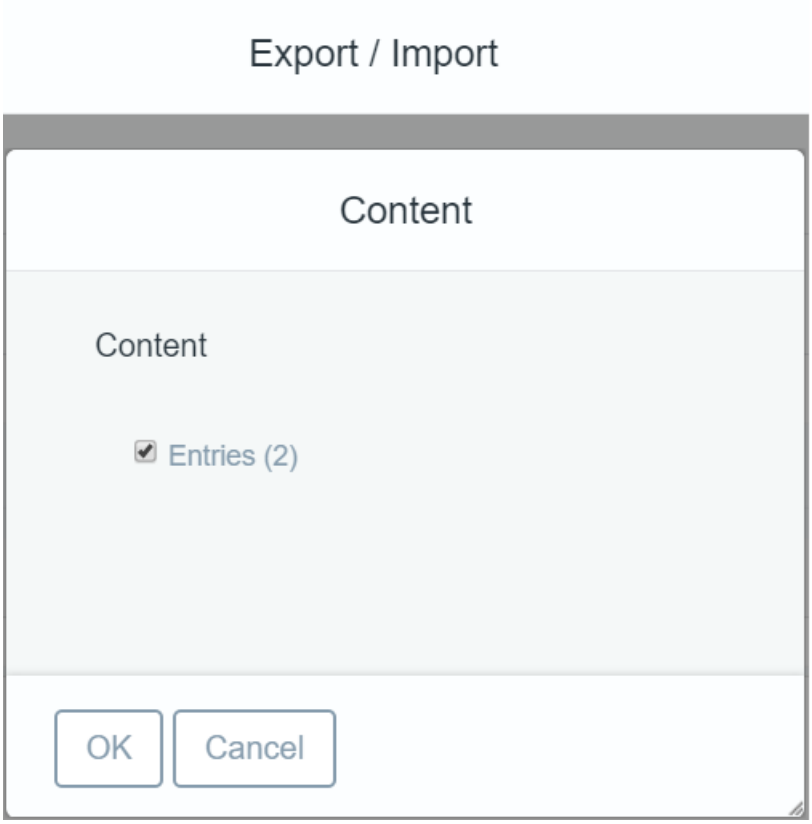


Figure 130.5: You can select the content types you'd like to export/import in the UI.

The set methods must be annotated with the @Reference annotation. Visit the Invoking Local Services tutorial for more information on using the `@Reference` annotation in Liferay DXP.

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 a new framework, but is a viable option when setting up your portlet data handlers. For more information on this, see the Providing Entity-Specific Local Services for Staging tutorial section. Since local services are more widely used in custom apps, this tutorial covers those instead.

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(
        BookmarksConstants.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(
        BookmarksConstants.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.getBooleanParam` method checks if the user selected Bookmarks entries for export. Later, the `ExportImportActionableDynamicQue`

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.

```
**Note:** This is a legacy feature that was useful when deletions were not
propagated between Sites. This cleaned the portlet's data, allowing you to
see everything associated with the portlet during every publication. It's
unnecessary now that Staging can recognize deletions across all Sites. It's,
however, still offered as a feature of Staging and is implemented in
Liferay's Bookmarks app, so it's included here.
```

---

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();
}
```

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.

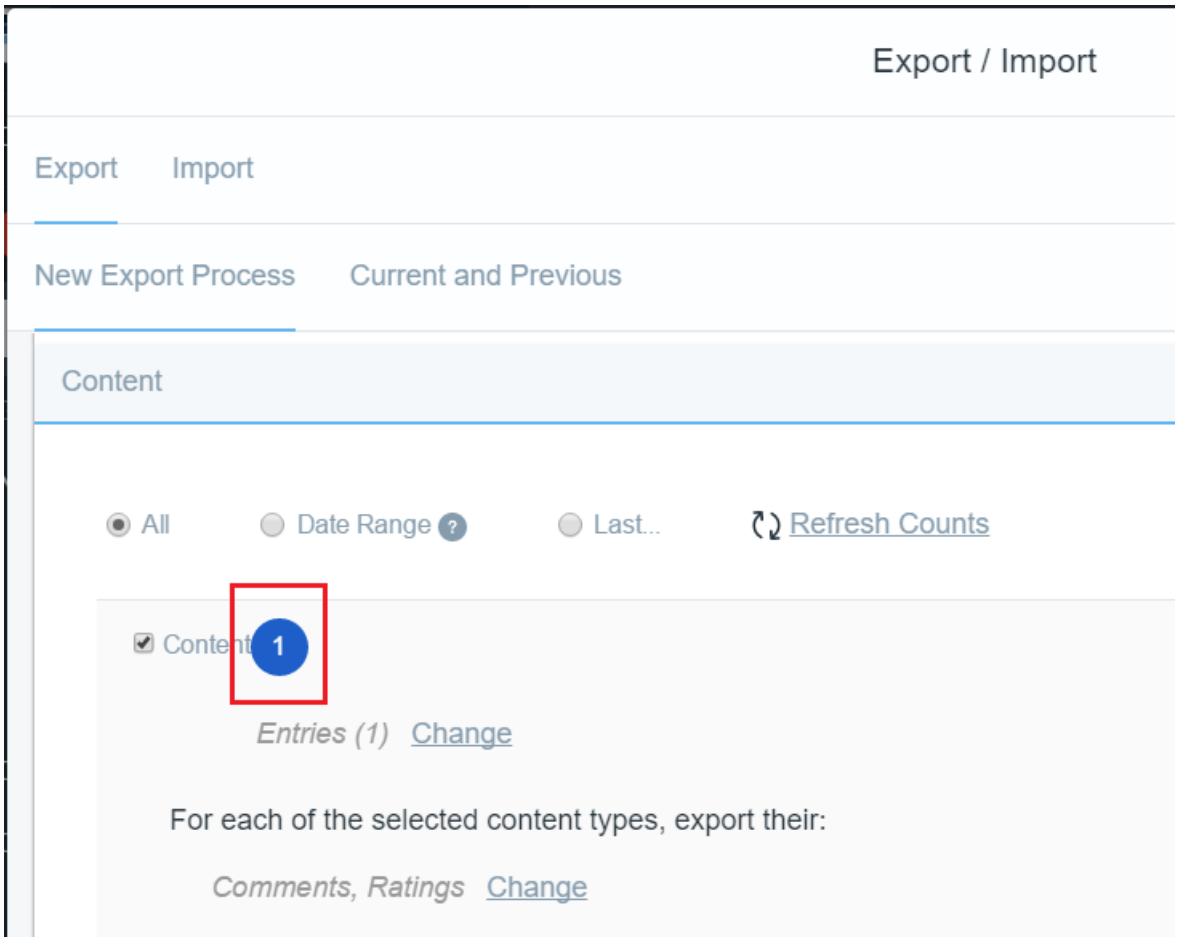


Figure 130.6: The number of modified Bookmarks entities are displayed in the Export UI.

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;
}

@Override
public boolean validateSchemaVersion(String schemaVersion) {
    return _portletDataHandlerHelper.validateSchemaVersion(
        schemaVersion, getSchemaVersion());
}
```

The schema version is used to perform component related validation before importing data. It's added to the LAR file for each application being processed. During import, the environment's schema version is compared to the LAR file's schema version. Validating the schema version avoids broken data when importing. See the PortletDataHandler.getVersionScheme() method's Javadoc for more information.

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. Be sure to also implement staged model data handlers for your staged models. See the Developing Staged Model Data Handlers to do this for the Bookmarks app.

130.7 Developing Staged Model Data Handlers

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. In this tutorial, you'll learn how to create a Staged Model Data Handler for a Bookmarks application.

Note: You must ensure your application is properly configured to use data handlers. For more information on how to do this, see the Data Handler Fundamentals section.

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`.

This tutorial assumes you've already created staged models. 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 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 `bookmarks-service` module's `com.liferay.bookmarks.internal.exportimport.data.handler` package.

2. Create a `-StagedModelDataHandler` class in the `-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> {
```

3. 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.
```

---

4. 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 a new framework, but is a viable option when setting up your staged model data handlers. For more information on this, see the Providing Entity-Specific Local Services for Staging tutorial section. Since local services are more widely used in custom apps, this tutorial covers those instead.

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 = {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.

6. 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.

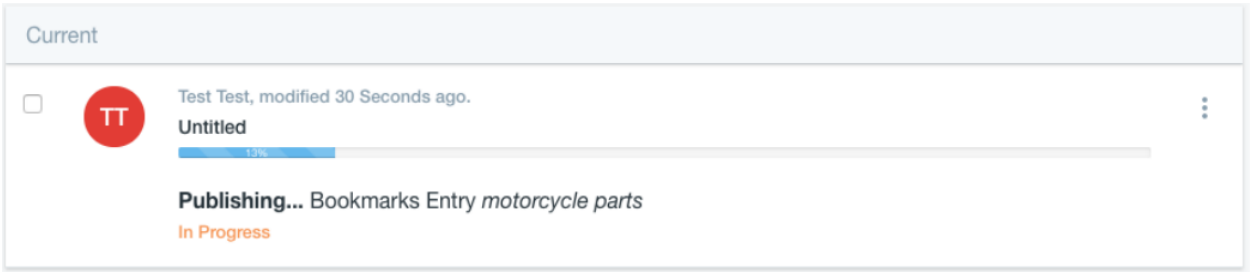


Figure 130.7: Your staged model data handler provides the display name in the Export/Import UI.

7. 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.ge
        }
    }
    else {
        importedEntry = _bookmarksEntryLocalService.addEntry(userId, portletDataContext.getScopeGroupId(), folderId,entry.getName(), entry.ge
    }

    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.

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 bookmarks folder 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 have references from two separate Sites with differing IDs, the system can't match them during publication. For example, suppose you export a bookmark 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 bookmarks folder 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 {

    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. Be sure to also implement a portlet data handler to manage portlet specific data. See the Developing Portlet Data Handlers. to do this for the Bookmarks app.

CHAPTER 131

PROVIDING ENTITY-SPECIFIC LOCAL SERVICES FOR STAGING

When creating your data handlers, you must leverage your app's local services to perform Staging-related tasks for its entities. When the Staging framework operates on entities (i.e., staged models), it often cannot manage important information from the entity's local services alone; instead, you're forced to reinvent basic functionality so the Staging framework can access it. This is caused by services not sharing a common ancestor (i.e., interface or base class).

The *Staged Model Repository* framework removes this barrier by linking an app's staged model to a local service. This lets the Staging framework call a staged model repository independently based on the entity being processed. This gives you access to entity-specific methods tailored specifically for the staged model data you're handling.

What kind of *entity-specific* methods are we talking about here? Your data handlers only expose a specific set of actions, like export and import methods. The Staged Model Repository framework provides CRUD operations for a specific staged model that are not exposed using local services.

The staged model repository does not avoid using your app's local services. It only provides an additional layer that provides Staging-specific functionality. So how does this work? A brief Staging process is outlined below:

- `*StagedModelDataHandler` de-serializes the provided LAR file's XML into a model.
- `*StagedModelRepository` updates the model based on the environment and business logic, providing entity-specific CRUD operations for Staging purposes (e.g., UUID manipulation).
- Local services are called from the `*StagedModelRepository` and handles the remainder of the process.

Pretty cool, right? Read on to learn how to implement and use the Staged Model Repository framework in your app.

131.1 Implementing the Staged Model Repository Framework

Providing specialized local services for your app's staging functionality lets you abstract the additional staging-specific information away from your data handlers. Before you can begin using the Staged Model Repository framework in your app, you must implement it.

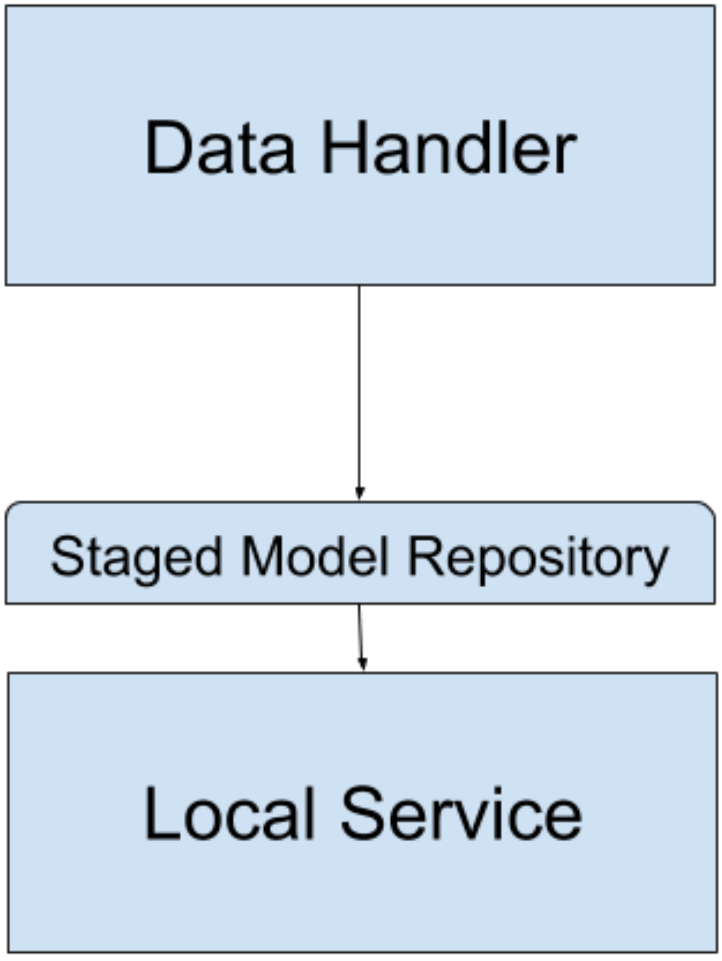


Figure 131.1: Staged Model Repositories provide a Staging-specific layer of functionality for your local services.

Below is a quick example that demonstrates implementing the `StagedModelRepository` interface to use for a staged model. This example references Liferay's Bookmarks app and Bookmarks Entry entities.

1. In your app's `-service` bundle, create a package that holds your Staged Model Repository classes (e.g., `com.liferay.bookmarks.exportimport.staged.model.repository`). If you do not have a `-service` bundle, visit the Service Builder tutorials for info on generating an app's services. You must have them to leverage most Staging features.

2. Create your `-StagedModelRepository` class in the new package and implement the `StagedModelRepository` interface in the class' declaration. For example,

   ```
   public class BookmarksEntryStagedModelRepository
       implements StagedModelRepository<BookmarksEntry> {
   ```

   Be sure also to include the staged model type parameter for this repository (e.g., `BookmarksEntry`).

3. Add an `@Component` annotation for your staged model repository class that looks like this:

   ```
   @Component(
       immediate = true,
       property = "model.class.name=FULLY_QUALIFIED_MODEL_CLASS",
       service = StagedModelRepository.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 model class you wish to handle with this service so it's recognized by the data handlers leveraging it. You'll learn more about this later.
   - The `service` element should point to the `StagedModelRepository.class` interface.

   The `BookmarksEntryStagedModelRepository`'s `@Component` annotation looks like this:

   ```
   @Component(
       immediate = true,
       property = "model.class.name=com.liferay.bookmarks.model.BookmarksEntry",
       service = StagedModelRepository.class
   )
   ```

4. Implement the `StagedModelRepository` interface's methods in your staged model repository. You can reference the Javadoc for this interface to learn what each method is intended for.

   As an example, you'll step through a couple method implementations to get a taste for how it works.

   Implementing the `addStagedModel(...)` method for a Bookmarks entry looks like this:

```
@Override
public BookmarksEntry addStagedModel(
        PortletDataContext portletDataContext,
        BookmarksEntry bookmarksEntry)
    throws PortalException {

    long userId = portletDataContext.getUserId(
        bookmarksEntry.getUserUuid());

    ServiceContext serviceContext = portletDataContext.createServiceContext(
        bookmarksEntry);

    if (portletDataContext.isDataStrategyMirror()) {
        serviceContext.setUuid(bookmarksEntry.getUuid());
    }

    return _bookmarksEntryLocalService.addEntry(
        userId, bookmarksEntry.getGroupId(), bookmarksEntry.getFolderId(),
        bookmarksEntry.getName(), bookmarksEntry.getUrl(),
        bookmarksEntry.getDescription(), serviceContext);
}
```

This method sets the user ID and service context based on the portlet data context. The `PortletDataContext` is used to populate the LAR file with your application's data during the export process. Next it sets the UUID, which is required to differentiate staged content between Sites. Lastly, the entity's local service is called.

Just calling the `BookmarksEntryLocalService.addEntry(...)` method would not have been enough to satisfy the staged model data handler's needs (i.e., the UUID requirement). With the staged model repository layer, however, you can add staging specific requirements on top of the present local services to serve your data handlers' needs.

Not every method implementation requires additional staging information. For example, deleting Bookmarks Entries and deleting Bookmarks Entry staged models are functionally the same, so your staged model repository's method would look like this:

```
@Override
public void deleteStagedModels(PortletDataContext portletDataContext)
    throws PortalException {

    _bookmarksEntryLocalService.deleteEntries(
        portletDataContext.getScopeGroupId(),
        BookmarksFolderConstants.DEFAULT_PARENT_FOLDER_ID);
}
```

Since nothing additional is required for deleting staged models, the staged model repository calls the local service's `deleteEntries(...)` method with no additional changes.

Finish implementing the `StagedModelRepository` so it's usable in your data handlers.

Awesome! You've implemented the Staged Model Repository framework for your app! If you're interested in leveraging this framework after the implementation process, see the Using the Staged Model Repository Framework tutorial.

131.2 Using the Staged Model Repository Framework

Leveraging the Staged Model Repository framework in your app is easy once you've created staged model repository implementation classes.

You'll step through a quick example to demonstrate leveraging the `StagedModelRepository` interface in a staged model data handler. The code snippets originate from Liferay's Bookmarks app and Bookmarks Entries.

1. Create a getter and setter method to make a `StagedModelRepository` object available for the `BookmarksEntry` entity:

```
@Override
protected StagedModelRepository<BookmarksEntry> getStagedModelRepository() {
    return _stagedModelRepository;
}

@Reference(
    target = "(model.class.name=com.liferay.bookmarks.model.BookmarksEntry)",
    unbind = "-"
)
protected void setStagedModelRepository(
    StagedModelRepository<BookmarksEntry> stagedModelRepository) {

    _stagedModelRepository = stagedModelRepository;
}

private StagedModelRepository<BookmarksEntry> _stagedModelRepository;
```

   This instantiates a `_stagedModelRepository` object that the staged model data handler can use to access `BookmarksEntry` CRUD operations. Notice the setter method's `@Reference` annotation. This injects the component service of the `BookmarksEntryStagedModelRepository` into the `_stagedModelRepository` object. The component service was created in the Implementing the Staged Model Repository Framework tutorial when setting the `@Component` annotation for the staged model repository.

2. Call your `_stagedModelRepository` object to leverage its specialized staging logic. Now that you have access to CRUD operations via the `_stagedModelRepository` object, you can skip the headache of providing a slew of parameters and additional functionality in the local service to do simple things like add a Bookmarks entry. For example, here's the old way:

```
serviceContext.setUuid(entry.getUuid());

newEntry = _bookmarksEntryLocalService.addEntry(
    userId, portletDataContext.getScopeGroupId(), folderId, entry.getName(), entry.getUrl(), entry.getDescription(), serviceContext);
```

   Now with access to the entry's staged model repository, updating an entry the data handler can use looks like this:

```
newEntry = _stagedModelRepository.updateStagedModel(portletDataContext, importedEntry);
```

   The large number of parameters and UUID setter the local service method requires aren't needed when leveraging the staged model repository, because the staged model repository abstracts these requirements away from the data handler. The `_bookmarksEntryLocalService.addEntry(...)` method is called from the `BookmarksEntryStagedModelRepository` class.

Great! You've successfully leveraged your staged model repository from a data handler!

131.3 Using the Export/Import Lifecycle Listener Framework

The `ExportImportLifecycleListener` framework lets developers 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 could 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, the cache is cleared 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 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 Liferay DXP's `osgi/modules` folder. Once your module is installed and activated in your instance'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 immediately know that it deals with lifecycle events.

The first method `getStagedModelLogFragment` retrieves the staged model's log fragment, which is the lifecycle listener's logging information on events. The next 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 following 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. Anther good example of an event lifecycle listener is the CacheExportImportLifecycleListener.

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.

131.4 Initiating New Export/Import Processes

The Staging and Export/Import features are the building blocks for creating, managing, and publishing a site. These features can be accessed in the *Publishing Tools* menu. You can also, however, start these processes programmatically. 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.

For example, when you want to implement the export feature, 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 an Liferay DXP 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 objects. Once they're created, you can easily implement 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 builds `ExportImportConfiguration` objects used for default local/remote publishing.
- ExportImportConfigurationParameterMapFactory: This factory builds 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 Liferay DXP 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 Liferay DXP. Although this example code resides in Liferay DXP, you could easily use this framework from your own project.

```
**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.

Chapter 132

Liferay Forms

The Liferay Forms application is a full-featured form building tool for collecting data. There's lots of built-in functionality, and for the pieces you're missing, there's lots of extensibility.

This section of tutorials shows developers how to

1. Store form entry data in an alternative format. The default storage type is JSON.
2. Create new form field types.

This list will continue to grow as new tutorials on customizing and extending Liferay Forms are written.

CHAPTER 133

FORM FIELD TYPES

The Forms application contains many highly configurable field types out-of-the-box. Most use cases are met with one of the existing field types.

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 color names. Some users don't, however, know that *Gamboge* is the color of spicy mustard (maybe a little darker), and anyway, seeing colors is better than reading their names, so you can create a field that shows colors.

Another example is a dedicated *time* field. Even with a text field and a tooltip that tells people to *enter the time in the format `hour:minute`*, some users still enter something indecipherable. You can fix this by adding a *time* field to Liferay DXP's Forms application. If you have your own ideas for create your own field types, keep reading to find out how.

These tutorials show you how 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 the structure of a form field type.

133.1 Anatomy of a Field Type Module

All form field type modules have a similar structure. Here's the directory structure of the `dynamic-data-mapping-type-time` module developed in these tutorials:

```
.babelrc
.npmbundlerrc
bnd.bnd
build.gradle
package-lock.json
package.json
```

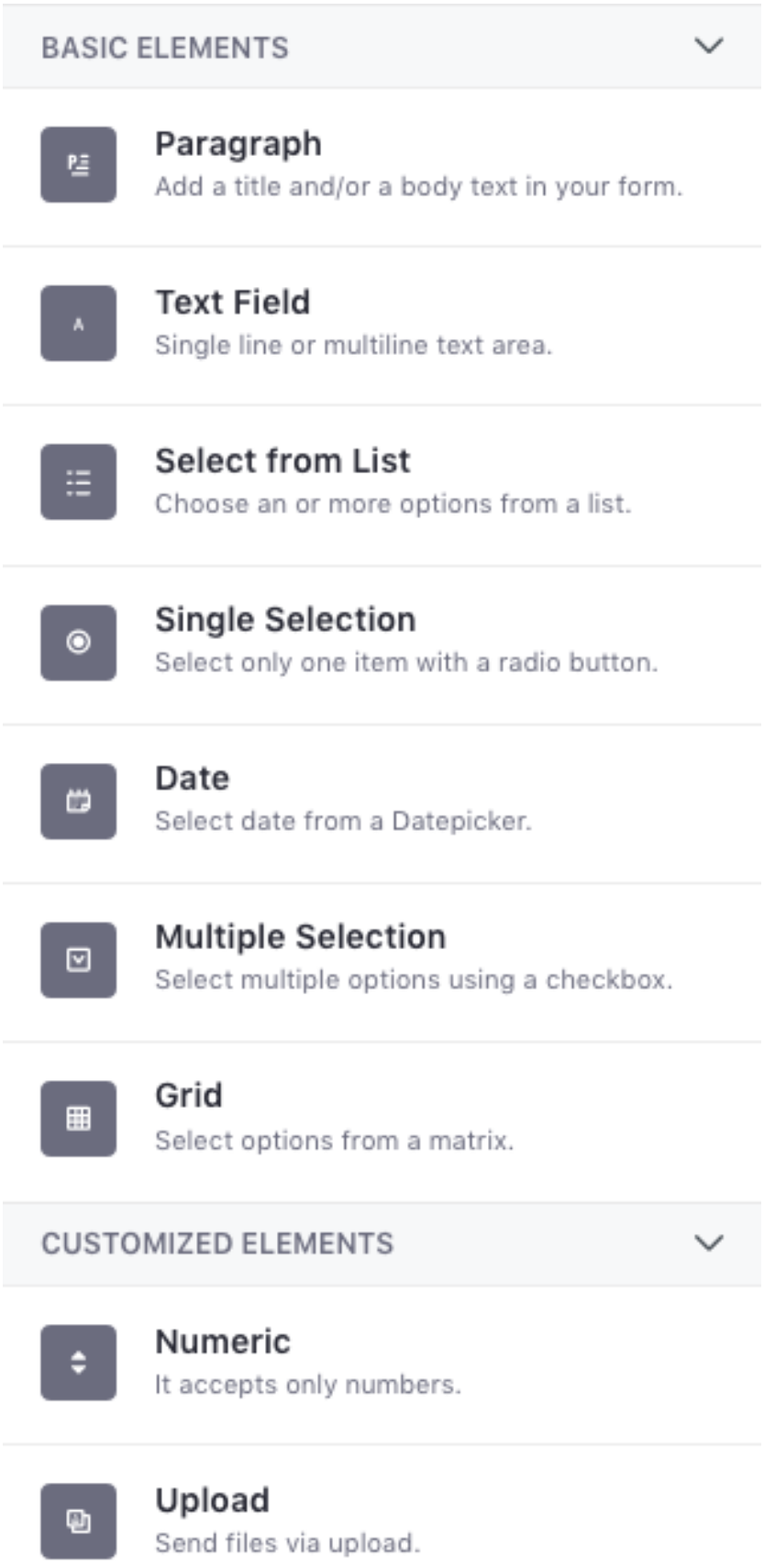


Figure 133.1: The Forms application has useful out-of-the-box field types, but you can add your own if you need to.

```
src
└── main
    ├── java
    │   └── com
    │       └── liferay
    │           └── dynamic
    │               └── data
    │                   └── mapping
    │                       └── type
    │                           └── time
    │                               ├── TimeDDMFormFieldRenderer.java
    │                               ├── TimeDDMFormFieldTemplateContextContributor.java
    │                               ├── TimeDDMFormFieldType.java
    │                               └── TimeDDMFormFieldTypeSettings.java
    └── resources
        ├── content
        │   └── Language.properties
        └── META-INF
            └── resources
                ├── config.js
                ├── time.es.js
                ├── time_field.js
                └── time.soy
```

You don't need `*TemplateContextContributor.java` or `*TypeSettings.java` in the initial module (see Rendering Form Field Settings to learn more about these classes). The initial module consists of these Java classes and resources:

`*DDMFormFieldRenderer.java`: Controls the template's rendering. 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`: Defines the form field type in the back-end. 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 inherits, look at the `DefaultDDMFormFieldTypeSettings` class in the `dynamic-data-mapping-api` module.

`config.js`: Auto-generated if you use Blade CLI, `config.js` defines the dependencies of all declared JavaScript components.

`[name-of-field-type]_field.js`: The JavaScript file modeling your field.

`[name-of-field-type].es.js`: The JavaScript file that configures the template rendering (the `[name-of-field-type].soy` rendering).

`[name-of-field-type].soy`: The template that defines the appearance of the field.

`Language_xx_XX.properties`: Define any terms that must be translated into different languages.

In addition to the Java classes, Soy templates, and JavaScript files, a form field type contains the following files:

`.babelrc`: The Babel configuration file.

`.npmbundlerrc`: The liferay-npm-bundler configuration file.

`bnd.bnd`: The module's metadata.

`build.gradle`: The module's dependencies and build properties.

`package.json`: The npm module manager.

`package-lock.json`: Automatically generated to track the npm modules dependencies.

Get started creating the time field in the next tutorial.

133.2 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 time data is collected, you'll 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. Creating the Form Field Type's Java class.
2. Creating the Form Field Type Renderer Java class.
3. Defining the field's behavior in JavaScript and Soy templates.

Blade Template: To jump-start your project, use Blade CLI or Liferay Dev Studio. There's a Blade template for creating form fields. Using the CLI, enter

```
blade create -t form-field -v 7.1 -p com.liferay.docs.formfieldtype -c Time DDMTypeTime
```

This gives you a `DDMTypeTime` module with a similar structure to what's above. The Java classes are in the package `com.liferay.docs.formfield` under `src/main/java/` and the frontend resources (JavaScript and Soy files) are in `sr/main/resources/META-INF/resources`.

A known limitation in the form-field template requires the use of camel case in the project name (`DDMTypeTime`). Trying to use kebab case instead (`ddm-type-time`) generates a non-functioning module. This is fixed with the release of Blade 3.3. Run `blade version` from the command line to see the version of Blade you're running.

Using Blade CLI or Liferay Dev Studio, you get a project skeleton with much of the boilerplate filled in, so you can focus immediately on coding.

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
```

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 to the modules root folder, so your module's resources are made available upon module activation.

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.description=time-field-type-description",
    "ddm.form.field.type.display.order:Integer=10",
    "ddm.form.field.type.icon=time",
    "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.description` An optional property describing the field type. Its localized value appears in the form builder's sidebar, just below the field's label.

`ddm.form.field.type.display.order` An Integer defining the field type's position in the sidebar.

`ddm.form.field.type.icon` The icon for the field type. Choosing one of the Lexicon icons makes your form field blend in with the existing form field types.

`ddm.form.field.type.js.class.name` The field type's JavaScript class name—the JavaScript file defines 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 form builder's sidebar.

`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.

133.3 Rendering Field Types

Before you get to the front-end coding necessary to render your field type, there's another Component to define and a Java class to code.

Implementing a `DDMFormFieldRenderer`

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
)
```

Extend `BaseDDMFormFieldRenderer`, an abstract class implementing the API's only required method, `render`. The Form engine calls the render method for every form field type present in a form, and returns the plain HTML 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 "DDMTime.render";
    }

    @Override
    public TemplateResource getTemplateResource() {
        return _templateResource;
    }

    @Activate
    protected void activate(Map<String, Object> properties) {
        _templateResource = getTemplateResource("/META-INF/resources/time.soy");
    }

    private TemplateResource _templateResource;

}
```

Set the templating language (Soy closure templates), the template namespace (`DDMTime`) and name (`render`), and point to the location of the templates within your module (`/META-INF/resources/time.soy`).

Writing the Soy Template

Now it's time to write the template you referenced in the renderer class: `time.soy` in the case of the time field type.

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 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.

Create

`src/main/resources/META-INF/resources/time.soy`

and populate it with this:

```
{namespace DDMTime}

/**
* Defines the delegated template for the time field.
*/
{deltemplate ddm.field variant="'time'"}
    {call .render data="all" /}
{/deltemplate}

/**
* Prints the time field.
*/
{template .render}
    {@param name: string}
    {@param pathThemeImages: string}
    {@param value: ?}
    {@param visible: bool}
    {@param? dir: string}
    {@param? label: string}
    {@param? predefinedValue: string}
    {@param? readOnly: bool}
    {@param? required: bool}
    {@param? showLabel: bool}
    {@param? tip: string}

    {let $displayValue: $value ? $value : $predefinedValue ? $predefinedValue : '' /}

    <div class="form-group {$visible ? '' : 'hide'} liferay-ddm-form-field-time"
        data-fieldname="{$name}">
        {if $showLabel or $required}
            <label for="{$name}">
                {if $showLabel}
                    {$label}{sp}
                {/if}

                {if $required}
                    <svg aria-hidden="true" class="lexicon-icon lexicon-icon-asterisk reference-mark">
                        <use xlink:href="{$pathThemeImages}/lexicon/icons.svg#asterisk" />
                    </svg>
                {/if}
            </label>
        {/if}

        {if $showLabel}
            {if $tip}
                <span class="form-text">{$tip}</span>
            {/if}
        {/if}

        <div class="input-group">
            <div class="input-group-item">
                <input class="field form-control"
```

```
                {if $dir}dir="{$dir}"{/if}
                {if $readOnly}disabled{/if}
                id="{$name}"
                name="{$name}"
                type="text"
                value="{$displayValue}">
            </div>
        </div>
    </div>
{/template}
```

There are four important things to do in the template:

1. Define the template namespace. The template namespace can define multiple templates for your field type by adding the namespace as a prefix.

   ```
   {namespace DDMTime}
   ```

2. Set the template that's called to render the time field. The `variant="'time'"` identifies the time field, and the `.render` names the template that renders it. The template itself follows and is defined through the block `{template .render}...{/template}`.

   ```
   /**
   * Defines the delegated template for the time field.
   */
   {deltemplate ddm.field variant="'time'"}
       {call .render data="all" /}
   {/deltemplate}
   ```

3. 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). All listed parameters are available by default.

   ```
   {@param name: string}
   {@param pathThemeImages: string}
   {@param value: ?}
   {@param visible: bool}
   {@param? dir: string}
   {@param? label: string}
   {@param? predefinedValue: string}
   {@param? readOnly: bool}
   {@param? required: bool}
   {@param? showLabel: bool}
   {@param? tip: string}
   ```

4. Write the template logic (everything encapsulated by the `{template .render}...{/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 `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 the template is defined, write the JavaScript file modeling your field.

Writing the JavaScript Files

Create a `time_field.js` file 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.

Next write the `*.es.js` file to configure the Soy template's rendering. Create a file called `time.es.js` and populate it:

```
import Component from 'metal-component';
import Soy from 'metal-soy';
import templates from './time.soy';
/**
 * Time Component
 */
class Time extends Component {}
// Register component
Soy.register(Time, templates, 'render');
if (!window.DDMTime) {
    window.DDMTime = {
    };
}
window.DDMTime.render = Time;
export default Time;
```

This dictates that the Soy template is called to render the `Time` component. Then create the `config.js` file:

```
;(function() {
    AUI().applyConfig(
        {
            groups: {
                'field-time': {
                    base: MODULE_PATH + '/',
                    combine: Liferay.AUI.getCombine(),
                    filter: Liferay.AUI.getFilterConfig(),
```

```
                modules: {
                    'liferay-ddm-form-field-time': {
                        condition: {
                            trigger: 'liferay-ddm-form-renderer'
                        },
                        path: 'time_field.js',
                        requires: [
                            'liferay-ddm-form-renderer-field'
                        ]
                    }
                },
                root: MODULE_PATH + '/'
            }
        }
    }
);
})();
```

This file is entirely boilerplate. In fact, if you use Blade CLI to generate a field type module, you won't need to touch this file. Functionally, it's a JavaScript file that defines the dependencies of the declared JavaScript components (`requires...`), and where the files are located (`path...`). The Alloy loader uses `config.js` when it satisfies dependencies for each JavaScript component. For more information about the Alloy loader see its tutorial.

If you build and deploy your new field type module, 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

To do more than provide a text input field, define additional 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 of the `time_field.js` and `config.js`:

```
{
    requires: ['aui-timepicker','liferay-ddm-form-renderer-field']
}
```

Change the default rendering of the field by overwriting the base `render` logic, instantiating the time picker, and adding the chosen time to the field. 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
                },
                after: {
                    selectionChange: A.bind('afterSelectionChange', instance)
                }
            }
        );
    },

    afterSelectionChange: function(event) {
        var instance = this;
```

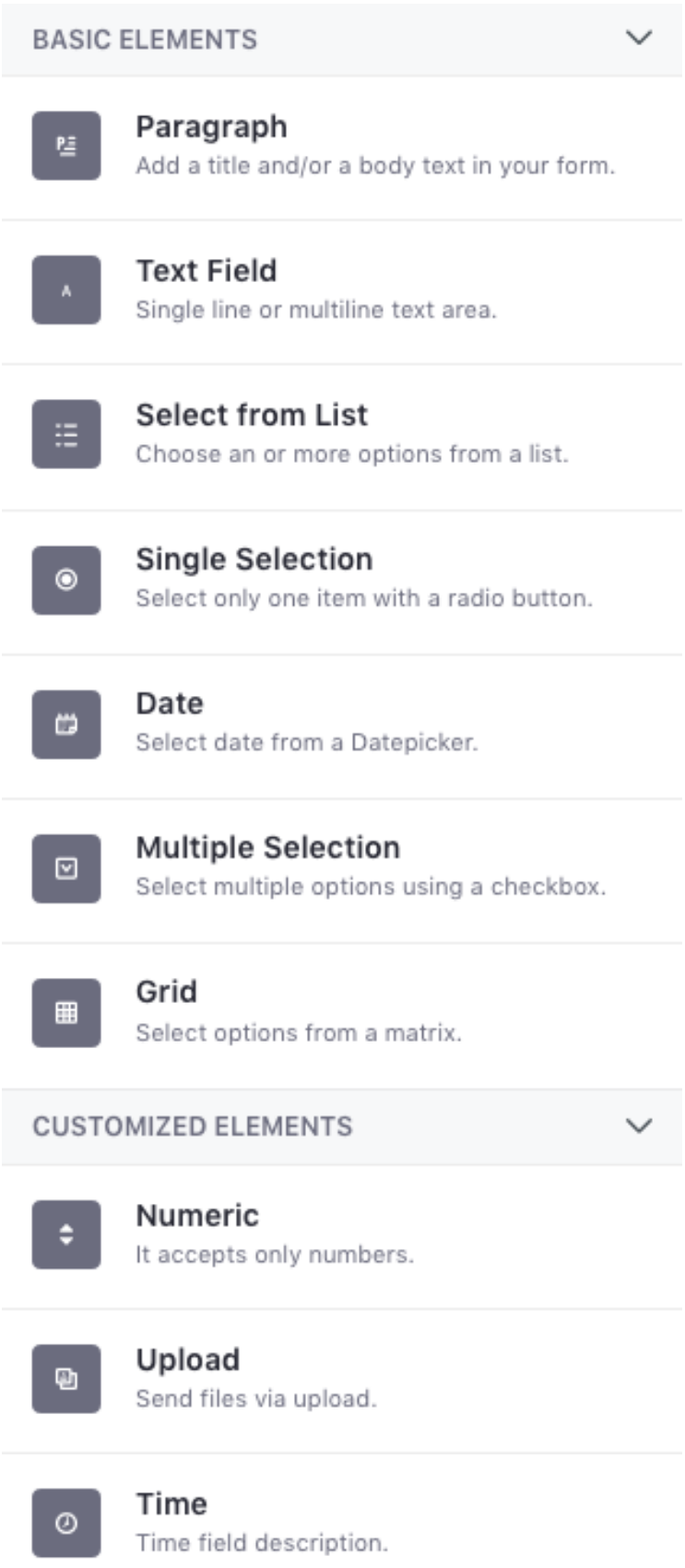


Figure 133.2: Add your own form field types to the Forms application.

```
        var time = event.newSelection;

        instance.set('value', time);
    }
}
```

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`. In addition, add a callback method (`afterSelectionChange`) to be executed after the time is chosen in the time picker. This method updates the field's value. See the Alloy documentation for more information.

Now when the field is rendered, there's a real time picker.

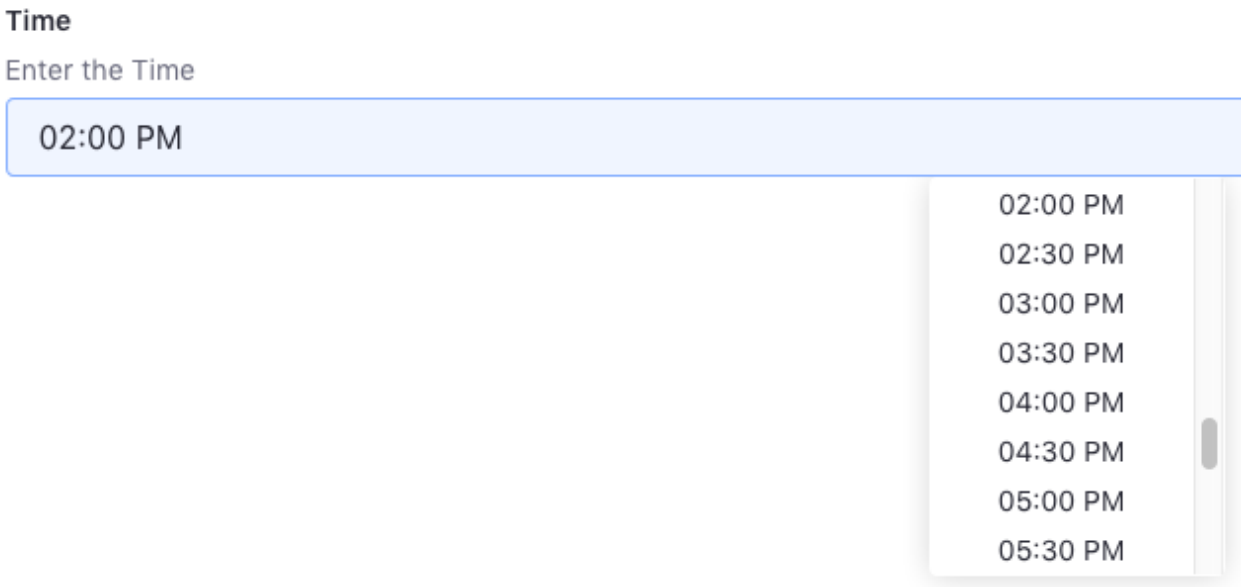


Figure 133.3: The Alloy UI Timepicker in action.

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 to learn how.

133.4 Adding Settings to Form Field Types

Once you develop a Form Field Type, you might need to add settings to it. For example, a Time field might 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 used to display the time choices. Use the strftime format to pick the mask you want.

To add settings to form field types, take these steps:

- Write an interface that extends the default field type configuration, `DefaultDDMFormFieldTypeSettings`.

- Update the *FormFieldType to refer the new interface created on the previous step.
- 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 settings that must be rendered in a form (the placeholder, in our example).

First craft the interface that controls your field's settings.

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 *Field Type* configuration form.

Here's what it looks like:

```
package com.liferay.dynamic.data.mapping.type.time;

import com.liferay.dynamic.data.mapping.annotations.DDMForm;
import com.liferay.dynamic.data.mapping.annotations.DDMFormField;
import com.liferay.dynamic.data.mapping.annotations.DDMFormLayout;
import com.liferay.dynamic.data.mapping.annotations.DDMFormLayoutColumn;
import com.liferay.dynamic.data.mapping.annotations.DDMFormLayoutPage;
import com.liferay.dynamic.data.mapping.annotations.DDMFormLayoutRow;
import com.liferay.dynamic.data.mapping.form.field.type.DefaultDDMFormFieldTypeSettings;

@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 = {
                                "dataType", "name", "showLabel", "repeatable",
                                "type", "validation", "visibilityExpression"
                            }
                        )
```

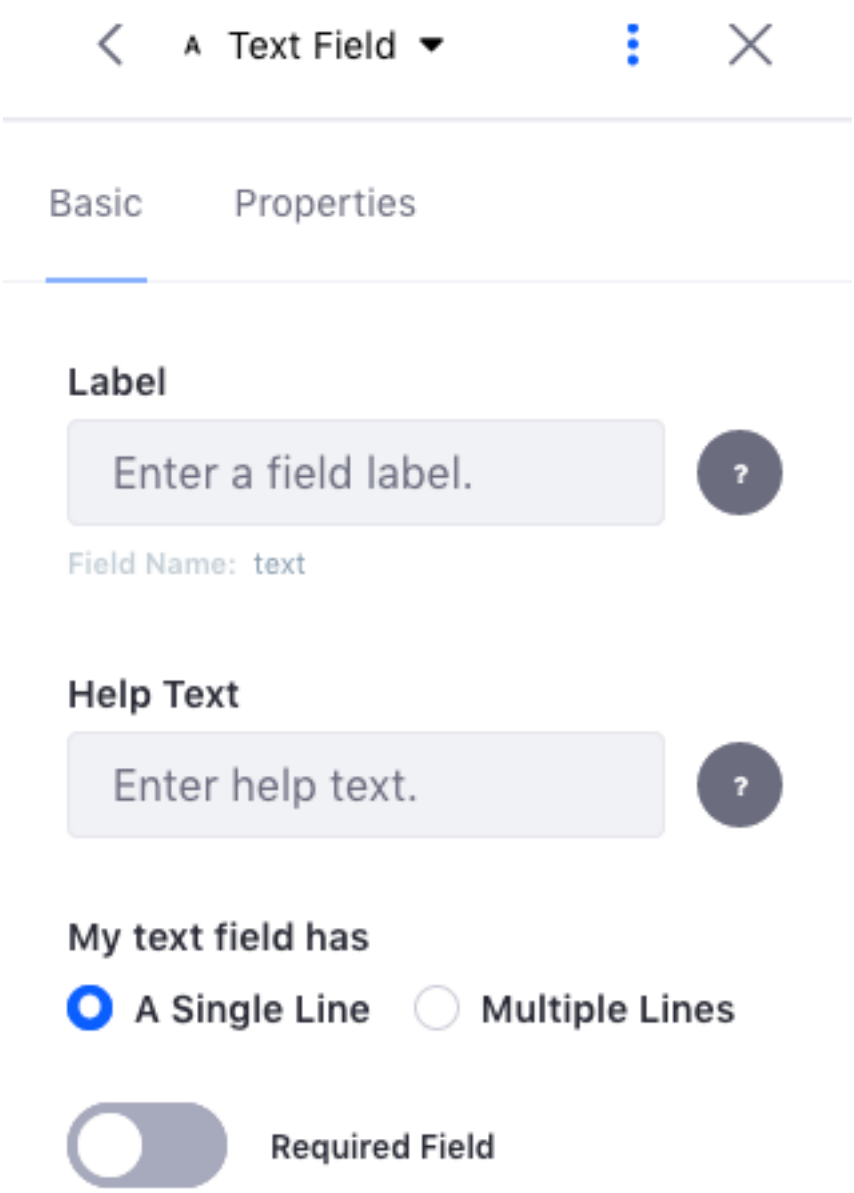


Figure 133.4: Like your custom field types, the text field type's settings are configured in a Java interface.

```
                }
            )
        }
    )
  }
)
public interface TimeDDMFormFieldTypeSettings
    extends DefaultDDMFormFieldTypeSettings {

    @DDMFormField(label = "%mask", predefinedValue="%I:%M %p")
    public String mask();

    @DDMFormField(label = "%placeholder-text")
    public String placeholder();

}
```

Most of the work you need to do is in the class's annotations.

This class sets up a dynamic form with all the settings the 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 must 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 `@DDMFormRule` annotation.

The interface extends `DefaultDDMFormFieldTypeSettings`. That's why the default settings can be used 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 `paginationMode` 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. It 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`.

The layout page titles `%basic` and `%properties` are common to all of Liferay DXP's field types, but you can use whatever titles you want. To change the title of a layout page, specify the title in the annotation properties (`title = "%advanced"`, for example), and then create a new key in the language resources files. For example, use `advanced=Advanced` in the `Language.properties`.

`@DDMFormLayoutRow` Lay out the number of columns you want in the row. Most settings forms have just one row and one column.

`@DDMFormLayoutColumn` 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` Add new fields to the settings form. In this example, the `mask` and `placeholder` settings are configured with this annotation. Don't forget to add the settings language keys (`mask` and `placeholder-text`) to the language resources files.

Once your *TypeSettings class is finished, update the *Type class for your form field type.

Updating the Type Class

The class TimeDDMFormFieldType currently has one method, getName, returning the name of the current form field. Add a new method to reference TimeDDMFormFieldTypeSettings that holds the specific settings of the Time field. This method already exists in the base class (BaseDDMFormFieldType), so override it:

```
@Override
public Class<? extends DDMFormFieldTypeSettings>
    getDDMFormFieldTypeSettings() {

    return TimeDDMFormFieldTypeSettings.class;
}
```

Next, render new Time field settings.

133.5 Rendering Form Field Settings

Once the settings are added to the class backing the field's settings, make sure the *Renderer can get the settings and update the front-end code.

Passing Settings to the Renderer Class

Send the new configuration settings to the Soy template so they can be displayed to the end user. Create a new Java class implementing the interface DDMFormFieldTemplateContextContributor and modify the existing class *DDMFormFieldRenderer.

The DDMFormFieldTemplateContextContributor interface has a single method named getParameters. It gets the new configuration settings, specific for a form field type, and sends for the resources that need them, like the Soy template. To get these settings, create a new class, TimeDDMFormFieldTemplateContextContributor. First create its OSGI component annotation and the class declaration:

```
@Component(
    immediate = true,
    property = "ddm.form.field.type.name=time",
    service = {
        DDMFormFieldTemplateContextContributor.class,
        TimeDDMFormFieldTemplateContextContributor.class
    }
)
public class TimeDDMFormFieldTemplateContextContributor
    implements DDMFormFieldTemplateContextContributor {
```

Then override getParameters to get the new configurations settings, placeholder and mask:

```
    @Override
    public Map<String, Object> getParameters(
        DDMFormField ddmFormField,
        DDMFormFieldRenderingContext ddmFormFieldRenderingContext) {

        Map<String, Object> parameters = new HashMap<>();
```

```
        parameters.put(
            "placeholder", (String)ddmFormField.getProperty("placeholder"));
        parameters.put("mask", (String)ddmFormField.getProperty("mask"));

        return parameters;
    }

}
```

Now pass the configuration settings to the template with a new method, `populateOptionalContext`, in `TimeDDMFormFieldRenderer`:

```
@Override
protected void populateOptionalContext(
    Template template, DDMFormField ddmFormField,
    DDMFormFieldRenderingContext ddmFormFieldRenderingContext) {

    Map<String, Object> parameters =
        timeDDMFormFieldTemplateContextContributor.getParameters(
            ddmFormField, ddmFormFieldRenderingContext);

    template.putAll(parameters);
}

@Reference
protected TimeDDMFormFieldTemplateContextContributor
    timeDDMFormFieldTemplateContextContributor;
```

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 like the user's locale, the HTTP request and response objects, the portlet namespace, and more (all of its included properties can be found here).

The OSGI reference (`@Reference`) provides access to the `TimeDDMFormFieldTemplateContextContributor` service.

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 is used for server side and 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 entered 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 must 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
            },
            after: {
                selectionChange: A.bind('afterSelectionChange', instance)
            }
        }
    );
},
```

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

Add the placeholder setting to your Soy template's logic.

The whole template is included below, but the only additions are in the list of parameters (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 DDMTime}

/**
* Defines the delegated template for the time field.
*/
{deltemplate ddm.field variant="'time'"}
    {call .render data="all" /}
{/deltemplate}

/**
* Prints the time field.
*/
{template .render}
    {@param name: string}
    {@param pathThemeImages: string}
    {@param value: ?}
    {@param visible: bool}
    {@param? placeholder: string}
    {@param? dir: string}
    {@param? label: string}
    {@param? predefinedValue: string}
    {@param? readOnly: bool}
    {@param? required: bool}
    {@param? showLabel: bool}
    {@param? tip: string}

    {let $displayValue: $value ? $value : $predefinedValue ? $predefinedValue : '' /}

    <div class="form-group {$visible ? '' : 'hide'} liferay-ddm-form-field-time"
        data-fieldname="{$name}">
        {if $showLabel or $required}
            <label for="{$name}">
                {if $showLabel}
                    {$label}{sp}
                {/if}

                {if $required}
                    <svg aria-hidden="true" class="lexicon-icon lexicon-icon-asterisk reference-mark">
                        <use xlink:href="{$pathThemeImages}/lexicon/icons.svg#asterisk" />
                    </svg>
                {/if}
            </label>
        {/if}

        {if $showLabel}
            {if $tip}
                <span class="form-text">{$tip}</span>
            {/if}
        {/if}

        <div class="input-group">
            <div class="input-group-item">
                <input class="field form-control"
                    {if $dir}dir="{$dir}"{/if}
                    {if $readOnly}disabled{/if}
                    id="{$name}"
                    name="{$name}"
                    placeholder="{$placeholder}"
                    type="text"
                    value="{$displayValue}">
            </div>
        </div>
    </div>
{/template}
```

The mask is not needed in the Soy 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 mask 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.

133.6 Forms Storage Adapters

When a User adds a form record, the Forms API routes the processing of the request through the storage adapter API. The same is true for the other *CRUD* operations performed on form entries (read, update, and delete operations). The default implementation of the storage service is called `JSONStorageAdapter`, and as its name implies, it implements the `StorageAdapter` interface to provide JSON storage of form entry data.

The DDM backend can *adapt* to other data storage formats for form records. Want to store your data in XML? YAML? No problem. Because the storage API is separated from the regular service calls used to populate the database table for form entries, a developer can even choose to store form data outside the Liferay database.

Define your own format to save form entries by writing an OSGi component that implements the `StorageAdapter` interface. The interface follows the *CRUD* approach, so implementing it requires that you write methods to create, read, update and delete form values.

Note: When you add a new storage adapter, it can only be used with new Forms. All existing Forms continue to use the adapter selected (JSON by default) at the time of their creation, and a different storage adapter cannot be selected.

The example storage adapter in this tutorial serializes form data to be stored in a simple file, stored on the file system.

Note that these code snippets include the references to the services they're calling directly beneath the first method that uses the service. It's a Liferay code convention to place these at the very end of the class.

Implementing a Storage Adapter

First declare the class a `Component` that provides a `StorageAdapter` implementation. To implement a storage adapter, extend the abstract `BaseStorageAdapter` class.

```
@Component(service = StorageAdapter.class)
public class FileSystemStorageAdapter extends BaseStorageAdapter {
```

Step 1: Name the Storage Type

The only method without a base implementation in the abstract class is `getStorageType`. For the file system storage example, just make it return `"File System"`.

```
@Override
public String getStorageType() {
    return "File System";
}
```

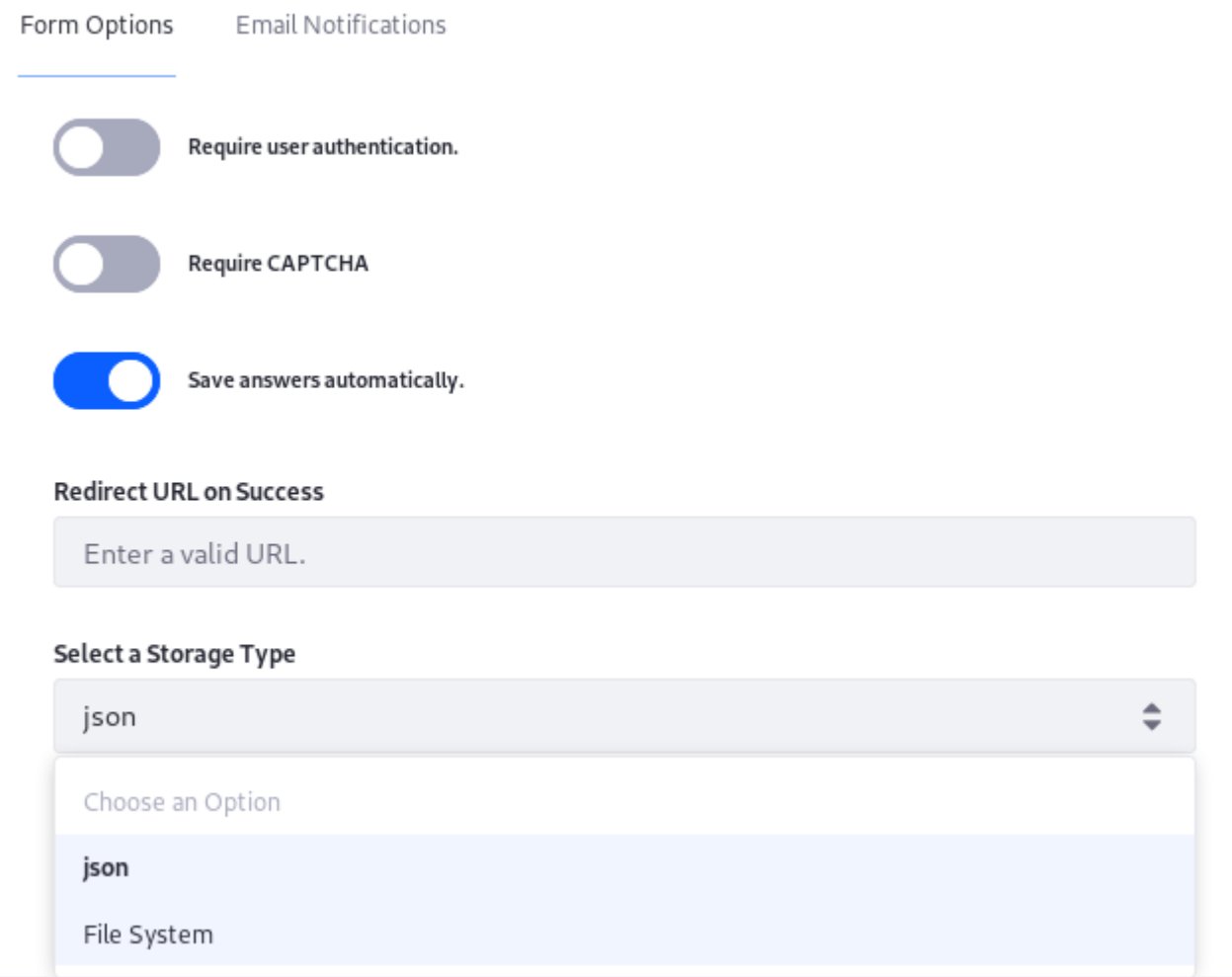


Figure 133.5: Choose a Storage Type for your form records.

Return a human readable String, as `getStorageType` determines what appears in the UI when the form creator is selecting a storage type for their form. The String value you return here is added to the `StorageAdapterRegistry`'s Map of storage adapters.

Step 2: Override the CRUD Methods

Next override the `doCreateMethod` to return a `long` that identifies each form record with a unique file ID:

```
@Override
protected long doCreate(
    long companyId, long ddmStructureId, DDMFormValues ddmFormValues,
    ServiceContext serviceContext)
    throws Exception {

    validate(ddmFormValues, serviceContext);

    long fileId = _counterLocalService.increment();
```

```
    DDMStructureVersion ddmStructureVersion =
        _ddmStructureVersionLocalService.getLatestStructureVersion(
            ddmStructureId);

    long classNameId = PortalUtil.getClassNameId(
        FileSystemStorageAdapter.class.getName());

    _ddmStorageLinkLocalService.addStorageLink(
        classNameId, fileId, ddmStructureVersion.getStructureVersionId(),
        serviceContext);

    saveFile(
        ddmStructureVersion.getStructureVersionId(), fileId, ddmFormValues);

    return fileId;
}

@Reference
private CounterLocalService _counterLocalService;

@Reference
private DDMStorageLinkLocalService _ddmStorageLinkLocalService;

@Reference
private DDMStructureVersionLocalService _ddmStructureVersionLocalService;
```

The first line in this method (and the subsequent `doUpdate` method) calls a `validate` method that's not yet written, so don't save the class until you've written that method.

In addition to returning the file ID, add a storage link via the `DDMStorageLinkLocalService`. The DDM Storage Link associates each form record with the DDM Structure backing the form.

The `addStorageLink` method takes class name ID as retrieved by `PortalUtil.getClassNameId`, the `fileId` (used as the primary key for the file storage type), the structure version ID, and the service context. There's also a call to a `saveFile` method, which serializes the forms record's values and uses two additional utility methods (`getStructureFolder` and `getFile`) to write a `java.io.File` object. There are some other utility methods invoked as well:

```
private File getFile(long structureId, long fileId) {
    return new File(
        getStructureFolder(structureId), String.valueOf(fileId));
}

private File getStructureFolder(long structureId) {
    return new File(String.valueOf(structureId));
}

private void saveFile(
        long structureVersionId, long fileId, DDMFormValues formValues)
    throws IOException {

    String serializedDDMFormValues = _ddmFormValuesJSONSerializer.serialize(
        formValues);

    File formEntryFile = getFile(structureVersionId, fileId);

    FileUtil.write(formEntryFile, serializedDDMFormValues);
}

@Reference
private DDMFormValuesJSONSerializer _ddmFormValuesJSONSerializer;
```

Note the call to the `write` method.`FileUtil` is a Liferay utility class for manipulating `java.io.File` objects. By default, the `write` method writes the data into the user home folder of your system.

Override the `doDeleteByClass` method to delete the `File` using the `classPK`:

```
@Override
protected void doDeleteByClass(long classPK) throws Exception {
    DDMStorageLink storageLink =
        _ddmStorageLinkLocalService.getClassStorageLink(classPK);

    FileUtil.delete(getFile(storageLink.getStructureId(), classPK));

    _ddmStorageLinkLocalService.deleteClassStorageLink(classPK);
}
```

Once the file is deleted, its storage links should also be deleted. Use `doDeleteByDDMStructure` for this logic:

```
@Override
protected void doDeleteByDDMStructure(long ddmStructureId)
    throws Exception {

    FileUtil.deltree(getStructureFolder(ddmStructureId));

    _ddmStorageLinkLocalService.deleteStructureStorageLinks(ddmStructureId);
}
```

To retrieve the form record's values from the `File` object where they were written, override `doGetDDMFormValues`:

```
@Override
protected DDMFormValues doGetDDMFormValues(long classPK) throws Exception {
    DDMStorageLink storageLink =
        _ddmStorageLinkLocalService.getClassStorageLink(classPK);

    DDMStructureVersion structureVersion =
        _ddmStructureVersionLocalService.getStructureVersion(
            storageLink.getStructureVersionId());

    String serializedDDMFormValues = FileUtil.read(
        getFile(structureVersion.getStructureVersionId(), classPK));

    return _ddmFormValuesJSONDeserializer.deserialize(
        structureVersion.getDDMForm(), serializedDDMFormValues);
}

@Reference
private DDMFormValuesJSONDeserializer _ddmFormValuesJSONDeserializer;
```

Override the `doUpdate` method so the record's values can be overwritten. This example calls the `saveFile` utility method provided earlier:

```
@Override
protected void doUpdate(
        long classPK, DDMFormValues ddmFormValues,
        ServiceContext serviceContext)
    throws Exception {

    validate(ddmFormValues, serviceContext);

    DDMStorageLink storageLink =
        _ddmStorageLinkLocalService.getClassStorageLink(classPK);

    saveFile(
        storageLink.getStructureVersionId(), storageLink.getClassPK(),
        ddmFormValues);
}
```

Once the CRUD logic is defined, deploy and test the storage adapter.

Step 3: Validating Form Entries

The `doCreate` and `doUpdate` methods above both include this line:

```
validate(ddmFormValues, serviceContext);
```

Because the Storage Adapter handles User entered data, it's important to validate that the entries include only appropriate data. Add a `validate` method to the `StorageAdapter`:

```
protected void validate(
    DDMFormValues ddmFormValues, ServiceContext serviceContext)
    throws Exception {

    boolean validateDDMFormValues = GetterUtil.getBoolean(
        serviceContext.getAttribute("validateDDMFormValues"), true);

    if (!validateDDMFormValues) {
        return;
    }

    _ddmFormValuesValidator.validate(ddmFormValues);
}
```

Make sure to do three things:

1. Retrieve the value of the `validateDDMFormValues` attribute.
2. If `validateDDMFormValues` is false, exit the validation without doing anything.

   When a User accesses a form at its dedicated link, there's a periodic auto-save process of in-progress form values. There's no need to validate this data until the User hits the *Submit* button on the form, so the auto-save process sets the `validateDDMFormValues` attribute to `false`.
3. Otherwise, call the validate method from the `DDMFormValuesValidator` service.

Once the necessary logic is in place, deploy and test the Storage Adapter.

Enabling the Storage Adapter

The storage adapter is enabled at the individual form level. Create a new form, and select the Storage Adapter *before saving or publishing the form*. If you wait until first Saving the Form, the default Storage Adapter is already assigned to the Form, and this setting is no longer editable.

1. Go to the Site Menu → Content → Forms, and click the *Add* button ().
2. In the Form Builder view, click the *Options* button () and open the *Settings* window.
3. From the select list field called *Select a Storage Type,* choose the desired type and click *Done*.

Now all the form's entries are stored in the desired format.

CHAPTER 134

WORKFLOW

Use the workflow framework to run assets through a business process that suit your organization's needs. Workflow processes are created using XML or via the handy Kaleo Designer tool that comes with DE subscriptions.

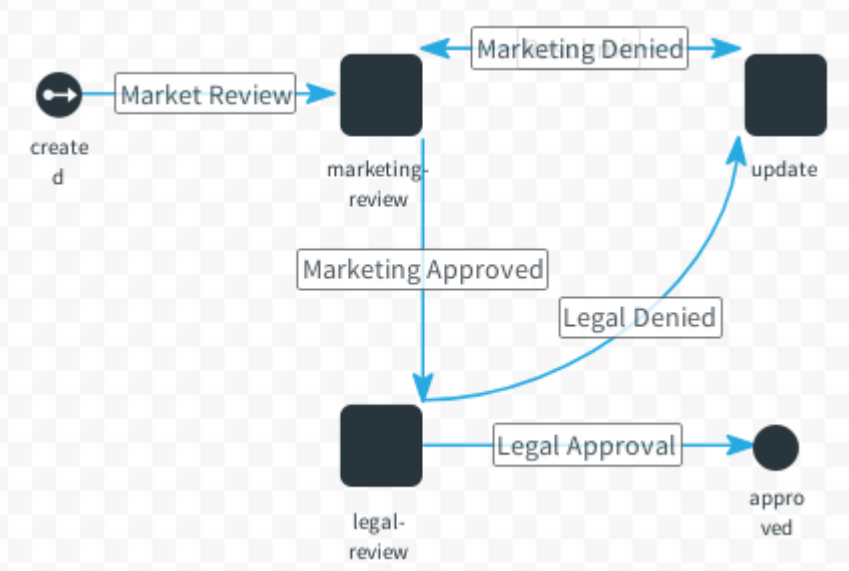


Figure 134.1: If you don't like XML, the visual Kaleo Designer makes designing workflows easy and intuitive.

This set of tutorials delves into the following workflow framework topics:

- Creating Workflow Definitions (using XML)
- Enabling Assets for Workflow
- Creating Workflow Engine Adapters

To manage workflow definitions and define their assignment schemes, check out the workflow section of the administrator documentation(not yet written).

CHAPTER 135

CRAFTING XML WORKFLOW DEFINITIONS

You don't need a fancy visual designer to build workflows. To be clear, Kaleo Designer may make you a faster workflow designer through its graphical interface. If you plan to build lots of workflow processes, a Digital Enterprise subscription gets you access to Kaleo Designer. But with a little copy and paste from existing workflows and a little handcrafted XML, you can build any workflow and attain workflow wizard-hood in the process. Follow this set of tutorials to learn what elements you can put into your definitions.

135.1 Existing Workflow Definitions

Only one workflow definition is installed by default: Single Approver. Several more, however, are embedded in the source code of your Liferay DXP installation. If you're comfortable extracting the XML files from a JAR file embedded in an LPKG file, you're welcome to follow the steps below to obtain the workflow definitions. To obtain the files more conveniently, download a ZIP file here.

To extract the definitions for yourself, navigate to

```
[Liferay Home]/osgi/marketplace
```

and open (using an archive manager) `Liferay CE Forms and Workflow.lpkg`. Open the JAR file named

```
com.liferay.portal.workflow.kaleo.runtime.impl-[version].jar
```

In the JAR file, navigate to

```
META-INF/definitions/
```

and extract the four XML workflow definition files. These definitions provide good reference material for many of the workflow features and elements described in these articles. In fact, most of the XML snippets you see here are lifted directly from these definitions.

135.2 Schema

The XML structure of a workflow definition is defined in an XSD file:

```
liferay-worklow-definition-7_0_0.xsd
```

Declare the schema at the top of the workflow definition file:

```
<?xml version="1.0"?>
<workflow-definition
    xmlns="urn:liferay.com:liferay-workflow_7.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:liferay.com:liferay-workflow_7.0.0
        http://www.liferay.com/dtd/liferay-workflow-definition_7_0_0.xsd">
```

To read 464 lines of beautifully formatted XML that defines how to write more XML (it's practically poetic), check out the XSD here. Otherwise, move on to entering the definition's metadata.

135.3 Metadata

Give the definition a name, description, and version:

```
<name>Category Specific Approval</name>
<description>A single approver can approve a workflow content.</description>
<version>1</version>
```

All these tags are optional. If present the first time a definition is saved, the `<name>` tag serves as a unique identifier for the definition. If not specified (or added sometime after the first save), a random unique name is generated and used to identify the workflow.

Once the schema and metadata are in place, it's time to turn up the funky beats and get into the flow (the workflow). Learn about workflow nodes in the next article.

135.4 Workflow Definition Nodes

After your definition's schema and metadata are in place, begin defining the process. *Node* elements, with their sub-elements, are fundamental building blocks making up workflow definitions.

State Nodes don't require user input. The workflow does whatever is specified in the state node's `actions` tag (a notification and/or a custom script), and then moves to the provided transition. Workflows start and end with a state. The initial state node often only contains a transition:

```
    <state>
        <name>created</name>
        <initial>true</initial>
        <transitions>
            <transition>
                <name>Determine Branch</name>
                <target>determine-branch</target>
                <default>true</default>
            </transition>
        </transitions>
    </state>
```

If a notification or script is required in your state node, use an `actions` tag. Here's an `action` element containing a Groovy script. This is found in many terminal state nodes and marks the asset as approved in the workflow.

```
<actions>
    <action>
        <name>Approve</name>
        <description>Approve</description>
        <script>
            <![CDATA[
            com.liferay.portal.kernel.workflow.WorkflowStatusManagerUtil.
                updateStatus(com.liferay.portal.kernel.workflow.WorkflowConstants.
                    getLabelStatus("approved"), workflowContext);]]>
        </script>
        <script-language>groovy</script-language>
        <execution-type>onEntry</execution-type>
    </action>
</actions>
```

Conditions let you inspect the asset (or its execution context) and do something, like send it to a particular transition.

Here's the `determine-branch` condition from the Category Specific Approval workflow definition:

```
<condition>
    <name>determine-branch</name>
    <script>
        <![CDATA[
            import com.liferay.asset.kernel.model.AssetCategory;
            import com.liferay.asset.kernel.model.AssetEntry;
            import com.liferay.asset.kernel.model.AssetRenderer;
            import com.liferay.asset.kernel.model.AssetRendererFactory;
            import com.liferay.asset.kernel.service.AssetEntryLocalServiceUtil;
            import com.liferay.portal.kernel.util.GetterUtil;
            import com.liferay.portal.kernel.workflow.WorkflowConstants;
            import com.liferay.portal.kernel.workflow.WorkflowHandler;
            import com.liferay.portal.kernel.workflow.WorkflowHandlerRegistryUtil;

            import java.util.List;

            String className = (String)workflowContext.get(WorkflowConstants.CONTEXT_ENTRY_CLASS_NAME);

            WorkflowHandler workflowHandler = WorkflowHandlerRegistryUtil.getWorkflowHandler(className);

            AssetRendererFactory assetRendererFactory = workflowHandler.getAssetRendererFactory();

            long classPK = GetterUtil.getLong((String)workflowContext.get(WorkflowConstants.CONTEXT_ENTRY_CLASS_PK));

            AssetRenderer assetRenderer = workflowHandler.getAssetRenderer(classPK);

            AssetEntry assetEntry = assetRendererFactory.getAssetEntry(assetRendererFactory.getClassName(), assetRenderer.getClassPK());

            List<AssetCategory> assetCategories = assetEntry.getCategories();

            returnValue = "Content Review";

            for (AssetCategory assetCategory : assetCategories) {
                String categoryName = assetCategory.getName();

                if (categoryName.equals("legal")) {
                    returnValue = "Legal Review";

                    return;
                }
            }
        ]]>
```

```
    </script>
    <script-language>groovy</script-language>
    <transitions>
        <transition>
            <name>Legal Review</name>
            <target>legal-review</target>
            <default>false</default>
        </transition>
        <transition>
            <name>Content Review</name>
            <target>content-review</target>
            <default>false</default>
        </transition>
    </transitions>
</condition>
```

This example checks the asset category to choose the processing path, whether to transition to the *Legal Review* task or the *Content Review* task.

The `returnValue` variable points from the condition to a transition, and its value must match a valid transition name. This script looks up the asset in question, retrieves its asset category, and sets an initial `returnValue`. Then it checks to see if the asset has been marked with the *legal* category. If not it goes through *Content Review* (to the content-review task in the workflow), and if it does it goes through *Legal Review* (to the legal-review task in the workflow).

Forks and Joins: Forks split the workflow process, and joins bring the process back to a unified branch. Processing must always be brought back using a Join (or a Join XOR), and the number of forks and joins in a workflow definition must be equal.

```
<fork>
    <name>fork-1</name>
    <transitions>
        <transition>
            <name>transition-1</name>
            <target>task-1</target>
            <default>true</default>
        </transition>
        <transition>
            <name>transition-2</name>
            <target>task-2</target>
            <default>false</default>
        </transition>
    </transitions>
</fork>
<join>
    <name>join-1</name>
    <transitions>
        <transition>
            <name>transition-4</name>
            <target>EndNode</target>
            <default>true</default>
        </transition>
    </transitions>
</join>
```

The workflow doesn't move past the join until the asset transitions to it from both of the forks. To fork the workflow process, but then allow the processing to continue when only one fork is completed, use a Join XOR.

A Join XOR differs from a join in one important way: it removes the constraint that both forks must be completed before processing can continue. The asset must complete just one of the forks before processing continues.

```
<join-xor>
    <name>join-xor</name>
    <transitions>
        <transition>
            <name>transition3</name>
            <target>EndNode</target>
            <default>true</default>
        </transition>
    </transitions>
</join-xor>
```

Task nodes are at the core of the workflow definition. They're the part where a user interacts with the asset in some way. Tasks can also have sub-elements, including notifications, assignments, and task timers.

Here's the `content-review` task from the Category Specific Approval workflow, with some of the `role` assignment tags cut out for brevity:

```
<task>
    <name>content-review</name>
    <actions>
        <notification>
            <name>Review Notification</name>
            <template>You have a new submission waiting for your review in the workflow.</template>
            <template-language>text</template-language>
            <notification-type>email</notification-type>
            <notification-type>user-notification</notification-type>
            <execution-type>onAssignment</execution-type>
        </notification>
    </actions>
    <assignments>
        <roles>
            <role>
                <role-type>organization</role-type>
                <name>Organization Administrator</name>
            </role>
            ...
        </roles>
    </assignments>
    <task-timers>
        <task-timer>
            <name></name>
            <delay>
                <duration>1</duration>
                <scale>hour</scale>
            </delay>
            <blocking>false</blocking>
            <timer-actions>
                <timer-notification>
                    <name></name>
                    <template></template>
                    <template-language>text</template-language>
                    <notification-type>user-notification</notification-type>
                </timer-notification>
            </timer-actions>
        </task-timer>
    </task-timers>
    <transitions>
        <transition>
            <name>approve</name>
            <target>approved</target>
            <default>true</default>
        </transition>
        <transition>
            <name>reject</name>
            <target>update</target>
            <default>false</default>
```

```
        </transition>
    </transitions>
</task>
```

Learn more about workflow tasks in the next article.

135.5 Workflow Task Nodes

Task nodes are fundamental parts of a workflow definition. When you define your organization's business processes and design corresponding workflows, you likely first envision the tasks. As the name implies, tasks are the part of the workflow where *work* is done. A user enters the picture and must interact with the submitted asset. Users often take the role of reviewer, deciding if an asset from the workflow is acceptable for publication or needs more work.

Unlike other workflow nodes, task nodes have Assignments, because a user is expected to *do something* (often approve or reject the submitted asset) when a workflow process enters the task node.

Commonly, task nodes contain task timers, assignments, actions (which can include notifications and scripts), and transitions. Notifications and actions aren't limited to task nodes, but task nodes and their assignments deserve their own article (this one).

Check out the Review task in the Single Approver definition, noting that several `<role>` tags are excluded from this snippet for brevity:

```
<task>
    <name>review</name>
    <actions>
        <notification>
            <name>Review Notification</name>
            <template>${userName} sent you a ${entryType} for review in the workflow.</template>
            <template-language>freemarker</template-language>
            <notification-type>email</notification-type>
            <notification-type>user-notification</notification-type>
            <execution-type>onAssignment</execution-type>
        </notification>
        <notification>
            <name>Review Completion Notification</name>
            <template><![CDATA[Your submission was reviewed<#if taskComments?has_content> and the reviewer applied the following ${taskComments}</#i
            <template-language>freemarker</template-language>
            <notification-type>email</notification-type>
            <recipients>
                <user />
            </recipients>
            <execution-type>onExit</execution-type>
        </notification>
    </actions>
    <assignments>
        <roles>
            <role>
                <role-type>organization</role-type>
                <name>Organization Administrator</name>
            </role>
            ...
        </roles>
    </assignments>
    <transitions>
        <transition>
            <name>approve</name>
            <target>approved</target>
        </transition>
        <transition>
```

```
            <name>reject</name>
            <target>update</target>
            <default>false</default>
        </transition>
    </transitions>
</task>
```

There are two `actions` in the review task, both `<notification>`s. Each notification may contain a name, template, notification-type, execution-type, and recipients. Besides notifications, You can also use the `<action>` tag. These have a name and a script and are more often used in state nodes than tasks.

Assignments

Workflow tasks are completed by a user. Assignments make sure the right users can access the tasks. You can choose how you want to configure your assignments.

You can choose to add assignments to specific roles, to multiple roles of a role type (organization, site, or regular role types), to the asset creator, to resource actions, or to specific users. Additionally, you can write a script to define the assignment. For an example, see the `single-approver-definition-scripted-assignment.xml`.

```
<assignments>
    <roles>
        <role>
            <role-type>organization</role-type>
            <name>Organization Administrator</name>
        </role>
    </roles>
</assignments>
```

The above assignment specifies that an Organization Administrator must complete the task.

```
<assignments>
    <user>
        <user-id>20156</user-id>
    </user>
</assignments>
```

The above assignment specifies that only the user with the user ID of 20156 may complete the task. Alternatively, specify the `<screen-name>` or `<email-address>` of the user.

```
    <assignments>
        <scripted-assignment>
            <script>
                <![CDATA[
                        import com.liferay.portal.kernel.model.Group;
                        import com.liferay.portal.kernel.model.Role;
                        import com.liferay.portal.kernel.service.GroupLocalServiceUtil;
                        import com.liferay.portal.kernel.service.RoleLocalServiceUtil;
                        import com.liferay.portal.kernel.util.GetterUtil;
                        import com.liferay.portal.kernel.workflow.WorkflowConstants;

                        long companyId = GetterUtil.getLong((String)workflowContext.get(WorkflowConstants.CONTEXT_COMPANY_ID));

                        long groupId = GetterUtil.getLong((String)workflowContext.get(WorkflowConstants.CONTEXT_GROUP_ID));

                        Group group = GroupLocalServiceUtil.getGroup(groupId);

                        roles = new ArrayList<Role>();
```

```
                    Role adminRole = RoleLocalServiceUtil.getRole(companyId, "Administrator");

                    roles.add(adminRole);

                    if (group.isOrganization()) {
                        Role role = RoleLocalServiceUtil.getRole(companyId, "Organization Content Reviewer");

                        roles.add(role);
                    }
                    else {
                        Role role = RoleLocalServiceUtil.getRole(companyId, "Site Content Reviewer");

                        roles.add(role);
                    }

                    user = null;
                ]]>
            </script>
        <script-language>groovy</script-language>
    </scripted-assignment>
</assignments>
```

The above assignment assigns the task to the *Administrator* role, then checks whether the *group* of the asset is an Organization. If it is, the *Organization Content Reviewer* role is assigned to it. If it's not, the task is assigned to the *Site Content Reviewer* role.

Note the `roles = new ArrayList<Role>();` line above. In a scripted assignment, the `roles` variable is where you specify any roles the task is assigned to. For example, when `roles.add(adminRole);` is called, the Administrator role is added to the assignment.

Assigning tasks to Roles, Organizations, or Asset Creators is a straightforward concept, but what does it mean to assign a workflow task to a Resource Action? Imagine an *UPDATE* resource action. If your workflow definition specifies the UPDATE action in an assignment, then anyone who has permission to update the type of asset being processed in the workflow is assigned to the task. You can configure multiple assignments for a task.

Resource Action Assignments

Resource actions are operations performed by users on an application or entity. For example, a user might have permission to update Message Boards Messages. This is called an UPDATE resource action, because the user can update the resource. If you're uncertain about what resource actions are, refer to the developer tutorial on the permission system for a more detailed explanation.

To find all the resource actions that have been created, you need access to the Roles Admin application in the Control Panel (in other words, you need permission for the VIEW action on the roles resource).

- Navigate to Control Panel → Users → Roles.
- Add a new Regular Role. See the article on managing roles for more information.
- Once the role is added, navigate to the Define Permissions interface for the role.
- Find the resource whose action should define your workflow assignment.

Here's what the assignment's XML looks like:

```
<assignments>
    <resource-actions>
        <resource-action>UPDATE</resource-action>
    </resource-actions>
</assignments>
```

Now when the workflow proceeds to the task with the resource action assignment, users with `UPDATE` permission on the resource (for example, Message Boards Messages) are notified of the task and can assign it to themselves (if the notification is set to Task Assignees). Specifically, users see the tasks in their *My Workflow Tasks* application under the tab *Assigned to My Roles*.

Use all upper case letters for resource action names. Here are some common resource actions:

```
UPDATE
ADD
DELETE
VIEW
PERMISSIONS
SUBSCRIBE
ADD_DISCUSSION
```

Determine the probable resource action name from the permissions screen for a resource. For example, in Message Boards, one of the permissions displayed on that screen is *Add Discussion*. Convert that to all uppercase and replace the space with an underscore, and you have the action name.

Task Timers

Task timers trigger an action after a specified time period passes. Timers are useful for ensuring a task does not go unattended for a long time. Available timer actions include sending an additional notification, reassigning the asset, or creating a timer action.

```
<task-timers>
    <task-timer>
        <name></name>
        <delay>
            <duration>1</duration>
            <scale>hour</scale>
        </delay>
        <blocking>false</blocking>
        <recurrence>
            <duration>10</duration>
            <scale>minute</scale>
        </recurrence>
        <timer-actions>
            <timer-notification>
                <name></name>
                <template></template>
                <template-language>text</template-language>
                <notification-type>user-notification</notification-type>
            </timer-notification>
        </timer-actions>
    </task-timer>
</task-timers>
```

The above task timer creates a notification. Specify a time period in the `<delay>` tag, and specify what action to take when the time expires in the `<timer-actions>` block. The `<blocking>` element specifies whether the timer actions may recur. If blocking is set to `false`, timer actions may recur. In a `recurrence` element, specify the recurrence interval using a `duration` and a `scale`, as demonstrated above. The above recurrence element specifies that the timer actions run again every ten minutes after the initial occurrence. Setting blocking to true prevents timer actions from recurring.

```
<timer-actions>
    <reassignments>
       <assignments>
         <roles>
```

```
          <role>
              <role-type></role-type>
              <name></name>
          </role>
          ...
        </roles>
      </assignments>
    </reassignments>
</timer-actions>
```

The above snippet demonstrates how to set up a reassignment action.
Like `<action>` elements, `<timer-action>` elements can contain scripts.

```
<timer-actions>
    <timer-action>
        <name>doSomething</name>
        <description>Do something cool when time runs out.</description>
        <script>
            ...
        </script>
        <script-language>groovy</script-language>
    </timer-action>
</timer-actions>
```

The above example isn't functional but it demonstrates setting up a `<script>` in your task timer. Read the Scripting in Workflow article for more information.

Note: A `timer-action` can contain all the same tags as an `action`, with one exception: `execution-type`. Timer actions are always triggered once the time is up, so specifying and execution type of `onEntry`, for example, isn't meaningful inside a timer.

Tasks are at the core of the workflow definition. Once you understand how to create tasks and the other workflow nodes and add transitions between the nodes, you're on the cusp of workflow wizard-hood.

135.6 Workflow Notifications

While an asset is in a workflow, relevant Users should be notified about certain events, like when a review task is completed. Any workflow node with an `<actions>` element can have notifications.

```
<actions>
    <action>
        <notification>
            <name>Creator Modification Notification</name>
            <template>Your submission was rejected by ${userName}, please modify and resubmit.</template>
            <template-language>freemarker</template-language>
            <notification-type>email</notification-type>
            <notification-type>user-notification</notification-type>
            <execution-type>onAssignment</execution-type>
        </notification>
    </actions>
</actions>
```

The above Creator Modification Notification sends a notification message in two ways: via email and via user notification (this goes to the Notifications widget in the User's Site). The message is defined in a FreeMarker template and sent once a task assignment is created. But who receives the notification? If no recipients are explicitly specified via a `recipients` tag, the asset's creator receives the notification.

Notification Options

There are several elements that can be specified in a <notification>:

Name Set the name of the notification in the <name> element. This information is used to display the notification in the *My Workflow Tasks* widget of a User's personal Site.

Template The <template> element contains the message of the notification. The syntax is determined by the template language you're using.

Template Language Choose from freemarker, velocity, or plain text in the <template-language> tag.

Notification Type Choose whether to send an email, user-notification (via the Notification widget), im (instant message), or private-message in the <notification-type> tag.

```
<notification-type>email</notification-type>
```

Execution Type Choose to link the sending of the notification to entry into the node (onEntry), when a task is assigned (onAssignment), or when the workflow processing is leaving a node (onExit). If you specify a notification to be sent on assignment, the assignee is notified automatically.

Recipients Decide who should receive the notification in the <recipients> tag:

```
<recipients>
    [SEE BELOW FOR THE AVAILABLE RECIPIENT TAGS]
</recipients>
```

Available recipient tags are

- <user>: notify the User that sent the asset through the workflow. Specify the tag as <user />. To notify a specific user, enter the userId:

```
<recipients>
    <user />
</recipients>
<recipients>
    <user>
        <user-id>20139</user-id>
    </user
</recipients>
```

- <roles>: notify specific Roles, either by ID or by their type and name.

```
<recipients>
    <roles>
        <role>
            <role-id>33621</role-id>
        </role>
    </roles>
</recipients>
<recipients>
    <roles>
        <role>
            <role-type>regular</role-type>
```

```
            <name>Power User</name>
            <auto-create>false</auto-create>
        </role>
    </roles>
</recipients>
```

- `<assignees />`: notify the task assignees.
- `<scripted-recipient>`: use a script to identify notification recipients.

```
<recipients>
    <scripted-recipient>
        <script>
            <![CDATA[Script logic goes here]]>
        </script>
        <script-language>groovy</script-language>
    </scripted-recipient>
</recipients>
```

If the notification type is `email`, you can specify the `recipientType` attribute of the `<recipients>` tag as *To, CC,* or *BCC*.

```
<recipients receptionType="cc">
    <roles>
        <role>
            <role-type>regular</role-type>
            <name>Manager</name>
        </role>
    </roles>
</recipients>
```

By default, `recipientType` is `to`.
As always, read the schema for all the details.

135.7 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 two 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, along with methods in the service layer that update them.

You have, therefore, two tasks to enable workflow:

1. Create a `WorkflowHandler` class for your application.
2. Update your service layer to integrate it with workflow.

Time to get started.

Creating a Workflow Handler

The workflow handler should go in the module containing service implementations. It's nice to keep your back-end code separate from your view layer and controller (in the MVC pattern).

1. 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.

   ```
   public class FooEntityWorkflowHandler extends BaseWorkflowHandler<FooEntity>
   ```

2. Since you're publishing a service to be consumed in the OSGi runtime, your workflow handler class must 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 must also declare the type of service being implemented (`WorkflowHandler.class`).

3. There are three methods to override in the workflow handler, and the first two are boilerplate methods:

   ```
   @Override
   public String getClassName() {
       return FooEntity.class.getName();
   }
   ```

   `getClassName` returns the model class's fully qualified class name (`com.my.app.package.model.FooEntity`, for example).

   ```
   @Override
   public String getType(Locale locale) {
       return ResourceActionsUtil.getModelResource(locale, getClassName());
   }
   ```

   `getType` returns the model resource name (`model.resource.com.my.app.package.model.FooEntity`, for example).

   ```
   @Override
   public FooEntity updateStatus(int status, Map<String, Serializable> workflowContext) {
   ```

   Most of the heavy lifting is in the `updateStatus` method. It returns a call to a local service method of the same name (for example, `FooEntityLocalService.updateStatus`), so the status returned from the workflow back-end can be persisted to the entity table in the database.

4. The `updateStatus` method needs 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. The other parameters 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.

Updating the Service Layer

In most Liferay applications, Service Builder is used to create database fields. First, you must update the service layer:

- The service layer must populate the new fields when entities are added to the database.
- The service layer must 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 by workflow status (usually *approved*).

After this is done, the View layer should account for the workflow status of displayed entities.

1. 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" />
```

2. 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 is in the local service implementation class (`-LocalServiceImpl`).

When an entity is added to the database, the application must detect whether workflow is enabled. If not, it automatically marks the entity as approved so it appears in the UI. Otherwise, it's left in draft status and the workflow back-end handles it. Thankfully, this whole process is easily done with a single call to `WorkflowHandlerRegistryUtil.startWorkflowInstance`.

1. At the end of any method that adds a new entity to your database, call the workflow service:

```
WorkflowHandlerRegistryUtil.startWorkflowInstance(fooEntity.getCompanyId(),
        fooEntity.getGroupId(), fooEntity.getUserId(), FooEntity.class.getName(),
        fooEntity.getPrimaryKey(), fooEntity, serviceContext);
```

2. Once you've set the database fields for workflow status and started the workflow instance, implement the `updateStatus` method that must be called 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);
```

3. 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 must change the visibility of the asset depending on its workflow status, so the Asset Publisher doesn't show 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, Workflow updates the asset 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`.

4. 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.

5. 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" />
```

For an example of a fully implemented `updateStatus` method, see the `com.liferay.portlet.blogs.service.impl.Bl` class in `portal-impl`.

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.

Workflow Status and the View Layer

If you have an application with database entities, you're likely displaying them. Once you enable workflow, you should only display approved entities to your end users.

This 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).

1. 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>
```

2. 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);
}
```

3. Finally, 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() %>" />
```

Great! You created one new class, updated your `add` methods, added one new method in the service layer, and updated your view. Workflow is fully implemented and ready to use in your Liferay application.

CHAPTER 136

MANAGING USER-ASSOCIATED DATA STORED BY CUSTOM APPLICATIONS

7.0 makes it possible for administrators to delete or anonymize User Associated Data (UAD), providing a useful tool for compliance with the EU's General Data Protection Regulation (GDPR). Out of the box, this tool only supports Liferay applications (blogs, web content, etc.), but you can also anonymize data stored by your custom apps.

If your app was created using Service Builder, anonymization is easy. Follow these steps:

1. Include dependencies on `com.liferay.petra.string` and `com.liferay.portal.kernel` in your service module's build script.

2. Identify the fields that must be anonymized in the service module's `service.xml` file.

3. Run Service Builder. Provide a build script for the `-uad` module that is generated.

4. Provide your application's name to the anonymization UI. If you skip this step, your app is labeled using the `Bundle-SymbolicName` from the `-uad` module's `bnd.bnd` file.

Anonymization of apps not created using Service Builder will be covered separately.

136.1 Include Dependencies

To compile the code that Service Builder generates, you need dependencies on Petra and 3.23.0 or later of Liferay `kernel` in your service module's `build.gradle`:

```
dependencies {
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.23.0"
    compileOnly group: "com.liferay", name: "com.liferay.petra.string", version: "2.0.0"
    ...
}
```

136.2 Choose Fields to Anonymize

Next you must identify fields to anonymize by attaching anonymization attributes to elements in the `-service` module's `service.xml` file. There are two ways to do this.

- The `uad-anonymize-field-name` attribute indicates a field whose value is replaced by that of the anonymous user in the UAD deletion process.
- The `uad-nonanonymizable` attribute indicates data that cannot be anonymized automatically and must be reviewed by an administrator.

For example, in the blogs application, `uad-anonymize-field-name="fullName"` is appended to the `userName` column in `service.xml`:

```
<column name="userName" type="String" uad-anonymize-field-name="fullName" />
```

This indicates that the user name of a blog entry's author should be replaced by the anonymous user's `fullName`.

The content of a blog post, in contrast, cannot be anonymized automatically:

```
<column name="content" type="String" uad-nonanonymizable="true" />
```

The `uad-nonanonymizable` value of `true` indicates that the `content` field must be reviewed by an administrator to remove a blog author's UAD.

136.3 Run Service Builder!

At this point, you're ready to run Service Builder. This generates a new `-uad` module based on the values you added to `service.xml`. The new module is generated without a build script, so you must provide one. It should include dependencies on `osgi.service.component.annotations`, `kernel`, `Petra`, the `-api` module of the UAD application, as well as your own application's `-api` module. The build script should look like this:

```
dependencies {
  compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.23.0"
  compileOnly group: "com.liferay", name: "com.liferay.user.associated.data.api", version: "3.0.2"
  compileOnly group: "com.liferay", name: "com.liferay.petra.string", version: "1.0.0"
  compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.3.0"
  compileOnly project(":modules:custom:custom-api")
  ...
  }
```

At this point you can compile your application. Before you deploy it, however, you should to make sure the UAD application recognizes it in a way that makes sense to administrators.

Note: Depending on how you created your project—for instance, if you used Blade's Service Builder template rather than Liferay Dev Studio's—you may have to include the new `-uad` module in your `settings.gradle` file before you can compile:

```
include "myapp-api", "myapp-service", "myapp-uad"
```

136.4 Provide Your App's Name to the UI

The simplest way to provide your app's name to the anonymization UI is to include a language key in your `Language.properties` file: `application.name.[Bundle-SymbolicName]=` where the bracketed text is the `Bundle-SymbolicName` from your `-uad` module's `bnd.bnd` file. For example: `application.name.com.liferay.docs.custom.portlet=Custom App`.

That's the recommended approach for custom apps, but if you look at the source code for Liferay DXP itself, you see that it isn't used. Why not? Because it has the downside of creating multiple language keys to label a single application, which can be confusing. To avoid multiplying language keys, Liferay applications use the `com.liferay.lang.merger.` plugin. Here's what it looks like:

```
apply plugin: "com.liferay.lang.merger"

dependencies {

...

}

mergeLang {
    setting("../blogs-web/src/main/resources/content") {
        transformKey "javax.portlet.title.com_liferay_blogs_web_portlet_BlogsPortlet", "application.name.com.liferay.blogs.uad"
    }

    sourceDirs = ["../blogs-web/src/main/resources/content"]
}
```

This is from the `-uad` module's `build.gradle` file in Liferay DXP's Blogs application. The `setting` property identifies the location of the `Language.properties` file (by Gradle convention, the `sourceDirs` property must match `setting`). `transformKey` passes in first the language key for the application's name, and then the `Bundle-SymbolicName` from the `-uad` module's `bnd.bnd` file. The plugin takes the value of the first parameter and assigns it to the second parameter. The end result is that a key from `Language.properties` provides the name of the application to the anonymization UI—but no additional language keys need to be created or maintained.

That's it! You can now delete or anonymize User Associated Data stored by your app.

Chapter 137

Configurable Applications

Many applications must be configurable, whether by end users or administrators. A viable configuration solution must support use cases ranging from setting a location for a weather display to more complex cases like settings for a mail or time sheet application.

The Portlet standard's portlet preferences API can be used for portlet configuration, but it's intended for storing user preferences. This limits its usefulness for enabling administrator configuration; plus it can only be used with portlets. Instead, application developers tend to create ad hoc configuration methods. But this isn't necessary.

There's a full-featured configuration API that's easy to use, and it's not limited to portlets. Any class can use the configuration API to set configuration values in the UI. It's used throughout Liferay DXP's applications. We like it, and we think you'll like it too.

The following tutorials show you how to use it.

CHAPTER 138

MAKING APPLICATIONS CONFIGURABLE

A configurable application allows a user with appropriate permissions to change certain aspects of the application, within bounds set by the developer. Liferay's configuration framework simplifies the task by auto-generating a UI if you define the configuration options in a Java interface. This way, you don't have to create your own application configuration framework.

Complete these three high level tasks to integrate your application with the configuration framework:

1. Provide a way to set configurations in the user interface.
2. Set the scope at which the application can be configured.
3. Read configuration values in your business logic.

This tutorial demonstrates both adding your application's configuration form to the System Settings application in the Control Panel and categorizing the configuration. Subsequent tutorials show you how to

1. Set the *scope* of the configuration. Read more about configuration scope here.
2. Read configuration values from various contexts.

Note: To see a working application configuration, deploy the `configuration-action` Blade sample and navigate to System Settings (*Control Panel → Configuration → System Settings*). Go to *Platform → Third Party*. In the System Scope, open the *Message display configuration* entry and edit the fields as you wish.

Add the *Blade Message Portlet* to a page to test your configuration choices.

You don't need much prior knowledge to use the configuration API, but understanding a few key concepts is useful before diving into the code.

Typed Configuration The method described here uses *typed* configuration. The application configuration isn't just a list of key-value pairs. Values can have types, like `Integer`, a list of `Strings`, a URL, etc. You can even use your own 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. Configuration options should be programmatically explicit, so developers can use autocomplete in modern IDEs to find out all configuration options of a given application or one of its components.

Modularity Modern applications are *modular* and built as a collection of lightweight components.

Configuration Scope If your application must support different configurations at different scopes, the APIs described below handle most of the burden for you. You should still understand the term *configuration scope* even if you don't plan to scope the application's configuration. Here are the most common configuration scopes:

- *System* configurations are unique for the complete installation of the application.
- *Virtual Instance* configurations can vary per virtual instance.
- *Site* configuration can vary per site.
- *Portlet Instance* configurations apply to a single application 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.

138.1 Creating A Configuration Interface

First, you'll learn how to create a configuration at the system scope.

1. 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 the configuration interface for the Liferay Forms application:

```
@Meta.OCD(
    id = "com.liferay.dynamic.data.mapping.form.web.configuration.DDMFormWebConfiguration",
    localization = "content/Language", name = "ddm-form-web-configuration-name"
)
public interface DDMFormWebConfiguration {

    @Meta.AD(
        deflt = "1", description = "autosave-interval-description",
        name = "autosave-interval-name", required = false
    )
    public int autosaveInterval();

    @Meta.AD(
        deflt = "descriptive", name = "default-display-view",
        optionLabels = {"Descriptive", "List"},
        optionValues = {"descriptive", "list"}, required = false
    )
    public String defaultDisplayView();

}
```

   It defines two configuration options, the autosave interval (with a default of one minute) and the default display view, which can be descriptive or list, but defaults to descriptive. Here's what the two Java annotations in the above snippet 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 optional metadata about the field, such as whether it's a required field or if it has a default value. 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 name of the `Meta` class above is `aQute.bnd.annotation.metatype.Meta`. For more information about this class and the `Meta.OCD` and `Meta.AD` annotations, please refer to the bndtools documentation.

   The cool thing about configuration interfaces is that once you have one, you also have an auto-generated UI!

2. 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 as of OSGi standard version R6, they're included in the OSGi core
under the names `@ObjectClassDefinition` and `@AttributeDefinition`. The OSGi
annotations can be used for simple cases like the one described in this
tutorial. However, a key difference between the two libraries is that the bnd
annotations are available at runtime, while the OSGi annotations are not.
Because runtime availability is necessary for some of the Liferay-specific features
described below, we recommend defaulting to the bnd annotations.
```

---

3. 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 provides a lot of information about your application's configuration options. Enough, in fact, to generate a System Settings user interface automatically.

Just by registering a configuration interface, you get a fully capable UI form auto-generated in the System Settings application. By default, configurations are placed in Platform → Third Party. Make it easier to find for your application's users by categorizing the configuration somewhere logical. The next tutorial show you how to do that.

138.2 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, there are even more. To make it easier for portal administrators to find the right configuration options, specify a category for the configuration in the auto-generated System Settings UI.

By default, the following System Settings sections are defined. All available categories are nested beneath these sections:

1. Content Management
2. Social
3. Platform
4. Security
5. Commerce
6. Other

Note: Sections appear if they contain at least one configuration category. Categories appear if they contain at least one configuration. The visible sections and categories depend on the deployed modules.

If you don't specify a category, your application's configuration resides in Platform → Third Party. Usually, you'll want to place your configurations in an existing category or create your own.

.

Specifying a Configuration Category

If you looked in the source code at the Liferay Forms configuration interface (it's in the Forms & Workflow suite's Liferay Dynamic Data Mapping Form Web module), you'll notice something was left out of the code snippet above. The `@Meta.OCD` annotation is directly preceded by

```
@ExtendedObjectClassDefinition(
    category = "dynamic-data-mapping",
    scope = ExtendedObjectClassDefinition.Scope.GROUP
)
```

This annotation does two things:

1. Specifies the `dynamic-data-mapping` category.
2. Sets the scope of the configuration.

The fully qualified class name of the `@ExtendedObjectClassDefinition` class is `com.liferay.portal.configuration.m`

Note: The infrastructure used by System Settings assumes the `configurationPid` is the same as the fully qualified class name of the interface. If they don't match, it can't provide any information through `ExtendedObjectClassConfiguration`.

The `@ExtendedObjectClassDefinition` annotation is distributed through the `com.liferay.portal.configuration.met` module, which you can configure as a dependency.

Creating New Sections and Categories

Configurations should be in the most intuitive location (section and category) so administrators find them. If your configurations don't fit into the existing categories or category sections, create your own by implementing the `ConfigurationCategory` interface.

Here's code that creates the *Content Management* section and the *Dynamic Data Mapping* category:

```
@Component
public class DynamicDataMappingConfigurationCategory
    implements ConfigurationCategory {

    @Override
    public String getCategoryIcon() {
        return "dynamic-data-list";
    }

    @Override
    public String getCategoryKey() {
        return _KEY;
    }

    @Override
    public String getCategorySection() {
        return _CATEGORY_SET_KEY;
    }

    private static final String _CATEGORY_SET_KEY = "content-management";

    private static final String _KEY = "dynamic-data-mapping";

}
```

The `getCategorySection` method returns the String with the new section's key. Similarly, `getCategoryKey` returns the key for the new category. Provide localized values for these keys in your module's `src/main/resources/content/Language.properties` file.

Next you'll specify the scope of your application's configuration.

138.3 Scoping Configurations

Applications can have different configurations depending on the scope: per virtual instance (a.k.a. `Company`), site (a.k.a. `Group`), or portlet instance. The Configuration Provider API (based on the standard OSGi Configuration Admin API shown in the previous section) handles this for you.

Scoping the configuration is specifying the scope where the configuration values are set or overridden. Anything set at a less granular scope is just a default for the configuration. It can always be overridden at the configuration's current scope. For example, a site scoped configuration can have its defaults set at the system scope (via System Settings). However, once the configuration is changed at the site scope, it ignores the higher level scope forever. It can also be configured in other places at the same scope. From the database level, this means there could be multiple configuration values for the application, all scoped to the site level, because the values set in one site don't matter if the context in which you need the value is a different site. This is covered in more detail here.

Here's how to scope a configuration:

1. Set the scope in the configuration interface.

2. Enable the configuration for scoped retrieval by creating a configuration bean declaration.

3. Retrieve scoped configurations with a configuration provider.

The third step is covered in the configuration provider tutorial. This article covers the first two steps. Start by setting the scope in the configuration interface.

Step 1: Setting the Configuration Scope

Use the `@ExtendedObjectClassDefinition` annotation to specify the configuration's scope. The scope you choose must match how the configuration object is retrieved through the configuration provider configuration provider. Pass one of these valid scope options to `@ExtendedObjectClassDefinition`:

`Scope.GROUP`: for site scope `Scope.COMPANY`: for virtual instance scope `Scope.SYSTEM`: for system scope `Scope.PORTLET_INSTANCE`: for the portlet instance scope

Here is an example:

```
@ExtendedObjectClassDefinition(
    category = "dynamic-data-mapping",
    scope = ExtendedObjectClassDefinition.Scope.GROUP
)
@Meta.OCD(
    id = "com.liferay.dynamic.data.mapping.form.web.configuration.
        DDMFormWebConfiguration",
    localization = "content/Language",
    name = "ddm-form-web-configuration-name"
)

public interface DDMFormWebConfiguration {
```

The scope property makes it appear in System Settings so an administrator can change its value. In future releases it may serve additional purposes.

Step 2: Enabling the Configuration for Scoped Retrieval

If you set the configuration scope, you must retrieve the configuration values from the same scope. To retrieve a scoped configuration, use a Configuration Provider:

```
JournalGroupServiceConfiguration configuration =
    configurationProvider.getGroupConfiguration(
        JournalGroupServiceConfiguration.class, groupId);
```

This is an example from the Journal module that gets a site-scoped configuration from the configuration provider. To enable scoped retrieval of a configuration, the application's configuration must be registered with a `ConfigurationBeanDeclaration`.

Note: `ConfigurationProvider` is part of Liferay's kernel API so you don't need a new dependency to use it. However, its implementation is distributed as a module called `portal-configuration-module-configuration`, so make sure it is installed.

To create a configuration bean declaration:

1. Register the configuration class by implementing `ConfigurationBeanDeclaration`.

    ```
    @Component
    public class JournalGroupServiceConfigurationBeanDeclaration
        implements ConfigurationBeanDeclaration {
    ```

2. This class has one method that returns the class of the interface you created in the previous section. It enables the system to keep track of configuration changes as they happen, making requests for the configuration very fast.

```
@Override
public Class<?> getConfigurationBeanClass() {
    return JournalGroupServiceConfiguration.class;
}
```

Step 3: Retrieving Scoped Configurations

If you set the configuration scope, then you must retrieve the configuration values from the same scope. To retrieve a scoped configuration, use a Configuration Provider:

```
JournalGroupServiceConfiguration configuration =
    configurationProvider.getGroupConfiguration(
        JournalGroupServiceConfiguration.class, groupId);
```

This is an example from the Journal module that gets a site-scoped configuration from the configuration provider. The `groupId` variable is important since it identifies which site the configuration value should be read from.

That's all there is to it. Now the configuration is scoped and supports scoped retrieval.

138.4 Reading Configuration Values from a Component

Once you have the application configured so that administrators can configure it in System Settings, you might be wondering how to read the configuration from your application's Java code.

The answer is that there are several ways. Which one you use depends on the context from which the configuration must be read:

1. From any Component class
2. From an MVC portlet's JSP
3. From an MVC portlet's Portlet class
4. From a non-Component class

This tutorial shows the first usage, reading the configuration from a Component class.

Reading Configurations from a Component Class

1. First set the `configurationPid` Component property as the fully qualified class name of the configuration class:

```
@Component(configurationPid = "com.liferay.dynamic.data.mapping.form.web.configuration.DDMFormWebConfiguration")
```

2. Then provide an `activate` method, annotated with `@Activate` to ensure the method is invoked as soon as the Component is started, and `@Modified` so it's invoked whenever the configuration is modified.

```
@Activate
@Modified
protected void activate(Map<String, Object> properties) {
    _formWebConfiguration = ConfigurableUtil.createConfigurable(
        DDMFormWebConfiguration.class, properties);
}

private volatile DDMFormWebConfiguration _formWebConfiguration;
```

The `activate()` method calls the method `ConfigurableUtil.createConfigurable()` to convert a map of the configuration's properties to a typed class, which is easier to handle. The configuration is stored in a `volatile` field. Don't forget to make it `volatile` to prevent thread safety problems.

3. Once the activate method is set up, retrieve particular properties from the configuration wherever they're needed:

```
public void orderCar(String model) {
    order("car", model, _configuration.favoriteColor);
}
```

This is dummy code: don't try to find it in the Liferay source code. The String configuration value of `favoriteColor` is passed to the `order` method call, presumably so that whatever model car is ordered will be ordered in the configured favorite color.

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.metatyp` 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.

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.

138.5 Reading Configuration Values from a MVC Portlet

It's very common to read configuration values from a portlet class or its JSPs. This tutorial shows how to add a configuration to the request and read it from the view layer, and how to read it directly in the portlet class. This tutorial uses dummy code from a portlet we'll call the Example Configuration Portlet. The import statements are included in the code snippets so that you can see the fully qualified class names (FQCNs) of all the classes that are used.

Accessing the Configuration Object in the Portlet Class

Whether you need the configuration values in the portlet class or the JSPs, the first step is to get access to the configuration object in the `*Portlet` class.

1. Imports first:

```
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;
```

2. Portlet classes are Component classes. To mate the configuration with the Component, provide the `configurationPid` property with the FQCN of the configuration class.

```
@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 {
```

Note that you can specify more than one configuration PID here, by enclosing the values in curly braces ({}) and placing commas between each PID.

3. Write an `activate` method annotated with `@Activate` and `@Modified`. See the Making Applications Configurable tutorial if you're unsure why these annotations are necessary:

```
@Activate
@Modified
protected void activate(Map<String, Object> properties) {
    _configuration = ConfigurableUtil.createConfigurable(
    ExampleConfiguration.class, properties);
}

private volatile ExampleConfiguration _configuration;
```

A volatile field `_configuration` is created by the `createConfigurable` method. Now the field can be used to retrieve configuration values or to set the values in the request, so they can be retrieved in the application's JSPs.

Accessing the Configuration from a JSP

In the case of reading from a JSP, add the configuration object to the request object so its values can be read from the JSPs that comprise the application's view layer.

Note: There's a 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` article to learn about the shorter method.

---

1. Add the configuration object to the request. Here's what it looks like in a simple portlet's `doView` method:

```
@Override
public void doView(RenderRequest renderRequest,
    RenderResponse renderResponse) throws IOException, PortletException {

    renderRequest.setAttribute(
        ExampleConfiguration.class.getName(), _configuration);

    super.doView(renderRequest, renderResponse);
}
```

   The main difference between this example and the component class covered in the previous tutorial is that this class is a portlet class and it sets the configuration object as a request attribute in its `doView()` method.

2. Read configuration values from a JSP. First add these imports to the top of your `view.jsp` file:

```
<%@ page import="com.liferay.docs.exampleconfig.ExampleConfiguration" %>
<%@ page import="com.liferay.portal.kernel.util.GetterUtil" %>
```

3. In the JSP, obtain the configuration object from the request object and read the desired configuration value from it. Here's a `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:

```
Favorite color: blue
```

The word *blue* is 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 *Control Panel → Configuration → System Settings → Platform → Third Party* and click on the *Example configuration* link, you can find the `Favorite color` setting and change its value. The JSP reads the configuration, and refreshing the UI reflects this update.

Accessing the Configuration from the Portlet Class

Now that you've seen a detailed example of accessing the configuration values in a JSP, there's not much more to cover when accessing the configuration directly in the `-Portlet` class. Wherever you require the value of a configuration property, call `_configuration.propertyName` and you have access to the currently configured value. For example, this code compares the `favoriteColor` configuration value with a `userFavoriteColor` that's fetched from the request object:

```
public boolean isFavoriteColorMatched {

    String userFavoriteColor = ParamUtil.getString(request, "userFavoriteColor");

    if (_configuration.favoriteColor == userFavoriteColor) {

        SessionMessages.add(request, "congratulateUser");

        return true;
    }

    return false;
}
```

It returns true and adds a success message if the two Strings match each other, but you can do anything that makes sense for your application's controller logic.

That's all there is to reading configuration values in an MVC Portlet. The next tutorial covers categorizing the configuration and accessing it through a Configuration Provider.

138.6 Reading Configuration Values from a Configuration Provider

When an application is deployed, 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. Achieve this with little effort using the Configuration Provider API that is based on the standard OSGi Configuration Admin API.

Using the Configuration Provider

When using the Configuration Provider, instead of receiving the configuration directly, the class that wants to access it must

1. Receive a `ConfigurationProvider` to obtain the configuration
2. Be registered with a `ConfigurationBeanDeclaration`.

The tutorial on scoping configurations demonstrates how to register the configuration with a `ConfigurationBeanDeclaration`.

After registering with a `ConfigurationBeanDeclaration`, you're ready to use a `ConfigurationProvider` to retrieve the scoped configuration. Here's how you obtain a reference to it:

1. Here's the approach for components:

```
@Reference
protected void setConfigurationProvider(ConfigurationProvider configurationProvider) {
    _configurationProvider = configurationProvider;
}
```

2. Here's the approach for Service Builder services:

```
@ServiceReference(type = ConfigurationProvider.class)
protected ConfigurationProvider configurationProvider;
```

3. 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.

4. For anything else, call the same methods from the utility class, `ConfigurationProviderUtil`. Be sure you call the utility methods in contexts where the portal is guaranteed to be initialized prior to the method call. This class is useful in the scripting console, for example. Here's an example method that uses the utility class. It comes from the export-import service, which is only called during the import and export of content from a running portal:

   protected boolean isValidateLayoutReferences() throws PortalException { long companyId = CompanyThreadLocal.getCompanyId();

```
ExportImportServiceConfiguration exportImportServiceConfiguration =
    ConfigurationProviderUtil.getCompanyConfiguration(
        ExportImportServiceConfiguration.class, companyId);

return exportImportServiceConfiguration.validateLayoutReferences();


}
```

To retrieve the configuration, use 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 must 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 a 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 a nifty way to to access a portlet instance configuration from a JSP.

Accessing the Portlet Instance Configuration Through the PortletDisplay

Often you must access portlet instance settings from a JSP or from a Java class that isn't an OSGi component. To read the settings in these cases, a method was added to `PortletDisplay`, which is 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.

138.7 Customizing the System Settings User Interface

Liferay DXP applications use the Apache Felix Configuration Admin Service to provide application configuration. By specifying a single Configuration Interface class, the configuration data is typed and scoped, and the application gains an auto-generated configuration user interface, available in Control Panel → Configuration → System Settings once the configuration is registered. If this is new information for you, consider first reading the set of tutorials on Making Applications Configurable.

This tutorial describes how to customize the System Settings entry's user interface in the following ways:

- Provide a custom form for a configuration object.
- Write a completely custom configuration UI. This is useful especially if you aren't using the Configuration Admin service or any of Liferay's Configuration APIs.
- Exclude a configuration object from System Settings. If you're providing a completely custom configuration UI but are still using Configuration Admin, you'll want to hide the auto-generated UI. If your configuration is not meant to be accessible to administrative Users (perhaps because it's too low level), you might want to exclude it from the System Settings UI.

Providing Custom Configuration Forms

This method relies on an existing Config Admin configuration class, as described here. Here's an example configuration class, from Liferay's own Currency Converter application:

```
@ExtendedObjectClassDefinition(category = "localization")
@Meta.OCD(
    id = "com.liferay.currency.converter.web.configuration.CurrencyConverterConfiguration",
    localization = "content/Language",
    name = "currency-converter-configuration-name"
```

```
)
public interface CurrencyConverterConfiguration {

    @Meta.AD(deflt = "GBP|CNY|EUR|JPY|USD", name = "symbols", required = false)
    public String[] symbols();
}
```

There's one configuration option, symbols, that takes an array of values.

All that's necessary to customize an auto-generated form is one additional class, an implementation of the `ConfigurationFormRenderer` interface.

Implement its three methods:

1. `getPid`: Return the configuration object's ID. This is defined in the `id` property in the `*Configuration` class's `@Meta.OCD` annotation.

2. `getRequestParameters`: Read the parameters sent by the custom form and put them in a Map whose keys should be the names of the fields of the Configuration interface.

3. `render`: Render the custom form's fields, using your desired method (for example, JSPs or another template mechanism). The `<form>` tag itself is provided automatically and shouldn't be included in the `ConfigurationFormRenderer`.

Here's a complete `ConfigurationFormRenderer` implementation:

```
@Component(immediate = true, service = ConfigurationFormRenderer.class)
public class CurrencyConverterConfigurationFormRenderer
    implements ConfigurationFormRenderer {

    @Override
    public String getPid() {
        return "com.liferay.currency.converter.web.configuration.CurrencyConverterConfiguration";
    }

    @Override
    public void render(HttpServletRequest request, HttpServletResponse response)
            throws IOException {

        String formHtml = "<input name=\"mysymbols\" />";

        PrintWriter writer = response.getWriter();

        writer.print(formHtml);

    }

    @Override
    public Map<String, Object> getRequestParameters(
            HttpServletRequest request) {

        Map<String, Object> params = new HashMap<>();

        String[] mysymbols = ParamUtil.getParameterValues(request, "mysymbols");

        params.put("symbols", mysymbols);

        return params;
    }
}
```

The above example generates a custom rendering (HTML) for the form in the `render()` method and reads the information entered in the custom form in the `getRequestParameters()` method.

To see a complete demonstration, including JSP markup, read the dedicated tutorial on creating a configuration form renderer.

Creating a Completely Custom Configuration UI

In some cases, you want a completely custom UI for your configuration. For example:

- Your application doesn't use Config Admin to provide its configuration. You have a completely different configuration backend, and you'll write a completely independent frontend.
- Your application needs more flexibility in its UI, such as multiple configuration screens.

To accomplish this, write a `ConfigurationScreen` implementation.
At a high level you must

1. Write a Component that declares itself an implementation of the `ConfigurationScreen` interface.
2. Implement `Configurationscreen`'s methods.
3. Create the UI by hand.

Here's an example implementation:

```
@Component(immediate = true, service = ConfigurationScreen.class)
public class SampleConfigurationScreen implements ConfigurationScreen {
```

First declare the class an implementation of `ConfigurationScreen`.

```
@Override
public String getCategoryKey() {

    return "third-party";

}

@Override
public String getKey() {

    return "sample-configuration-screen";

}

@Override
public String getName(Locale locale) {

    return "Sample Configuration Screen";

}
```

Second, set the category key, the configuration entry's key, and its localized name. This example puts the configuration entry, keyed `sample-cofniguration-screen`, into the `third-party` System Settings section. The String that appears in System Settings is *Sample Configuration Screen*.

```
@Override
public String getScope() {

    return "system";

}
```

Third, set the configuration scope.

```
@Override
public void render(HttpServletRequest request, HttpServletResponse response)
        throws IOException {

    _jspRenderer.renderJSP( _servletContext, request, response,
    "/sample_configuration_screen.jsp");

}

@Reference private JSPRenderer _jspRenderer;

@Reference(
    target ="(osgi.web.symbolicname=com.liferay.currency.converter.web)",
    unbind = "-")
private ServletContext _servletContext;
```

The most important step is to write the `render` method.This example relies on the `JSPRenderer` service to delegate rendering to a JSP.

It's beyond the scope of this tutorial to write the JSP markup. A separate tutorial will provide a complete demonstration of the `ConfigurationScreen` and implementation and the JSP markup to demonstrate its usage.

Excluding a Configuration UI from System Settings

Providing a custom UI in System Settings is well and good, but what if you instead must exclude your configuration from the System Settings UI? For instance, if you're using Config Admin but also providing a `ConfigurationScreen` implementation and a custom JSP, you'll get two System Settings entries: the custom one you wrote *and* the auto-generated UI from Config Admin. Other times, a configuration is required to be present for back-end developers but isn't intended to be changed in the UI.

To exclude the UI entry, use the `ExtendedObjectClassDefinition` annotation property called `generateUI`. It defaults to `true`, so set it to `false` to suppress the auto-generated UI. Here is an example:

```
@ExtendedObjectClassDefinition(generateUI=false)
@Meta.OCD(
  id = "com.foo.bar.LowLevelConfiguration",
)
public interface LowLevelConfiguration {

  public String[] foo();
  public String bar();

}
```

Now the configuration is available to be managed programmatically or via .config file, but not via the System Settings UI.

138.8 Configuration Form Renderer

There are various approaches to customizing the auto-generated System Settings UI for your configurable application. To replace an application's auto-generated configuration screen with a form built from scratch, you follow these steps:

1. Use a `DisplayContext` class to transfer data between back-end code and the desired JSP markup.
2. Implement the `ConfigurationFormRenderer` interface.
3. Render the configuration form. This tutorial demonstrates the use of a JSP and the previously created `DisplayContext` class.

A generalized discussion on System Settings UI customization is found in a separate tutorial.

This tutorial demonstrates replacing the configuration UI for the *Language Template* System Settings entry, found in Control Panel → Configuration → System Settings → Localization → Language Template. The same steps apply when replacing your custom application's auto-generated UI.

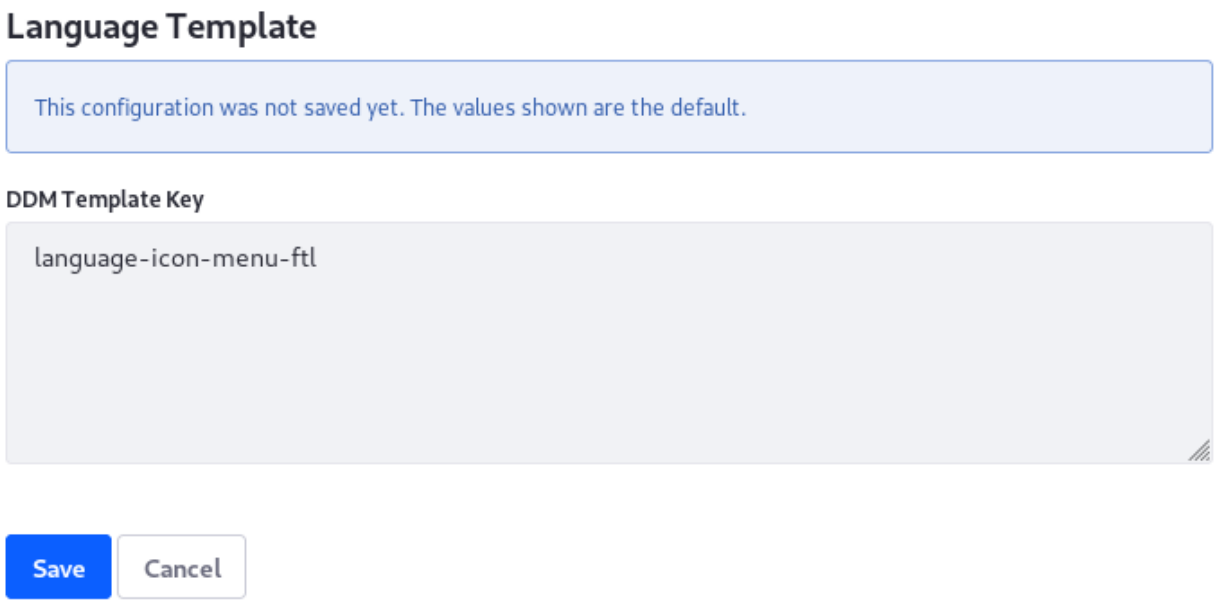


Figure 138.1: The auto-generated UI for the Language Template configuration screen is sub-optimal. A select list with more human readable options is preferable.

Specifically, the text input field labeled *DDM Template Key* in the auto-generated UI is replaced with a select list field type called *Language Selection Style*, populated with all possible DDM Template Keys.

Creating a `DisplayContext`

A `DisplayContext` class is a POJO that simplifies and minimizes the use of Java logic in JSPs. Display context usage isn't required, but it's a nice convention to follow. It's a kind of data transfer object, where the `DisplayContext`'s setters are called from the Java class providing the render logic (in this case the `ConfigurationFormRenderer`'s `render` method), and the getters are called from the JSP, removing the need for Java logic to be written inside the JSP itself.

For this example, create a `LanguageTemplateConfigurationDisplayContext` class with these contents:

```
public class LanguageTemplateConfigurationDisplayContext {

    public void addTemplateValue(
        String templateKey, String templateDisplayName) {
```

```
        _templateValues.add(new String[] {templateKey, templateDisplayName});
    }

    public String getCurrentTemplateName() {
        return _currentTemplateName;
    }

    public String getFieldLabel() {
        return _fieldLabel;
    }

    public List<String[]> getTemplateValues() {
        return _templateValues;
    }

    public void setCurrentTemplateName(String currentTemplateName) {
        _currentTemplateName = currentTemplateName;
    }

    public void setFieldLabel(String fieldLabel) {
        _fieldLabel = fieldLabel;
    }

    private String _currentTemplateName;
    private String _fieldLabel;
    private final List<String[]> _templateValues = new ArrayList<>();

}
```

Next implement the `ConfigurationFormRenderer`.

Implementing a `ConfigurationFormRenderer`

First create the component and class declarations. Set the `service` property to `ConfigurationFormRenderer.class`:

```
@Component(
    configurationPid = "com.liferay.site.navigation.language.web.configuration.SiteNavigationLanguageWebTemplateConfiguration",
    immediate = true, service = ConfigurationFormRenderer.class
)
public class LanguageTemplateConfigurationFormRenderer
    implements ConfigurationFormRenderer {
```

Next, write an `activate` method (decorated with `@Activate` and `@Modified`) to to convert a map of the configuration's properties to a typed class. The configuration is stored in a volatile field. Don't forget to make it volatile to prevent thread safety problems. See the article on reading configuration values from a component class for more information.

```
@Activate
@Modified
public void activate(Map<String, Object> properties) {
    _siteNavigationLanguageWebTemplateConfiguration =
        ConfigurableUtil.createConfigurable(
            SiteNavigationLanguageWebTemplateConfiguration.class,
            properties);
}

private volatile SiteNavigationLanguageWebTemplateConfiguration
    _siteNavigationLanguageWebTemplateConfiguration;
```

Next override the `getPid` and `getRequestParameters` methods:

```
@Override
public String getPid() {
    return "com.liferay.site.navigation.language.web.configuration." +
        "SiteNavigationLanguageWebTemplateConfiguration";
}
```

Return the full configuration ID, as specified in the *Configuration class's @Meta.OCD annotation.

```
@Override
public Map<String, Object> getRequestParameters(
    HttpServletRequest request) {

    Map<String, Object> params = new HashMap<>();

    String ddmTemplateKey = ParamUtil.getString(request, "ddmTemplateKey");

    params.put("ddmTemplateKey", ddmTemplateKey);

    return params;
}
```

In the getRequestParameters method, map the parameters sent by the custom form (obtained from the request) to the keys of the fields in the Configuration interface.

Provide the render logic via the overridden render method. The rendering approach demonstrated here uses a JSP. Recall that it's backed by a DisplayContext class set into the request object. The values set from this render method are available in the JSP via the DisplayContext object's getters.

Loop through the DDM Template Keys for the given groupId and set them into the display context with the addTemplateKey method. Then set the other necessary values that the JSP needs. In this case, set the title, the field label, and the redirect URL. Finally, call renderJSP and pass in the servletContext, request, response, and the path to the JSP:

```
@Override
public void render(HttpServletRequest request, HttpServletResponse response)
    throws IOException {

    Locale locale = LocaleThreadLocal.getThemeDisplayLocale();

    LanguageTemplateConfigurationDisplayContext
        languageTemplateConfigurationDisplayContext =
            new LanguageTemplateConfigurationDisplayContext();

    languageTemplateConfigurationDisplayContext.setCurrentTemplateName(
        _siteNavigationLanguageWebTemplateConfiguration.ddmTemplateKey());

    long groupId = 0;

    Group group = _groupLocalService.fetchCompanyGroup(
        CompanyThreadLocal.getCompanyId());

    if (group != null) {
        groupId = group.getGroupId();
    }

    List<DDMTemplate> ddmTemplates = _ddmTemplateLocalService.getTemplates(
        groupId, _portal.getClassNameId(LanguageEntry.class));

    for (DDMTemplate ddmTemplate : ddmTemplates) {
        languageTemplateConfigurationDisplayContext.addTemplateValue(
            ddmTemplate.getTemplateKey(), ddmTemplate.getName(locale));
    }

    languageTemplateConfigurationDisplayContext.setFieldLabel(
```

```
        LanguageUtil.get(
            ResourceBundleUtil.getBundle(
                locale, LanguageTemplateConfigurationFormRenderer.class),
            "language-selection-style"));

    request.setAttribute(
        LanguageTemplateConfigurationDisplayContext.class.getName(),
        languageTemplateConfigurationDisplayContext);

    _jspRenderer.renderJSP(
        _servletContext, request, response,
        "/configuration/site_navigation_language_web_template.jsp");
}
```

Specify the required service references at the bottom of the class. Be careful to target the proper servlet context, passing the `bundle-SymbolicName` of the module (found in its `bnd.bnd` file) into the `osgi.web.symbolicname` property of the reference target:

```
@Reference
private DDMTemplateLocalService _ddmTemplateLocalService;

@Reference
private GroupLocalService _groupLocalService;

@Reference
private JSPRenderer _jspRenderer;

@Reference
private Portal _portal;

@Reference(
    target = "(osgi.web.symbolicname=com.liferay.site.navigation.language.web)",
    unbind = "-"
)
private ServletContext _servletContext;
```

Once the configuration form renderer is implemented, you can write the JSP markup for the form.

Writing the JSP Markup

Now write the JSP:

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

<%
LanguageTemplateConfigurationDisplayContext
    languageTemplateConfigurationDisplayContext = (LanguageTemplateConfigurationDisplayContext)request.getAttribute(LanguageTemplateConfigurationDisp

String currentTemplateName = languageTemplateConfigurationDisplayContext.getCurrentTemplateName();
%>

<aui:select label="<%= HtmlUtil.escape(languageTemplateConfigurationDisplayContext.getFieldLabel()) %>" name="ddmTemplateKey" value="<%= currentTempl

    <%
    for (String[] templateValue : languageTemplateConfigurationDisplayContext.getTemplateValues()) {
    %>

        <aui:option label="<%= templateValue[1] %>" selected="<%= currentTemplateName.equals(templateValue[0]) %>" value="<%= templateValue[0] %>" /

    <%
    }
    %>

</aui:select>
```

The opening scriptlet gets the display context object from the request so that all its getters are invoked whenever information from the back-end is required. Right away, the `getCurrentTemplateName` method is called, since the current template name is needed for the first option's `ddmTemplateKey` display value as soon as the form is rendered. This happens in the `<aui:select>` tag. There's just a bit of logic used to create an option for each of the available DDM templates that can be chosen.

So what does this example look like when all is said and done?

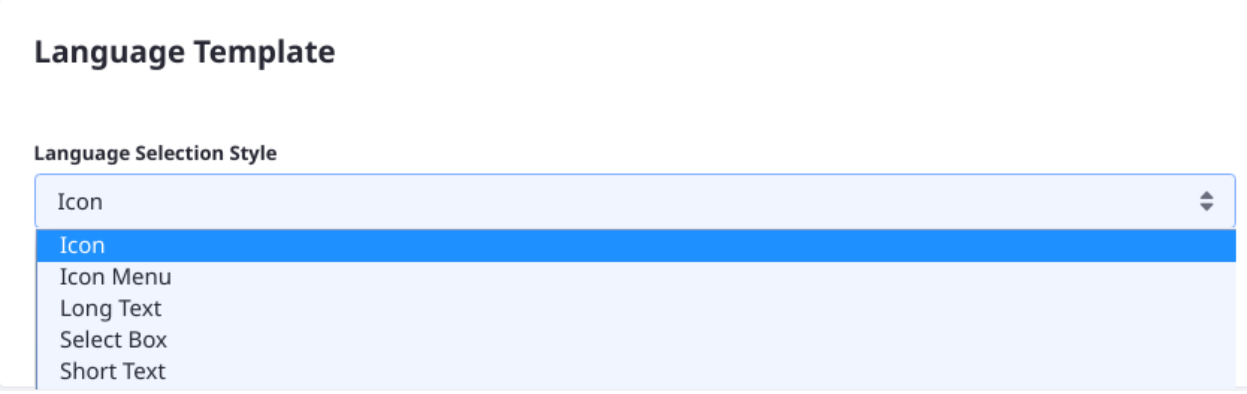


Figure 138.2: A select list provides a more user friendly configuration experience than a text field.

Some configuration UIs require tweaking with a `ConfigurationFormRenderer`. This tutorial shows a particularly good example. Administrators encountering the Language Template entry in System Settings won't know the DDM Template Keys they can use offhand. Providing the available values in a select field wildly enhances the user experience.

CHAPTER 139

INTERNATIONALIZATION

Localizing content and designing apps for different locales is a straightforward process. You can centralize messages (language keys) and translate them manually or automatically, including form localization and setting text in either direction (left-to-right or right-to-left). Customizing messages in apps is easy too. Read on to learn how to internationalize your applications.

139.1 Localizing Your Application

If you're writing a Liferay Application, you're probably a genius who is also really cool, which means your application will be used throughout the entire world. At least, if its messages can be translated into their language, it will. Thankfully, Liferay makes it easy to support translation of your application's language keys.

Note: Even if you don't plan to translate your application 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 Liferay DXP's 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 the translations (for example, enter `localhost:8080/es`).

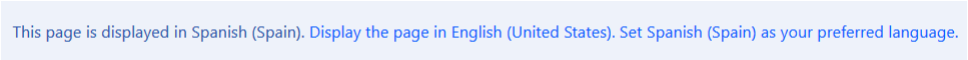


Figure 139.1: Append the locale to your running Liferay's URL and see Liferay's translation power in action.

Language keys are just keys to use in place of a hard coded, fully translated String value in your user interface code. You use a language key in your JSP with a `<liferay-ui:message />` tag.

If you wanted to hard code a message, you'd use the tag like this:

```
<liferay-ui:message key="Howdy, Partner!" />
```

In that case you'll get a properly capitalized and punctuated message in your application.

Instead, 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!
```

Either way, you get the same output. The properties file lets you put all your messages in one place, and you can add additional language properties files with translations later. You just need to make sure there's a locale that corresponds to your translation.

The values from your default `Language.properties` file appear if no locale is specified. If a locale is specified, a key from a file corresponding to that local is retrieved. 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) are used.

What Locales are Available By Default?

There are a bunch of locales available by default in Liferay. Look in the `portal.properties` 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 containing the translated keys (for example, `Language_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, 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 must do to point Liferay DXP at your translations. Users see the translations for the locales they select.

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.b
    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 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=my
servlet.context.name=myapp-web
```

The example `Provide-Capability` header has two parts:

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 Liferay's 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.

139.2 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. In this tutorial, you'll explore the following capabilities:

- Generating language properties files for each supported locale with a single command. This prevents you from having to create a language properties file for each locale manually. The same command also propagates the keys from the default language file to all translation files.
- Generating automatic translations using Microsoft's Translator Text API. This prevents you from translating each message manually.

Generating Language Files for Supported Locales

If you want to generate files automatically for all supported locales, 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 it in your build script's classpath. If you're using Liferay Workspace, the Lang Builder is already available to your modules.

   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"
       }
   }
   ```

2. Create (if necessary) a default `Language.properties` file in the `src/main/resources/content` folder.

3. Run the `gradlew buildLang` task from your project's root folder to generate default translation files.

   The generated files contain copies of all the keys and values in your default `Language.properties` 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 provide credentials to enable translation services.

Now you can begin 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 can now generate language files and provide automatic translations of your language keys.

139.3 Using Liferay's Language Settings

For a given locale, you can override Liferay DXP's core UI messages. Modifying language key values 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 `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 (left to right or right to left).

To see how these settings are configured, open Liferay DXP's core `Language.properties` file in one of two ways:

1. From Liferay Portal's source code. Navigate to

   ```
   liferay-portal/portal-impl/src/content/Language.properties
   ```

2. From a 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.default.portrait=initials
lang.user.initials.field.names=first-name,last-name
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 DXP'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

Customers come from all over the world, so naming conventions are different between locales. Because of this, user name fields are configurable in the following ways:

- Remove certain name fields and make others appear more than once. Some locales 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. Because of this, take these two points into consideration when configuring a locale's user name settings:

- 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
  ```

- Because a 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 a 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 (i.e., `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
```

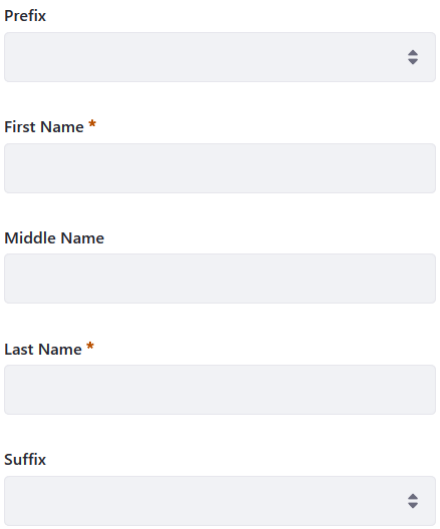


Figure 139.2: The user name settings impact the way user information and forms appear in Liferay.

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
```

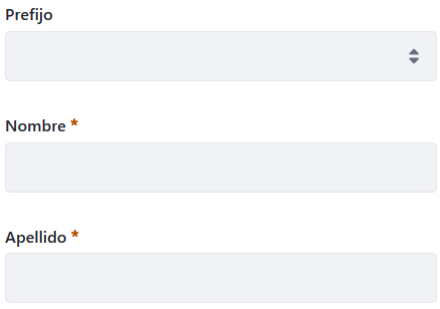


Figure 139.3: The Spanish user name settings omit the suffix and middle name fields entirely.

The biggest difference between the English and Spanish form fields 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 DXP's UI messages are localized: overriding one of its Lanuguage_xx.properties files.

Identifying User Initials

The default avatar displays a user's initials. Some cultures use initials differently, so there's a way to configure them in the Language.properties file.

```
lang.user.default.portrait=initials
lang.user.initials.field.names=first-name,last-name
```

The lang.user.default.portrait property sets the type of portrait to use for users. This can be set to initials or image. If set to image, the default images defined by the image.default.user.female.portrait or image.default.user.male.portrait properties residing in the portal.properties file are used. Therefore, the lang.user.initials.field.names property is ignored.

Figure 139.4: The user's initials are displayed for their avatar by default.

If you're leveraging the user's initials for the default avatar, the lang.user.initials.field.names property is used to organize how the initials are displayed. Valid values for this property include first-name, middle-name, and last-name, in any order.

Now you can manage how a user's initials are displayed!

Right to Left or Left to Right?

The first three properties in the Language.properties's Language Settings section change the direction in which the language's characters are displayed. Most languages are read from left to right, but some languages are read from right to left (e.g., Arabic, Hebrew, and Persian). You can also change it 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
```

With these customizations, you can transform your UI into a user-friendly environment no matter where your users are from.

Note: You can prevent specific CSS rules from transforming (flipping) with the /* @noflip */ decoration. Place the decoration to the left of the CSS rule to apply it. For example, this rule gives a left margin of 20em to the body no matter if the selected language is LTR or RTL:

```
/* @noflip */ body {
 margin-left: 20em;
}
```

You can also use the `.rtl` CSS selector for rules that exclusively apply to RTL languages.

CHAPTER 140

APPLICATION DISPLAY TEMPLATES

In the past, when you needed to modify the UI of a widget, you had to use a hook (e.g., HTML-related change) or a theme (e.g., CSS-related change). It would be nice to apply particular display changes to specific widget 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 widgets.

Be of good cheer! That's precisely what Application Display Templates (ADTs) provide–the ability to customize the way widgets appear on a page, removing limitations to the way your site's content is displayed. With ADTs, you can define custom display templates used to render asset-centric widgets. This isn't actually a new concept in Liferay DXP; some widgets 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 7.1 are

- *Asset Publisher*
- *Blogs*
- *Breadcrumb*
- *Categories Navigation*
- *Language Selector*
- *Media Gallery*
- *Navigation Menu*
- *RSS Publisher*
- *Site Map*
- *Tags Navigation*
- *Wiki*

Continue on to add support for ADTs in your custom portlet.

140.1 Implementing Application Display Templates

Application Display Templates (ADTs) let you add custom display templates to your widgets from the portal. The figure below shows what the Display Template option looks like in a widget's Configuration menu.

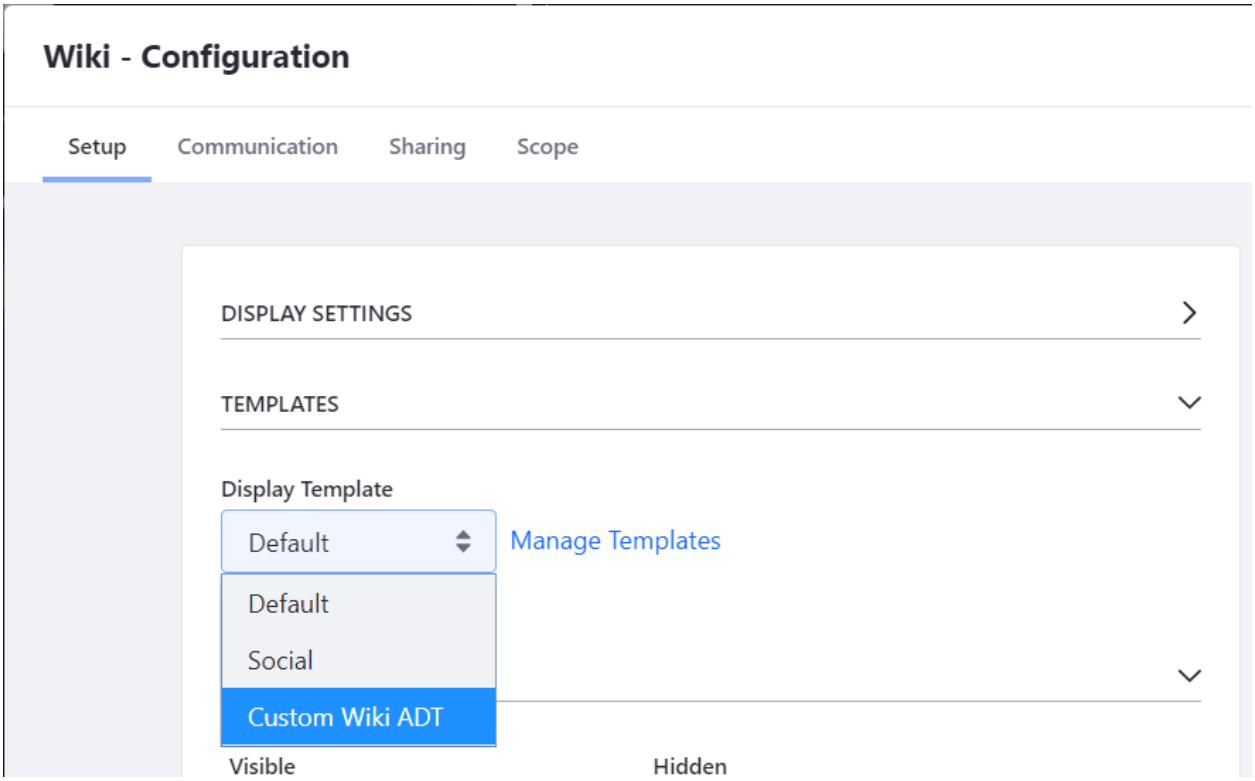


Figure 140.1: By using a custom display template, your portlet's display can be customized.

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 must follow. These steps involve

- registering your portlet to use ADTs
- defining permissions
- exposing the ADT functionality to users

You'll walk through these steps next.

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.

   The `@Component` annotation ties your handler to a specific portlet by setting the property `javax.portlet.name` to your portlet's name. 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. The list below highlights some of the methods 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 (e.g., `com_liferay_wiki_web_portlet_WikiPortlet`).

getTemplateVariableGroups(): Defines the variables exposed in the template editor.

As an example `*PortletDisplayTemplateHandler` implementation, you can look at the `WikiPortletDisplayTemplateHandler` class.

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 7.1.0//EN" "http://www.liferay.com/dtd/liferay-
resource-action-mapping_7_1_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 must ensure that Liferay DXP can find the updated `default.xml` with the new resource action when you deploy the module. Create a file named `portlet.properties` 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 should expose the ADT option to your users. 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 this:

```
<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 dropdown menu to be used in the widget'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 must obtain the value of those parameters and store them manually in your configuration class.

As an example JSP, see the Wiki widget's `configuration.jsp`.

5. You must now extend your view code to render your portlet with the selected ADT. This lets you decide which part of your view is rendered by the ADT and what is 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, grabs the entities to be rendered, and renders them using the ADT. The tag `<liferay-ddm:template-renderer>` aids with this process. It automatically uses the selected template, or renders 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 parameters 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`.

Awesome! Your portlet now supports ADTs! 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 Widgets with Application Display Templates section for more details on using ADTs.

140.2 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 → Template Engines → FreeMarker 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 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 Widgets 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 → Template Engines → FreeMarker Engine*.

The cool thing about ADTs is the power they provide to your portlets, providing infinite ways of editing your portlet to provide new interfaces for your portal users. Be sure to configure your FreeMarker templates appropriately for the most efficient customization process.

CHAPTER 141

Audience Targeting

Liferay's Audience Targeting application uses defined criteria to display content to users, who are organized into segments. You can target content to those user segments and create campaigns to expose user segments to a certain set of assets. Visit Targeting Content to your Audience for more information on Audience Targeting and how to use it.

Audience Targeting can be used to do all this without any customization, but it also contains a framework to be extended by developers.

There are a set of extension points you can use to customize its functionality, including

- 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.

Important: Not all Audience Targeting features function using the embedded HSQL database, so developers must use a Liferay-supported database to make full use of Audience Targeting features.

In these tutorials, you'll learn how to create these extension points for your Audience Targeting application.

141.1 Accessing the Content Targeting API

With the Content Targeting API you can integrate Audience Targeting features with third party applications or customize how Liferay's applications interact with Audience Targeting. For example, you could list user segments in your own application or update a campaign when someone creates a calendar event.

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 project's build file. Follow the instructions below to add the Content Targeting API dependency for a Gradle project.

1. Open the `build.gradle` file in your app's project folder.
2. Find the `dependencies { ... }` declaration and add the following line within that section:

```
compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.api", version: 5.0.0"
```

Your app now has access to the Content Targeting API and can take advantage of everything Audience Targeting has to offer. Next, you'll learn how to use the Content Targeting API by studying a few examples.

Using the Content Targeting Java API

You can call the Content Targeting API through Java or through JSON.

To display a list of existing user segments in your portlet using Java,

1. Obtain an implementation of the `UserSegmentLocalService` provided by Audience Targeting 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 the Audience Targeting application is installed and an implementation of the `UserSegmentLocalService` is available, the `_userSegmentLocalService` field is populated. Otherwise, the portlet is unavailable till this dependency is resolved.

2. 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 in 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.

3. Add this logic 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

You could do the same thing using the JSON API.

1. Open your portlet's `view.jsp` file and insert 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 want 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 to expose the Content Targeting API and use it in your application. Next you'll learn to create custom rule types.

CHAPTER 142

CREATING NEW AUDIENCE TARGETING RULE TYPES

In Audience Targeting, 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, or location. 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.

Audience Targeting ships with many rules for User Segments, but it's also extensible: if there isn't a rule that 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 want to promote goods 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 must do two things:

1. Retrieve the user's location so you can obtain that location's weather.
2. Let administrators set the value to compare 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.

There are four steps to create a custom rule type:

1. Create a module and ensure it has the necessary Content Targeting API dependencies.
2. Define how your rule works by implementing the `com.liferay.content.targeting.api.model.Rule` interface's methods.
3. Create the rule evaluation criteria.
4. Define the Rule's UI.

Now that you have an idea of how to plan your custom rule's development, you can create one!

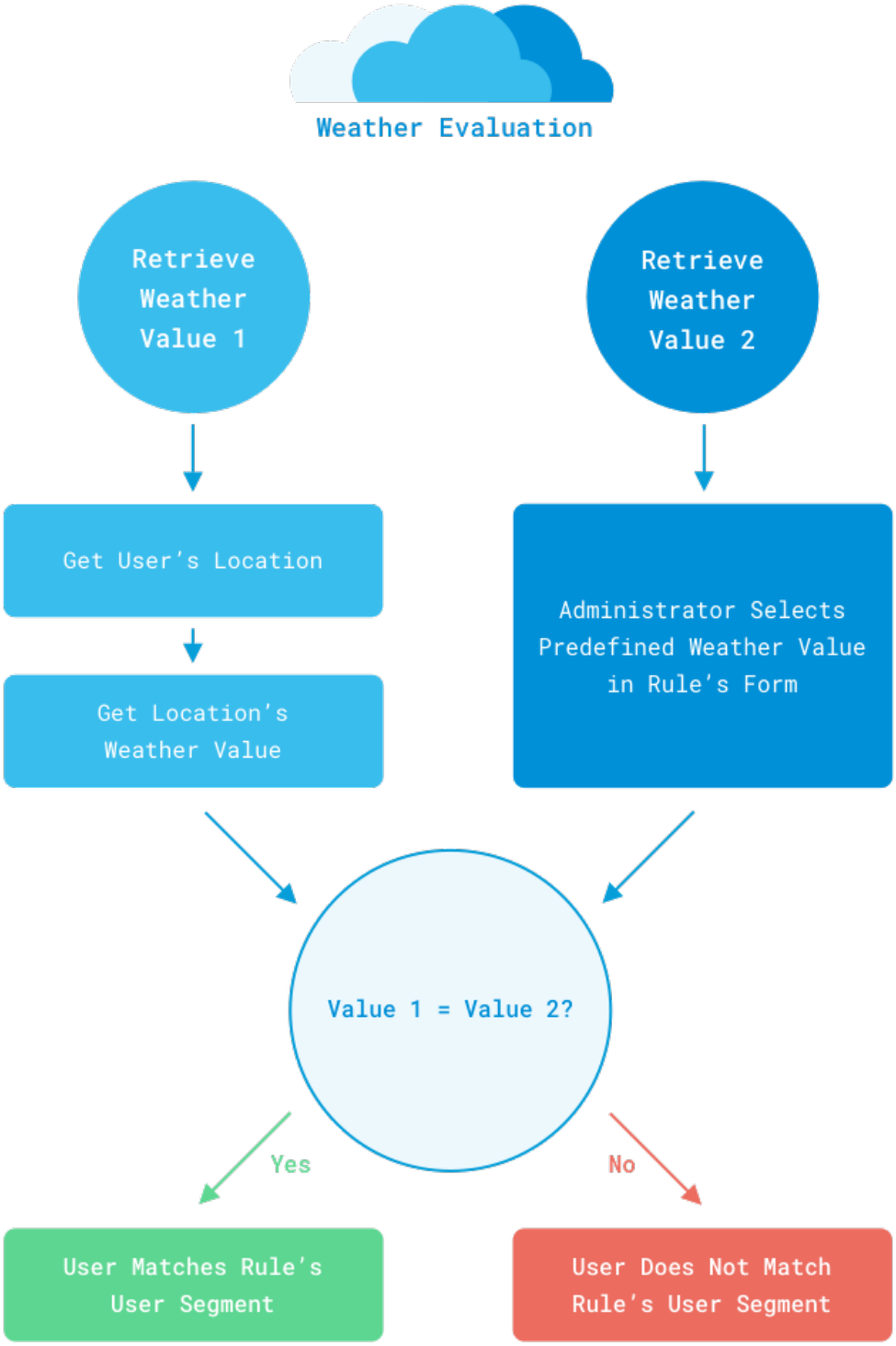


Figure 142.1: This diagram breaks down the evaluation process for the weather rule.

142.1 Creating a Custom Rule Type

First, you must create a module and ensure it has the necessary Content Targeting API dependencies.

Note: To view the Javadoc for the Content Targeting classes mentioned in this article, download the Javadoc JAR.

---

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. To use it, use this Blade command:

   ```
   blade create -t content-targeting-rule weather-rule
   ```

2. Make sure the dependencies are up to date, as sometimes the template gets out of sync with the latest release. Here are the dependency versions you should see in a Gradle based rule:

   ```
   dependencies {
       compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.analytics.api", version: "5.0.0"
       compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.anonymous.users.api", version: "3.0.0"
       compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.api", version: "5.0.0"
       compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.6.2"
       compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", version: "2.0.0"
       compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
       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 must define your rule's behavior. How your rule behaves is controlled by a Java class file that you create.

3. In the module's `src` directory appears a generated class. 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.

   You must 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 `com.liferay.content.targeting.api.model.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 Selecting a UI Technology.

4. Directly above the class's declaration should be the following annotation:

   ```
   @Component(immediate = true, service = Rule.class)
   ```

This annotation declares the implementation class of the Component, and specifies to start the module immediately once deployed to Liferay DXP.

Now that your Java class is set up, you must 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, 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.

Next you'll define the view/save lifecycle for the weather rule.

142.2 Defining a Rule's View/Save Lifecycle

The view/save lifecycle describes the process behind the scenes when an administrator applies a rule to a user segment using the User Segment Editor. You'll implement that now.

When the user opens the User Segment Editor, the render phase begins for the rule creation. During the render phase, the HTML for the form is generated and, if necessary, the `context` map is generated with any parameters that you need to create the form. Once the HTML is successfully retrieved and the user has set the values and clicked *Save*, the action phase begins.

When the action phase begins, the `processRule(...)` method takes the values provided by the form and persists them. 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.

In this section, you'll begin defining the weather rule's Java class. This assumes that you followed the instructions in the previous tutorial, creating the `WeatherRule` class and extending `com.liferay.content.targeting.api.model.BaseJSPRule`.

Note: To view the Javadoc for the Content Targeting classes mentioned in this article, download the Javadoc JAR.

If you used the `content-targeting-rule` Blade CLI template, your project already extends `BaseJSPRule` and has a default `view.jsp` file already created.

1. If you didn't use the template, 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 `com.liferay.content.targeting.api.model.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. Find the `getRuleCategoryKey()` method and replace it with the code below:

```
@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 `com.liferay.content.targeting.rule.categories.BehaviourRuleCategory`, `com.liferay.content.targeting.rule.categories.SessionAttributesRuleCategory`, `com.liferay.content.targeti` and `com.liferay.content.targeting.rule.categories.UserAttributesRoleCategory`.

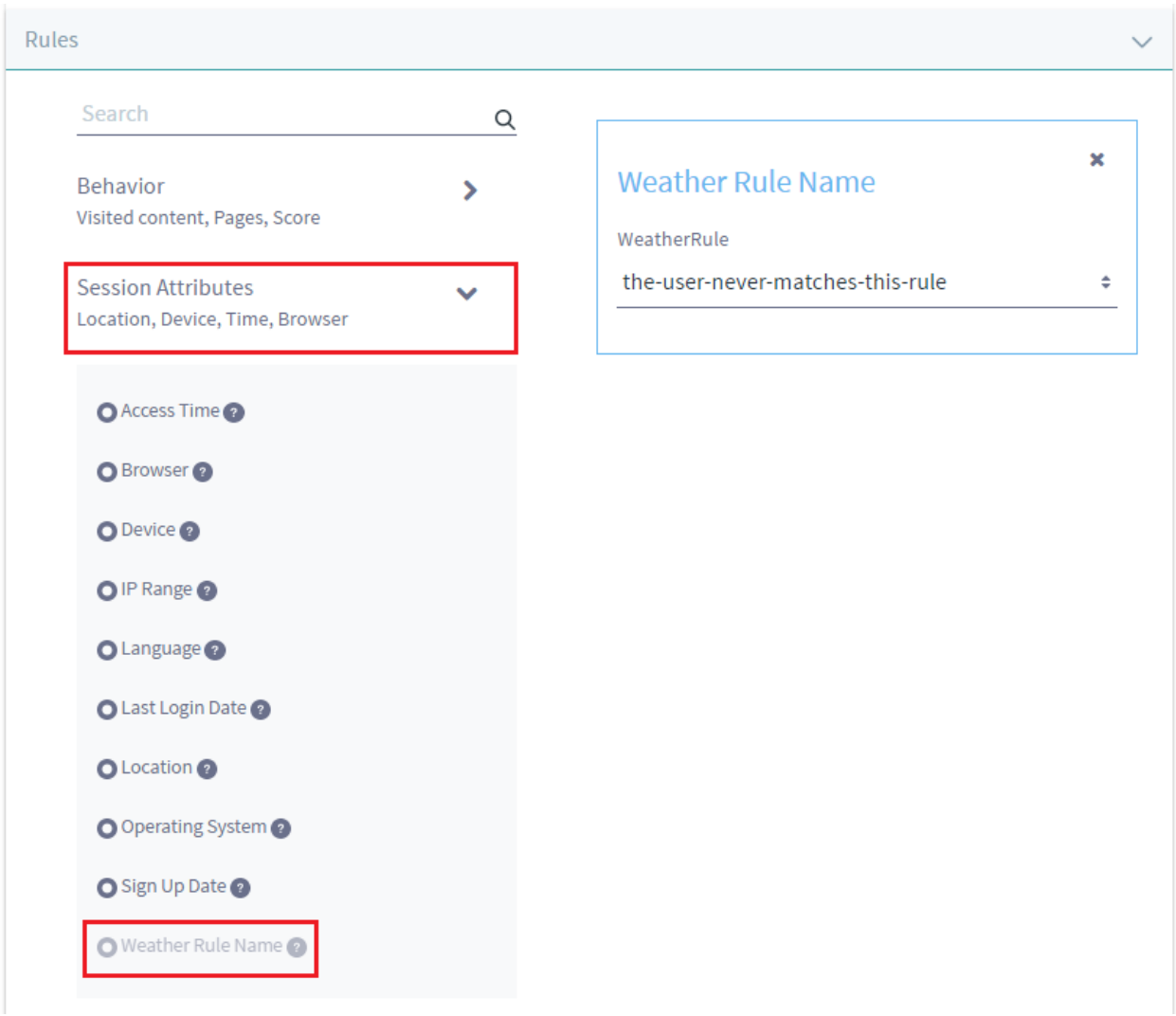


Figure 142.2: This example Weather rule was modified to reside in the Session Attributes category.

3. Find the `populateContext()` method and replace it with the code below:

```
@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);
}
```

To understand what this method accomplishes, you must examine the rule's configuration lifecycle.

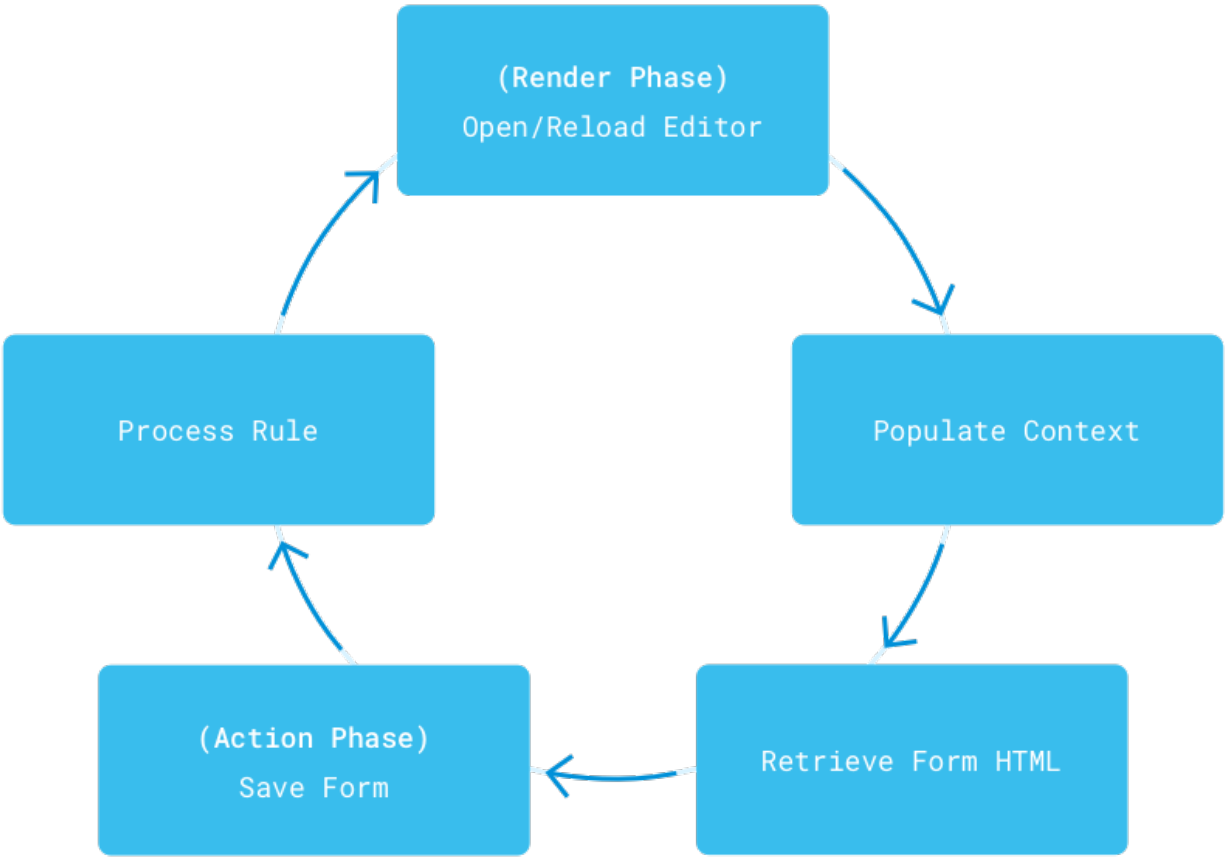


Figure 142.3: An Audience Targeting rule must be configured by the user and processed before it can become part of a User Segment.

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 `com.liferay.content.targeting.api.model.Rule` interface. This is because it's not needed in all cases. It's available by extending the `BaseJSPRule` class, and it needs more logic for the weather rule.

The `populateContext` method generates a map with all the parameters your JSP view needs to render the rule's HTML. This map is stored in the `context` variable. This variable is a map defining the form evaluation context for Audience Targeting rules. 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.

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 must obtain the values that the form should display from storage and inject them in the context so they appear 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 back-end code. Creating the weather rule's UI using a JSP is covered in Defining the Rule's UI. Once the HTML is successfully retrieved and the user has set the weather value and clicked *Save*, the action phase begins.

4. Replace the `processRule()` method with this code:

```
@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 `com.liferay.content.targeting.exception.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. You must add two more methods 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. Replace the generated `getSummary()` method with this one:

```
@Override
public String getSummary(RuleInstance ruleInstance, Locale locale) {
    return ruleInstance.getTypeSettings();
}
```

6. Set the servlet context for your rule. This method was generated and can be left alone:

```
@Override
@Reference(
    target = "(osgi.web.symbolicname=weather)",
    unbind = "-"
)
public void setServletContext(ServletContext servletContext) {
    super.setServletContext(servletContext);
}
```

Setting the servlet context 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.

142.3 Evaluating a Rule

After the administrator has successfully configured and saved your custom rule to his or her user segment, your rule needs to fulfill its purpose: 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.

142.4 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.

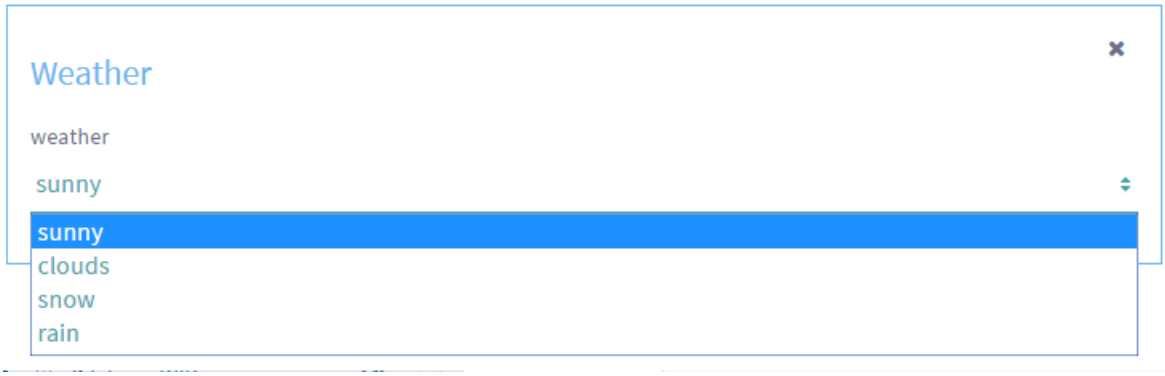


Figure 142.4: The weather rule uses a `select` drop-down box to set the weather value.

Note: 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.

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.

CHAPTER 143

TRACKING USER ACTIONS WITH AUDIENCE TARGETING

In Audience Targeting, a campaign defines a set of content targeted to specific user segments during a time period. 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 based on entity types that you can track, such as content, forms, links, pages, etc. 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.

Audience Targeting ships with many metrics, but it's also extensible. This means that if the default metrics available do not fulfill your needs, you can create one yourself.

First you need to define some criteria about your metric:

- Entity to Track
- Tracking Mechanism
- Tracking Events
- Differentiation Method

Creating a metric involves targeting what you want to track in a report. Suppose you're the owner of a hardware store and you want to send emails to your customers with 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 the newsletter.

To track how many customers view the newsletter, you must 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.

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. To do this you will,

1. Create a module with the necessary Content Targeting API dependencies.

2. Define the metric's View/Save lifecycle.
3. Implement a tracking mechanism and differentiation method.
4. Define the UI.

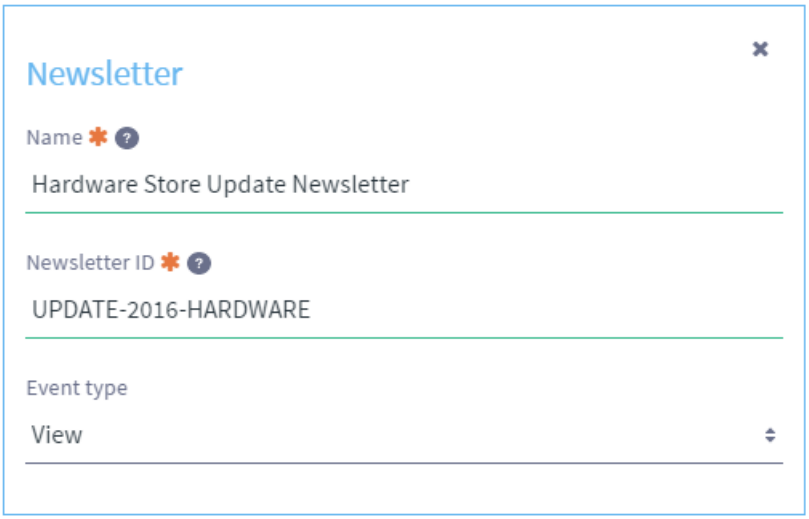


Figure 143.1: The sample Newsletter metric requires the newsletter name, ID, and event type.

Now that you have an idea of how to plan your new metric, you'll begin creating one next!

143.1 Related Topics

Creating Projects with Blade CLI
    Defining a Metric's View/Save Lifecycle
    Defining the Metric's UI

143.2 Creating a Metric

Now that all of your criteria has been defined, you can get started developing the actual metric:

Note: To view the Javadoc for the Content Targeting classes mentioned in this article, download the Javadoc JAR.

---

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: "5.0.0"
    compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.anonymous.users.api", version: "3.0.0"
    compileOnly group: "com.liferay.content-targeting", name: "com.liferay.content.targeting.api", version: "5.0.0"
    compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.6.2"
    compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", version: "2.0.0"
    compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
    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 `BaseJSPTrackingAction` 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 must define how your metric works by implementing the `TrackingAction` interface's methods. You'll begin implementing these methods next.

143.3 Defining a Metric's View/Save Lifecycle

In this section, you will define the metric's view/save lifecycle: what happens when a user applies a metric to a report using the Report Editor.

Note: To view the Javadoc for the Content Targeting classes mentioned in this article, download the Javadoc JAR.

You'll begin defining the newsletter metric's Java class. This assumes that you followed the instructions in the previous article to create the `NewsletterTrackingAction` class and extend `com.liferay.content.targeting.api.model.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 `com.liferay.content.targeting.api.model.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 look at the metric's configuration lifecycle.

   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 `populateContext` method generates 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

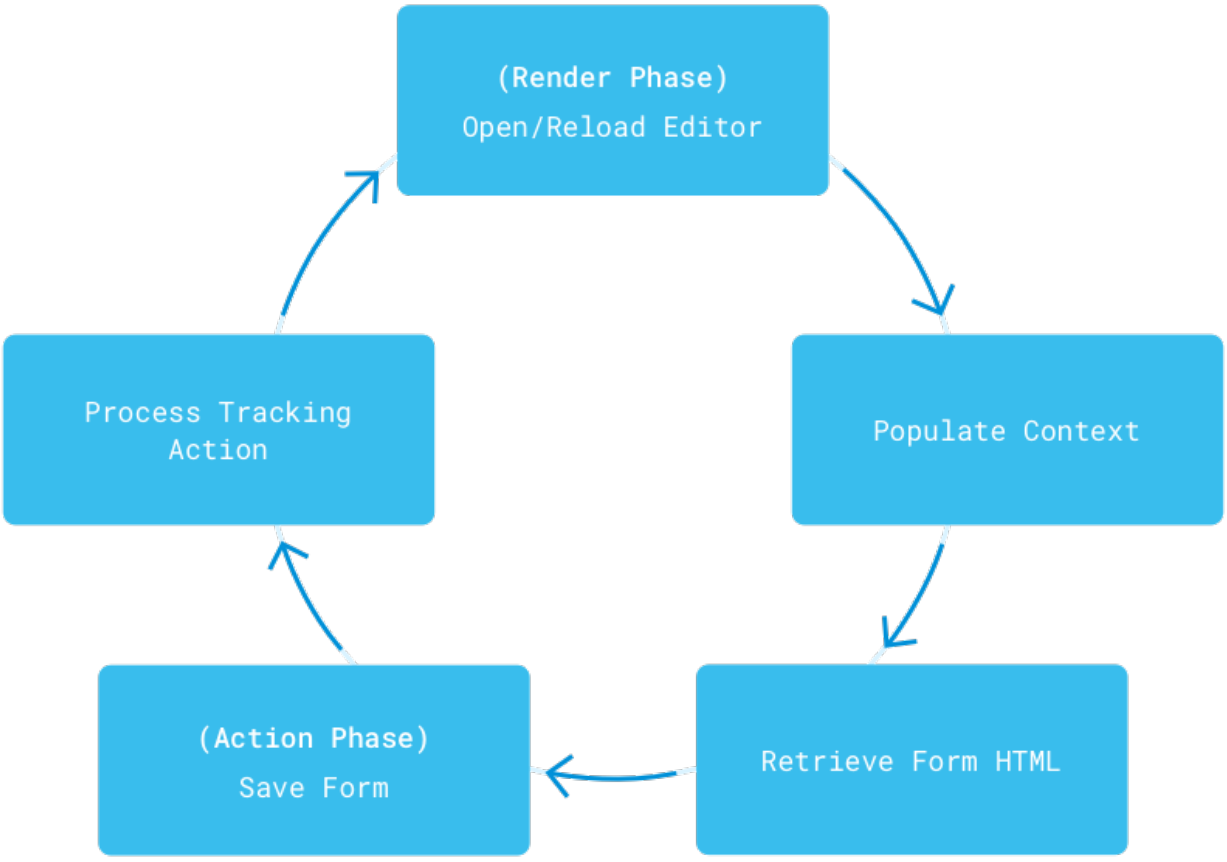


Figure 143.2: An Audience Targeting metric must be configured by the user and processed before it can become part of a Report.

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, Audience Targeting 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/7-1/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(
       TrackingActionInstance trackingActionInstance, Locale locale) {

       return LanguageUtil.get(
           locale, trackingActionInstance.getTypeSettings());
   }
   ```

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.

143.4 Using a Tracking Mechanism

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. 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 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.

143.5 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 + "eleme
    <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="<%= trackIma
</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.

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.

Figure 143.3: Once you've saved the metric, you can copy the generated transparent image URL into your newsletter's HTML to track who views it.

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.

143.6 Best Practices for Audience Targeting

Now that you've created a rule, here are some best practices to keep in mind when creating additional 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 To learn more about language keys and how to create, use, and generate them, visit the Internationalization tutorials.

Selecting a UI Technology

Audience Targeting gives you the option to choose whatever frontend technology you like, but JSP is the preferred technology for Audience Targeting extension views. FreeMarker views, however, are still supported through their respective base classes (e.g., `BaseFreemarkerRule` or `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 `-Rule` or `-TrackingAction` 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` or `-TrackingAction` 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.

CHAPTER 144

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 of tutorials shows how to customize these WYSIWYG editors for your apps and sites.

144.1 Adding a WYSIWYG Editor to a Portlet

It's easy to include WYSIWYG editors in your portlet, thanks to the `<liferay-editor:editor />` tag.

Note: The `<liferay-ui:input-editor />` tag is deprecated as of 7.0 in favor of the `<liferay-editor:editor />` tag. Use the `<liferay-editor:editor />` tag to avoid future issues.

Below is an example configuration:

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

<div class="alloy-editor-container">
    <liferay-editor: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 uri="http://liferay.com/tld/editor" prefix="liferay-editor" %>

<div class="alloy-editor-container">
    <liferay-editor: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-editor: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
    Modifying the AlloyEditor

144.2 Modifying an Editor's Configuration

You can use many different kinds of WYSIWYG editors to edit content in portlets. Depending on the content you're editing, you may want to modify the editor to provide a customized configuration for your needs. In this tutorial, you'll learn how to modify the default configuration for Liferay DXP's supported WYSIWYG editors to meet your requirements.

Updating the Editor's Configuration

To modify the editor's configuration, create a module with a component that implements the `EditorConfigContributor` interface. Follow these steps to modify one of Liferay DXP's WYSIWYG editors:

1. Create an OSGi module.
2. Open the portlet's `build.gradle` file and update the `com.liferay.portal.kernel` `version` to `3.6.2`. This is the version bundled with the Liferay DXP release.
3. Create a unique package name in the module's `src` directory, and create a new Java class in that package that extends the `BaseEditorConfigContributor` class:
4. Create a component class that implements the `EditorConfigContributor` service:

```
@Component(
    property = {

    },

    service = EditorConfigContributor.class
)
```

5. Add the following imports:

```
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;
```

6. Specify the editor's name, editor's configuration key, and/or the portlet name(s) where the editor resides. These three properties can be specified independently, or together, in any order. See the `EditorConfigContributor` interface's Javadoc for more information about the

available properties and how to use them. The example configuration below modifies the AlloyEditor's Content Editor, identified by the `contentEditor` configuration key and `alloyeditor` name 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 declared (Blogs and Blogs Admin), specifying that the
service applies to the content editors in those portlets. Lastly, the
configuration overrides the default one by providing a higher
[service ranking](/docs/7-1/tutorials/-/knowledge_base/t/fundamentals#services):

   @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
   )
```

````
**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
)
```

---

7. Override the `populateConfigJSONObject()` method to provide the custom configuration for the editor. This method updates the original configuration JSON object. It can also Update or delete existing configurations, or any other configuration introduced by another `*EditorConfigContributor`.

```
@Override
public void populateConfigJSONObject(
    JSONObject jsonObject, Map<String, Object> inputEditorTaglibAttributes,
    ThemeDisplay themeDisplay,
    RequestBackedPortletURLFactory requestBackedPortletURLFactory) {

}
```

8. In the `populateConfigJSONObject` method, you must instantiate a `JSONObject` to hold the current configuration of the editor. For instance, you could use the code snippet below to retrieve the available toolbars for the editor:

```
JSONObject toolbars = jsonObject.getJSONObject("toolbars");
```

```
**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 six. Use the
links the bottom of this tutorial to view each editor's configuration
options and requirements.
```

---

9. Now that you've retrieved the toolbar, you can modify it. The example below adds a camera button to the AlloyEditor's Add toolbar. It extracts the *Add* buttons out of the toolbar configuration object as a `JSONArray`, and then adds the button to that `JSONArray`:

```
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.

10. Finally, generate the module's JAR file and copy it to your `deploy` folder. Once the module is installed and activated in the 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
Modifying the AlloyEditor
Adding a WYSIWYG Editor to a Portlet

144.3 Adding New Behavior to an Editor

You can select from several different WYSIWYG editors for your users, and each is configurable and has its strengths and weaknesses. Configuration alone, however, doesn't always expose the features you want. In these cases, you can programmatically access the editor instance to create the editor experience you want, using the `liferay-util:dynamic-include` JavaScript extension point. It injects 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#additionalResources` or `alloyeditor#additionalResources` extension points to add the following code to the editor:

```
CKEDITOR.dtd.$removeEmpty.i = 0
```

In this tutorial, you'll learn how to use this JavaScript extension point.

Injecting JavaScript into a WYSIWYG Editor

The `liferay-util:dynamic-include` extension point is in configurable editors' JSP files: it's the gateway for injecting JavaScript into your editor instance:

1. Create a JS file containing your editor functionality in a folder that makes sense to reference, since you must register the file in your module. The extension point injects the JavaScript code right after editor initialization.

   Liferay injects JavaScript code for some applications:

   - creole_dialog_definition.js for the wiki

- creole_dialog_show.js also for the wiki
- dialog_definition.js for various applications

These JS files redefine the fields that show in 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.

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 annotation:

```
@Component(immediate = true, service = DynamicInclude.class)
```

This declares the component's implementation class and starts the module once deployed to Portal.

5. If you have not yet overridden the abstract methods from `DynamicInclude`, do that now. There are two implemented methods to edit: `include(...)` and `register(...)`.

6. In the `include(...)` method, retrieve the bundle containing your custom JS file. Retrieve the JS file as a URL and inject its contents into the editor. Here's the code 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 the 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. Liferay implemented this in 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 DXP supported WYSIWYG editor!

Related Topics

Adding New Behavior to an Editor
    Embedding Portlets in Themes
    Portlets

CHAPTER 145

ALLOYEDITOR

AlloyEditor is a modern WYSIWYG editor built on top of CKEDITOR, designed to create modern and gorgeous web content. AlloyEditor is the default WYSIWYG editor.

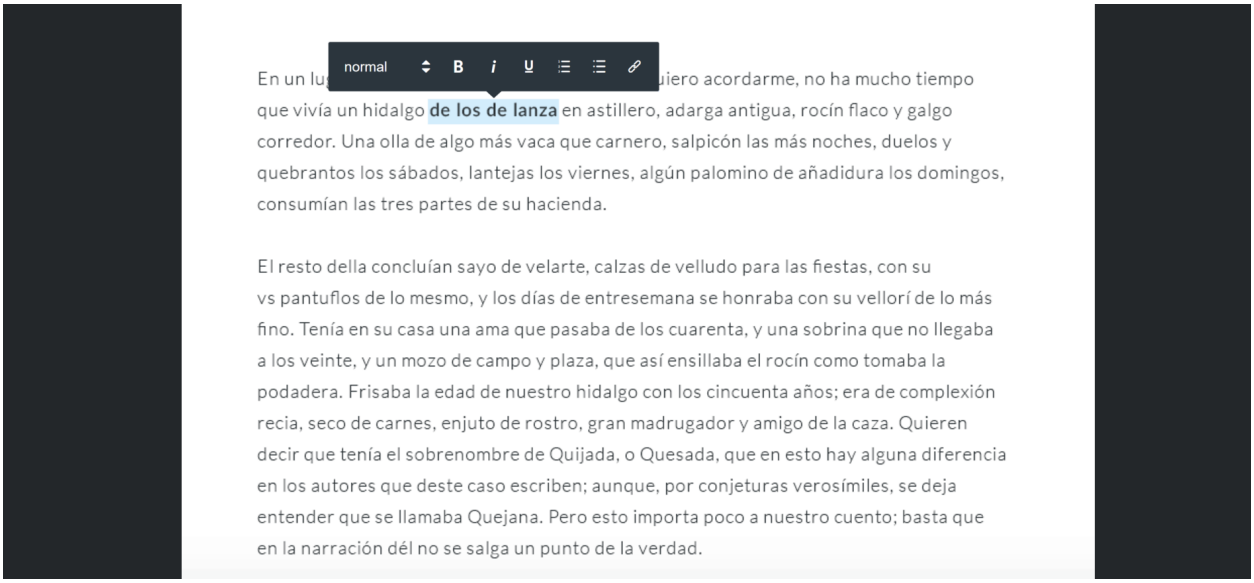


Figure 145.1: AlloyEditor is the default WYSIWYG editor built on top of CKEditor.

This section of tutorials shows how to modify the default AlloyEditor configuration to meet your requirements.

CHAPTER 146

ADDING BUTTONS TO ALLOYEDITOR'S TOOLBARS

AlloyEditor's toolbars contain several useful functions out-of-the-box. You may, however, want to customize the default configuration to include a button you've created, to add an existing button to a toolbar, or to add an existing CKEditor button that's bundled with Liferay DXP's AlloyEditor. The `EditorConfigContributor` interface, provides everything you need to modify an editor's configuration, including adding buttons to AlloyEditor's toolbars. CKEditor Configuration settings that modify the editor's behavior (excluding UI modifications) can also be passed down through this configuration object.

The `com.liferay.docs.my.button` module is the example throughout these tutorials. If you want to use it as a starting point for your own configuration or follow along with the tutorials, you can download the module's zip file from the Github repo.

146.1 Creating the OSGi Module and Configuring the EditorConfigContributor Class

To add a button to the AlloyEditor's toolbars, you must first create an OSGi component class of service type `EditorConfigContributor.class`. Follow these steps to create and configure the OSGi module:

1. Create an OSGi module, using Blade's portlet template:

```
blade create -t portlet -p com.liferay.docs.my.button -c
MyEditorConfigContributor my-new-button
```

2. Open the portlet's `build.gradle` file and update the `com.liferay.portal.kernel version` to `3.6.2`. This is the version bundled with the Liferay DXP release.

3. Open the portlet class you created in step one (`MyEditorConfigContributor`) and add the following imports:

```
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;
```

4. Replace the `@Component` and properties with the properties below:

```
@Component(
    immediate = true,
    property = {
      "editor.name=alloyeditor",
      "service.ranking:Integer=100"
    },
    service = EditorConfigContributor.class
)
```

   This targets AlloyEditor for the configuration and overrides the default service by providing a higher service ranking. If you want to target a more specific configuration, you can find the available properties in the `EditorConfigContributor` interface's Javadoc.

5. Extend `BaseEditorConfigContributor` instead of `GenericPortlet`.

6. Replace the `doView()` method and contents with the `populateConfigJSONObject()` method shown below:

```
@Override
public void populateConfigJSONObject(
        JSONObject jsonObject, Map<String, Object> inputEditorTaglibAttributes,
        ThemeDisplay themeDisplay,
        RequestBackedPortletURLFactory requestBackedPortletURLFactory) {

}
```

7. Inside the `populateConfigJSONObject()` method, retrieve the AlloyEditor's toolbars:

```
JSONObject toolbarsJSONObject = jsonObject.getJSONObject("toolbars");

if (toolbarsJSONObject == null) {
        toolbarsJSONObject = JSONFactoryUtil.createJSONObject();
}
```

8. If you're adding a button for one of the CKEditor plugins bundled with the AlloyEditor, add the code below to retrieve the extra plugins and add the plugin to the AlloyEditor's configuration. The example below adds the `clipboard` CKEditor plugin:

```
String extraPlugins = jsonObject.getString("extraPlugins");

if (Validator.isNotNull(extraPlugins)) {
  extraPlugins = extraPlugins + ",ae_uibridge,ae_autolink,
  ae_buttonbridge,ae_menubridge,ae_panelmenubuttonbridge,ae_placeholder,
  ae_richcombobridge,clipboard";
}
else {
  extraPlugins = "ae_uibridge,ae_autolink,ae_buttonbridge,ae_menubridge,
  ae_panelmenubuttonbridge,ae_placeholder,ae_richcombobridge,clipboard";
}

jsonObject.put("extraPlugins", extraPlugins);
```

   AlloyEditor also comes with several plugins to bridge the gap between the CKEditor's UI and the AlloyEditor's UI. These are prefixed with the `ae_` you see above. We recommend that you include them all to ensure compatibility.

The `*EditorConfigContributor` class is prepared. Now you must choose which toolbar you want to add the button(s) to: the Add Toolbar or one of the Styles Toolbars.

Related Topics

Adding New Behavior to an Editor
    CKEditor Plugin Reference Guide

146.2 Adding a Button to the Add Toolbar

The Add Toolbar appears in the AlloyEditor when your cursor is in the editor and you click the Add button:

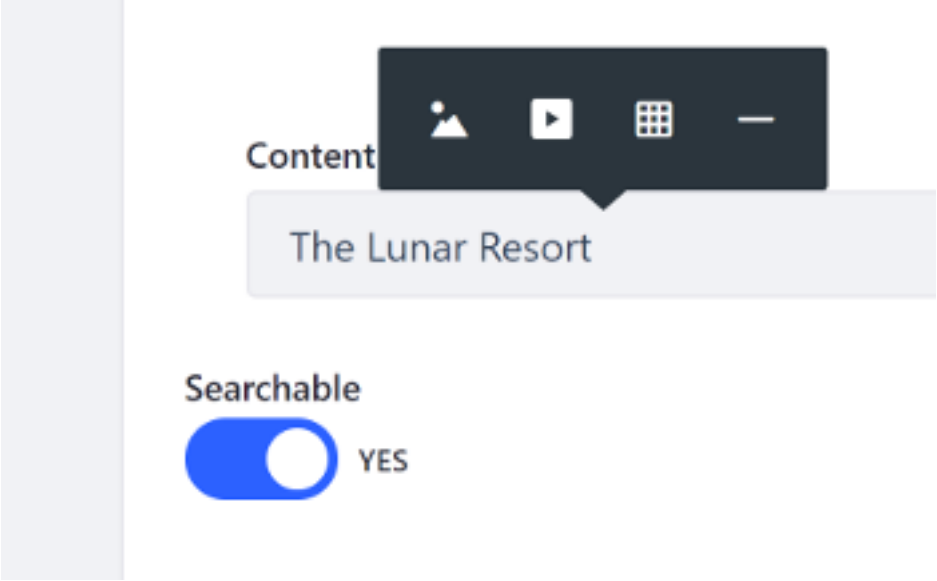


Figure 146.1: The Add toolbar lets you add content to the editor.

Follow these steps to add a button to the AlloyEditor's Add Toolbar:

1. Inside the `populateConfigJSONObject()` method, retrieve the Add Toolbar:

```
JSONObject addToolbar = toolbarsJSONObject.getJSONObject("add");
```

2. Retrieve the existing Add Toolbar buttons:

```
JSONArray addToolbarButtons = addToolbar.getJSONArray("buttons");
```

3. Add the button to the existing buttons. Note that the button's name is case sensitive. The example below adds the `camera` button to the Add Toolbar:

```
addToolbarButtons.put("camera");
```

   The camera button is just one of the buttons available by default with AlloyEditor, but they are not all enabled. Here's the full list of available buttons you can add to the Add Toolbar:

   - camera

- embed
- hline
- image
- table

See here for an explanation of each button's features.

4. Update the AlloyEditor's configuration with the changes you made:

```
addToolbar.put("buttons", addToolbarButtons);

toolbarsJSONObject.put("add", addToolbar);

jsonObject.put("toolbars", toolbarsJSONObject);
```

5. Deploy your module and create new content that uses the AlloyEditor—like a blog entry or web content article—to see your new configuration in action!

The `com.liferay.docs.my.button` module's updated Add Toolbar is shown in the figure below:

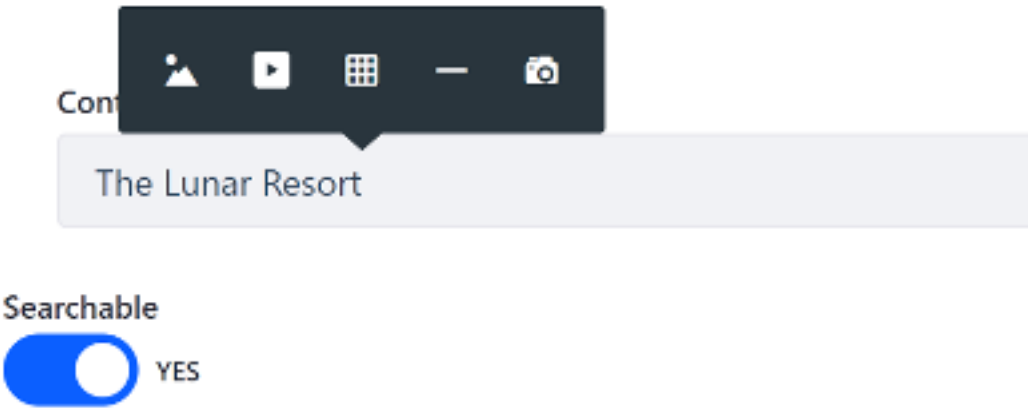


Figure 146.2: The Updated Add toolbar lets you add pictures from a camera directly to the editor.

Related Topics

Adding New Behavior to an Editor
    Adding a Button to a Styles Toolbar

146.3 Adding a Button to a Styles Toolbar

A Styles Toolbar appears when content is selected or highlighted in AlloyEditor. There are five Styles toolbars to choose from:

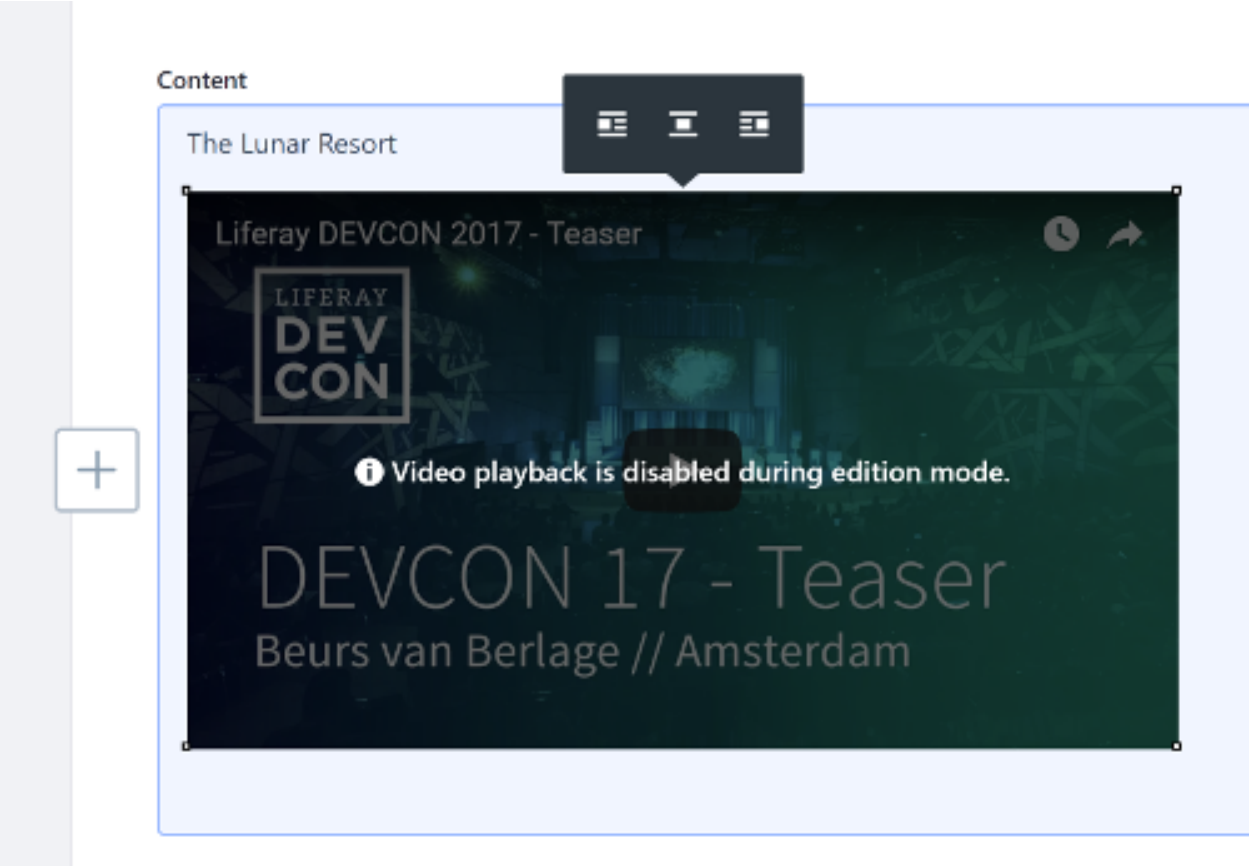


Figure 146.3: The embed URL Styles toolbar lets you format embedded content in the editor.

`embedurl`: Appears when embedded content is selected.
`image`: Appears when an image is selected.
`link`: Appears when a hyperlink is selected.
`table`: Appears when a table is selected.
`text`: Appears when text is highlighted.
Follow these steps to add a button to one of the Styles toolbars:

1. Inside the `populateConfigJSONObject()` method, retrieve the Styles toolbar:

```
JSONObject stylesToolbar = toolbarsJSONObject.getJSONObject("styles");

if (stylesToolbar == null) {
	stylesToolbar = JSONFactoryUtil.createJSONObject();
}
```

2. Retrieve the available selection toolbars:

```
JSONArray selectionsJSONArray = stylesToolbar.getJSONArray(
"selections");
```

3. Iterate through the selection toolbars, select the one you want to add the button(s) to (`embedurl`, `image`, `link`, `table`, or `text`), retrieve the existing buttons, and add your button. The example

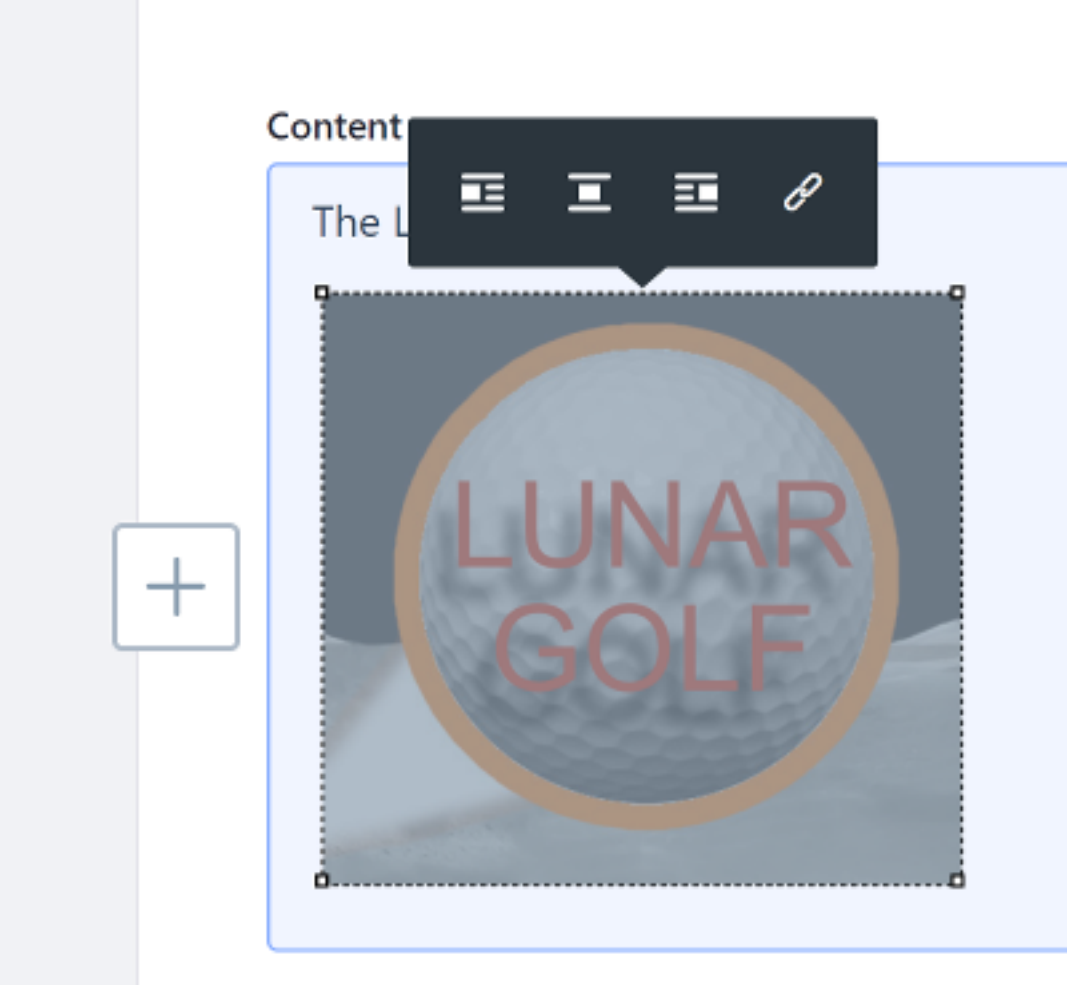


Figure 146.4: The image Styles toolbar lets you format images in the editor.

below adds the `clipboard` plugin's `Copy`, `Cut`, and `Paste` buttons to the `text` selection toolbar. Note that buttons are case sensitive and may be aliased or not match the name of the plugin. Search the plugin's `plugin.js` file for `editor.ui.addButton` to find the button's name:

```
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("Copy");
                buttons.put("Cut");
                buttons.put("Paste");
        }
}
```

The example above adds one of the CKEditor plugins bundled with Liferay DXP's AlloyEditor. There are also several buttons available by default with the AlloyEditor, but they are not all enabled. The full list of existing buttons you can add to the Styles toolbars is shown in the table below, ordered by Toolbar:

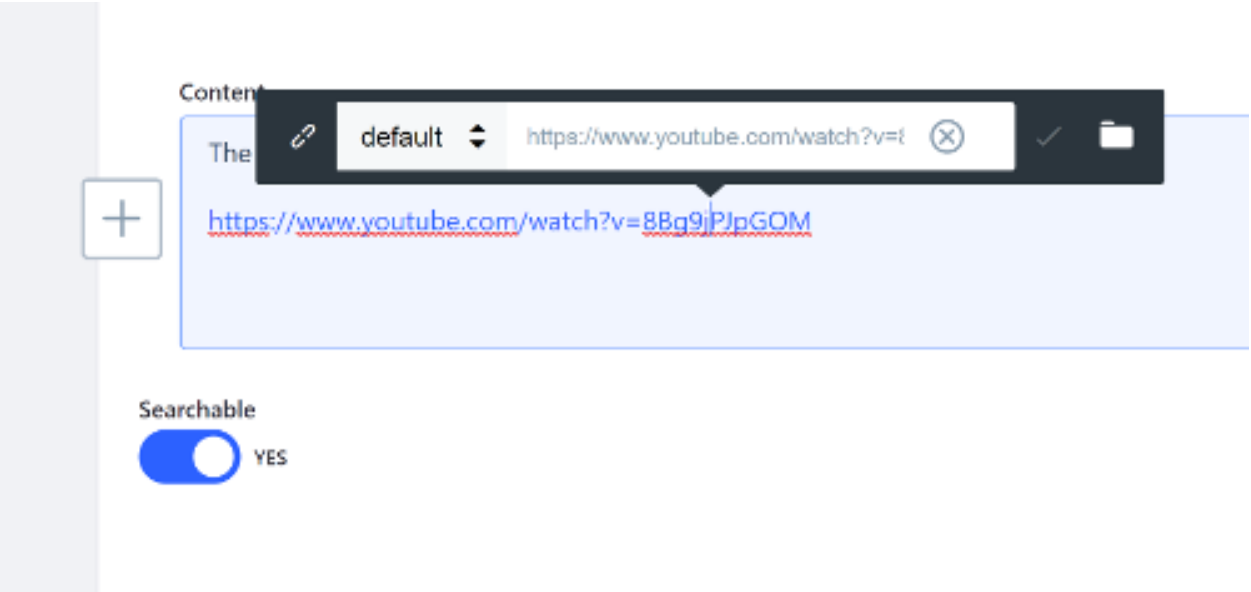


Figure 146.5: The link Styles toolbar lets you format hyperlinks in the editor.

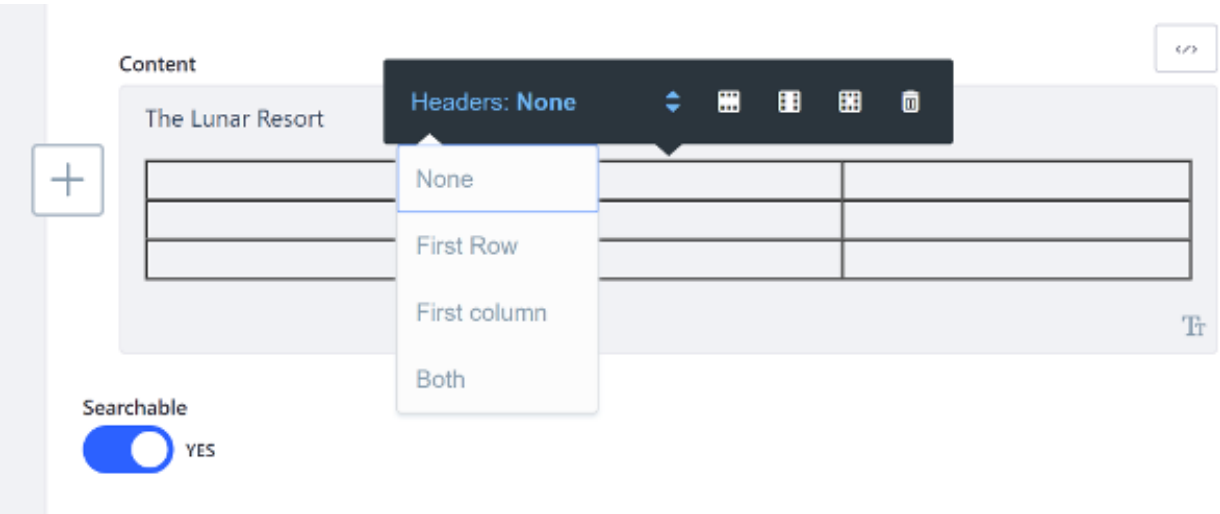


Figure 146.6: The table Styles toolbar lets you format tables in the editor.

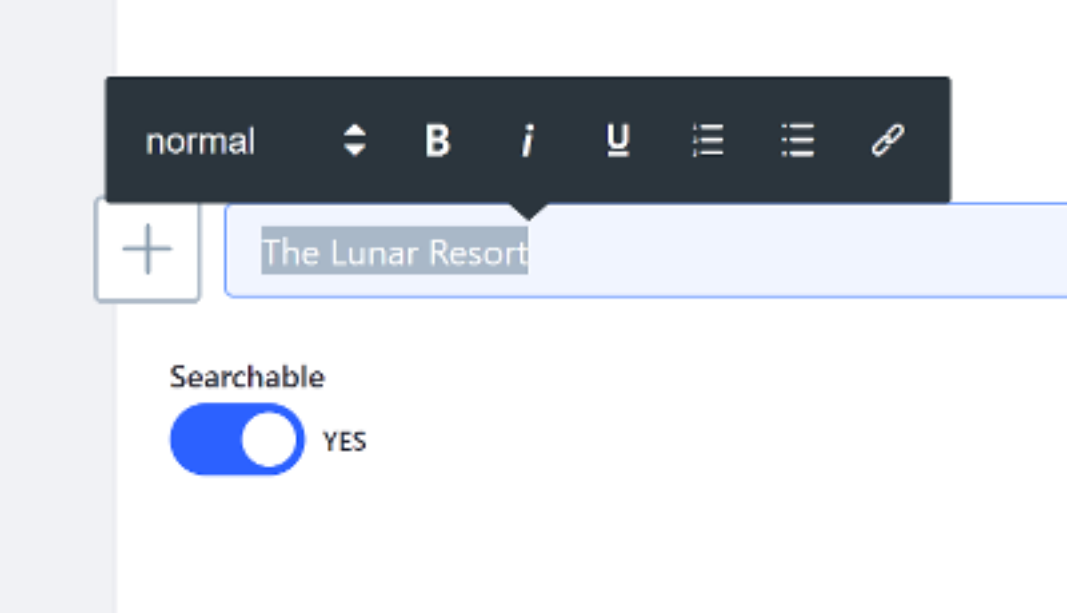


Figure 146.7: The text Styles toolbar lets you format highlighted text in the editor.

```
text | table | image | link |
---- | ----- | ----- | ---- |
bold | tableHeading | imageCenter | linkEdit |
code | tableRow | imageLeft | |
h1 | tableColumn | imageRight | |
h2 | tableCell | | |
indentBlock | tableRemove | | |
italic | | | |
link | | | |
ol | | | |
outdentBlock | | | |
paragraphLeft | | | |
paragraphRight | | | |
paragraphCenter | | | |
paragraphJustify | | | |
quote | | | |
removeFormat | | | |
strike | | | |
styles | | | |
subscript | | | |
superscript | | | |
twitter | | | |
ul | | | |
underline | | | |
```

```
See
[here](https://alloyeditor.com/docs/features/)
for an explanation of each button's features.
```

4. Update the AlloyEditor's configuration with the changes you made:

```
stylesToolbar.put("selections", selectionsJSONArray);
```

```
toolbarsJSONObject.put("styles", stylesToolbar);

jsonObject.put("toolbars", toolbarsJSONObject);
```

5. Deploy your module and create a new piece of content that uses the AlloyEditor—such as a blog entry or web content article—to see your new configuration in action!

The `com.liferay.docs.my.button` module's updated text styles toolbar is shown in the figure below:

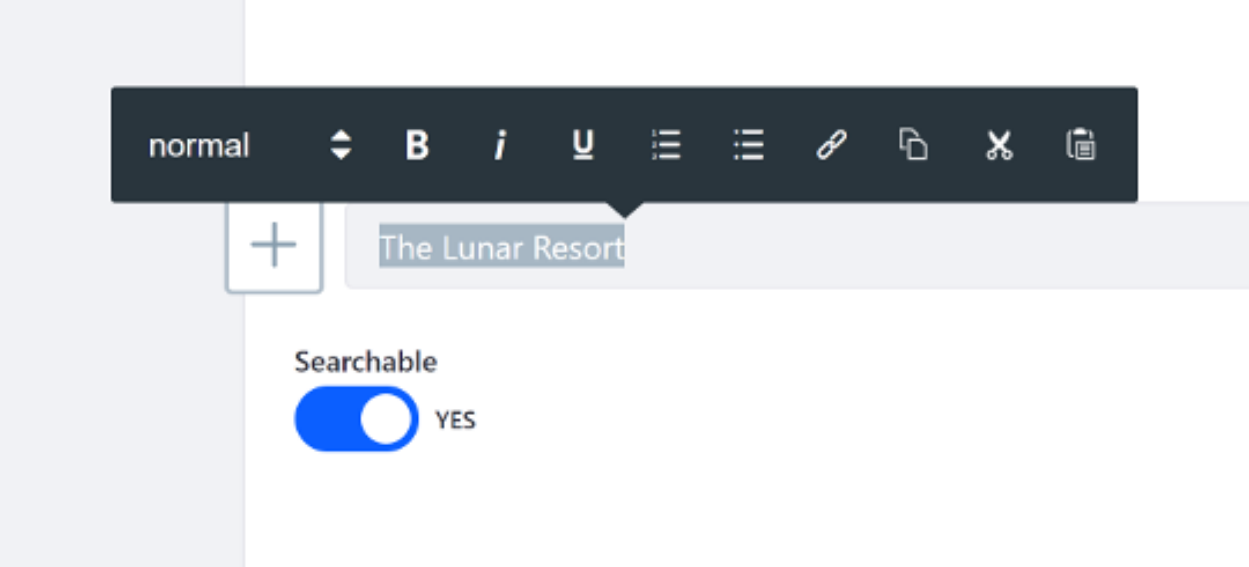


Figure 146.8: The Updated text styles toolbar lets you copy, cut, and paste text in the editor.

Related Topics

Adding a Button to the Add Toolbar
CKEditor Plugin Reference Guide

CHAPTER 147

CREATING NEW BUTTONS FOR ALLOYEDITOR

AlloyEditor is built on `React.js` and uses `jsx` to render each button in the editor. To add a new button to the AlloyEditor, you must create an OSGi bundle that contains three key pieces:

- A JSX file containing the button's configuration
- A Java class that contributes the button to the list of available buttons
- A Java class that adds the button to the AlloyEditor's toolbar

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`
    - `bnd.bnd`

- `build.gradle`
- `package.json`

The tutorials in this section cover the following topics:

- How to create your button's OSGi bundle
- How to create your button's JSX file
- How to contribute your button to the list of available buttons

You can learn how to add your button to the editor's toolbars in the Adding Buttons to AlloyEditor's Toolbars tutorials.

The `my-log-text-button` bundle is used as an example throughout this tutorial. You can download the bundle's zip file for reference, or use it as a starting point for your project if you wish.

147.1 Creating the AlloyEditor Button's OSGi Bundle

Follow these steps to create your OSGi bundle for your new button:

1. Create an OSGi module
2. Add a `resources\META-INF\resources\js` folder to your module's `src\main` folder.
3. Specify your bundle's `Web-ContextPath` in its `bnd.bnd` file. An example BND file configuration is shown below with the `Web-ContextPath` pointing to the bundle's root folder. This is required to properly locate and load the module's JavaScript:

```
Bundle-Name: my-log-text-button
Bundle-SymbolicName: com.liferay.docs.portlet
Bundle-Version: 1.0.0
Web-ContextPath: /my-button-portlet-project
```

4. Since the button's configuration is defined in a JSX file, it must be transpiled for the browser. You can do this by adding the `transpileJS` task to your `build.gradle` file. An example configuration is shown below:

```
configJSModules {
    enabled = false
}

dependencies {
  compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.6.2"
  compileOnly group: "com.liferay.portal", name: "com.liferay.util.taglib", version: "2.0.0"
  compileOnly group: "javax.portlet", name: "portlet-api", version: "3.0.0"
  compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
  compileOnly group: "jstl", name: "jstl", version: "1.2"
  compileOnly group: "org.osgi", name: "osgi.cmpn", version: "6.0.0"
  compileOnly group: "org.osgi", name: "org.osgi.core", version: "6.0.0"
}

transpileJS {
    bundleFileName = "js/buttons.js"
    globalName = "AlloyEditor.Buttons"
    modules = "globals"
    srcIncludes = "**/*.jsx"
}
```

5. Add the following `devDependencies` to your `package.json` file:

```
{
  "devDependencies": {
    "babel-preset-react": "^6.11.1",
    "metal-cli": "^4.0.1"
  },
  "name": "my-bundle-name",
  "version": "1.0.0"
}
```

6. Add the following preset to your module's `.babelrc` file to transpile your JSX file:

```
{
  "presets": [
    "react"
  ]
}
```

Related Topics

Adding New Behavior to an Editor
    Creating the Button's JSX File
    Contributing the Button to AlloyEditor

147.2 Creating the Button's JSX File

Follow these steps to create your button for AlloyEditor:

1. Create a `.jsx` file in your OSGi bundle's `resources\META-INF\resources\js` folder. This file defines your button's configuration.

2. Inside the JSX file, define the React variables your buttons require (React, ReactDOM). The log text button only uses AlloyEditor's React:

```
(function() {
        'use strict';

        var React = AlloyEditor.React;
```

3. Create your button's class:

```
var LogSelectedTextButton = React.createClass(
    {
        //button configuration goes here
    }
);
```

4. Inside the `React.createClass()` method's configuration object, specify the mixins your button requires. These provide additional functionality, making it easy to add features to your button, such as binding a shortcut key to your button. The example below uses the `ButtonStateClasses` and `ButtonKeystroke` mixins:

```
mixins: [AlloyEditor.ButtonStateClasses, AlloyEditor.ButtonKeystroke],
```

5. Pass validating props for your button. These are defined for each instance of the button. At the very least, the `editor` must be defined. The example below sets up properties for the editor, label, and tabIndex:

```
propTypes: {
    /**
     * The editor instance where the component is being used.
     *
     * @instance
     * @memberof LogSelectedTextButton
     * @property {Object} editor
     */
    editor: React.PropTypes.object.isRequired,

    /**
     * The label that should be used for accessibility purposes.
     *
     * @instance
     * @memberof LogSelectedTextButton
     * @property {String} label
     */
    label: React.PropTypes.string,

    /**
     * The tabIndex of the button in its toolbar current state. A value other than -1
     * means that the button has focus and is the active element.
     *
     * @instance
     * @memberof LogSelectedTextButton
     * @property {Number} tabIndex
     */
    tabIndex: React.PropTypes.number
},
```

6. Define the static properties for your button. You must at least provide the `key`. The `key` defines the button's name to specify in the AlloyEditor's configuration. The `my-log-text-button` module's static properties are shown below:

```
statics: {
    /**
     * The name which is used as an alias of the button in the configuration.
     *
     * @default myTestButton
     * @memberof LogSelectedTextButton
     * @property {String} key
     * @static
     */
    key: 'logSelectedText'
},
```

7. Optionally define any default properties your button has for each instance using the `getDefaultProps` property. The example below uses the `ButtonKeystroke` mixin's required `command` and `keystroke` properties to set the shortcut keys for the button's `logText()` function:

```
getDefaultProps: function() {
    return {
        command: 'logText',
        keystroke: {
```

```
            fn: 'logText',
            keys: CKEDITOR.CTRL + CKEDITOR.SHIFT + 89 /*Y*/
        }
    };
},
```

8. Define the HTML markup to render for your button. The example below uses the `getStateClasses()` method to retrieve the state class information provided by the `ButtonStateClasses` mixin and add it to the current `cssClass` value. It also uses Liferay Util's `getLexiconIconTpl()` method to retrieve a Lexicon icon to use for the button. See Lexicon's Design Site for a full list of the available icons.

```
render: function() {
        var cssClass = 'ae-button ' + this.getStateClasses();
        var svg = Liferay.Util.getLexiconIconTpl('desktop');

        return (
                <button
                    className={cssClass}
                    onClick={this._logText}
                    title="Log the selected text in the console"
                    dangerouslySetInnerHTML={{__html: svg}}
                />
        );
},
```

9. Define your button's main action. Retrieving the `nativeEditor`, as shown in the example below, gives you access to the full API of CKEditor. From there, you can use any of the available `CKEditor.editor` methods to interact with the editor's content. The example below chains the editor's `getSelection()` and `getSelectedText()` methods to retrieve the user's highlighted text, and then it logs it to the browser's console:

```
/**
 * @protected
 * @method  _logText
 */
_logText: function() {
    var editor = this.props.editor.get('nativeEditor');
    var selectedText = editor.getSelection().getSelectedText();

    console.log("Your selected text is " + selectedText);
}
```

10. Finally, add the button to the list of available buttons:

    AlloyEditor.Buttons[LogSelectedTextButton.key] = AlloyEditor.LogSelectedTextButton = LogSelectedTextButton;

Now you know how to create a button for AlloyEditor!

Related Topics

Adding New Behavior to an Editor
- Creating the AlloyEditor Button's OSGi Bundle
- Contributing the Button to AlloyEditor

147.3 Contributing the Button to AlloyEditor

Once you've created your button, you can add it to the list of 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:

1. Create a Component class that implements the `DynamicInclude.class` service and extends `BaseDynamicInclude`:

```
@Component(immediate = true, service = DynamicInclude.class)
public class MyButtonDynamicInclude extends BaseDynamicInclude {
```

2. Override the `include()` method to include a script with your transpiled JSX file. You can use the `StringBundler` to concatenate the script. Note the `sb.append("/js/buttons.js")` line below. This is the `bundleFileName` you defined in your bundle's `build.gradle` `transpileJS` task:

```
@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());
}
```

3. Override the `register()` method to use the `additionalResources` dynamic include to add your script. Note the `@Reference` annotation's `target` value is your bundle's symbolic name defined in its `bnd.bnd` file:

```
    @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 follow the steps covered in Adding Buttons to the AlloyEditor's Toolbars tutorials to add the button to the editor's toolbars.

Related Topics

Adding New Behavior to an Editor
    Creating the Button's JSX File
    WYSIWYG Editor Dynamic Includes

147.4 Embedding Content in the AlloyEditor

Whether it's a video from a popular streaming service, or an entertaining podcast, embedded content is commonplace on the web. Sharing content from a third party is sometimes required to properly cover a topic. The `EmbedProvider` mechanism lets you embed third party content in the AlloyEditor, while writing blog posts, web content articles, etc. By default, the `EmbedProvider` mechanism is only configured for embedding video content (Facebook, Twitch, Vimeo, and YouTube) into the AlloyEditor. This tutorial shows how to include additional video providers, and even add support for additional content types.

An `EmbedProvider` requires four pieces of information:

- An ID: The content's ID
- A Template: The required embed code for the provider
- A URL Schemes: URL patterns that are supported for the provider template
- A Type (optional): The provider category

When you add a supported URL to the editor, the `EmbedProvider` transforms the URL into the embed code.

Follow these steps to create an *`EmbedProvider`:

1. Create a module for the Embed Provider.

2. Add the following dependencies to the `build.gradle` file:

```
compileOnly group: "com.liferay", name:
"com.liferay.frontend.editor.api", version: "1.0.1"

compileOnly group: "com.liferay", name: "com.liferay.petra.string",
version: "2.0.0"
```

3. Create a component class that implements the `EditorEmbedProvider` service:

```
@Component(
  immediate = true,
  service = EditorEmbedProvider.class
)
```

4. Optionally set the type property to the content's type. If creating a provider for a content type other than video, you can create a new type constant and add a new button for the content type. If you do create your own button, we recommend that you use the existing embed video button's JSX files as an example to write your own files. By default, the provider is categorized as UNKNOWN. The example configuration below specifies the VIDEO type, using a constant provided by the EditorEmbedProviderTypeConstants class:

```
@Component(
  immediate = true,
  property = "type=" + EditorEmbedProviderTypeConstants.VIDEO,
  service = EditorEmbedProvider.class
)
```

5. Implement the EditorEmbedProvider interface. An example configuration is shown below:

```
public class MyEditorEmbedProvider implements EditorEmbedProvider {

}
```

6. Add the required imports:

```
import com.liferay.frontend.editor.api.embed.EditorEmbedProvider;
import com.liferay.frontend.editor.api.embed.EditorEmbedProviderTypeConstants;
import com.liferay.petra.string.StringBundler;
```

   Note the *TypeConstants import is only needed if you're adding a Video type provider.

7. Override the *EmbedProvider's getId() method to return the ID for the provider. An example configuration is shown below:

```
@Override
public String getId() {
    return "providerName";
}
```

8. Override the *EmbedProvider's getTpl() method to provide the embed template code (usually an iframe for the provider). The example below defines the template for a streaming video service. Note that {embedId} is a placeholder for the unique identifier for the embedded content:

```
@Override
public String getTpl() {
    return StringBundler.concat(
        "<iframe allow=\"autoplay; encrypted-media\" allowfullscreen ",
        "height=\"315\" frameborder=\"0\" ",
        "src=\"https://www.liferaylunarresortstreaming.com/embed/{embedId}?rel=0\" ",
        "width=\"560\"></iframe>");
}
```

9. Override the *EmbedProvider's getURLSchemes() method to return an array of supported URL schemes that have an embedded representation for the provider. URL schemes are defined using a JavaScript regular expression that indicates whether a URL matches the provider. Every URL scheme should contain a single matching group. Matches replace the {embedId} placeholder defined in the previous step:

```
@Override
public String[] getURLSchemes() {
    return new String[] {
        "https?:\\/\\/(?:www\\.)?liferaylunarresortstreaming.com\\/watch\\?v=(\\S*)$"
    };
}
```

10. Deploy your module and open an app that uses the AlloyEditor, such as Blogs, and create a new entry. Click the *add button* and select the video button—or your new content type button—and paste the content's URL. Click the *checkmark* to confirm that the URL scheme is supported. The content is embedded into the editor.

Now you know how to embed content in the AlloyEditor. Create a new content entry, such as a blog post, and click the embed video button—or the one you created—and paste the content's URL.

Related Topics

Adding Buttons to AlloyEditor's Toolbars

Adding New Behavior to an Editor

CHAPTER 148

SERVLETS

Java Servlets are foundational to Java EE. You can use servlets and servlet filters to provide applications in your portal context and to process requests and responses to specific URLs in your sites. The tutorials here cover servlet technology and how it integrates with Liferay DXP.

148.1 Servlets in a Module

You can use servlets or JAX-RS to provide a lightweight web integration or a web endpoint to a browser client. Servlets, rather than REST endpoints or portlets, let you control an application's entire UI experience. Liferay DXP supports servlet based applications and embeds HTTP Whiteboard for servlets.

Here you'll examine a servlet sample and create your own servlet based application.

Servlet Sample

The servlet sample uses HTTP Whiteboard to respond to requests at URLs that match the pattern `http://localhost:8080/o/blade/servlet/*`.

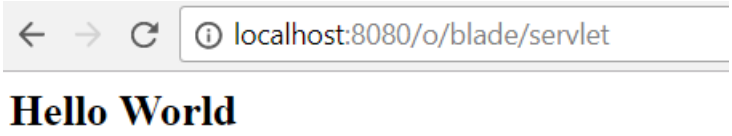


Figure 148.1: If users visit `http://localhost:8080/o/blade/servlet`, the servlet sample shows the message `Hello World`.

Here's the sample servlet class:

```
package com.liferay.blade.samples.servlet;

import java.io.IOException;

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.log.LogService;

/**
 * @author Liferay
 */
@Component(
    immediate = true,
    property = {
        "osgi.http.whiteboard.context.path=/",
        "osgi.http.whiteboard.servlet.pattern=/blade/servlet/*"
    },
    service = Servlet.class
)
public class BladeServlet extends HttpServlet {

    @Override
    public void init() throws ServletException {
        _log.log(LogService.LOG_INFO, "BladeServlet init");

        super.init();
    }

    @Override
    protected void doGet(
            HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {

        _log.log(LogService.LOG_INFO, "doGet");

        _writeSampleHTML(response);
    }

    /**
     * Dummy contents
     *
     * @return dummy contents string
     */
    private String _generateSampleHTML() {
        StringBuffer sb = new StringBuffer();

        sb.append("<html>");
        sb.append("<head><title>Sample HTML</title></head>");
        sb.append("<body>");
        sb.append("<h2>Hello World</h2>");
        sb.append("</body>");
        sb.append("</html>");

        return new String(sb);
    }

    /**
     * Write sample HTML
     *
     * @param resp
     */
    private void _writeSampleHTML(HttpServletResponse resp) {
        resp.setCharacterEncoding("UTF-8");
        resp.setContentType("text/html; charset=UTF-8");
        resp.setStatus(HttpServletResponse.SC_OK);

        try {
```

```
            resp.getWriter().write(_generateSampleHTML());
        }
        catch (Exception e) {
            _log.log(LogService.LOG_WARNING, e.getMessage(), e);

            resp.setStatus(HttpServletResponse.SC_PRECONDITION_FAILED);
        }
    }

    private static final long serialVersionUID = 1L;

    @Reference
    private LogService _log;

}
```

The sample servlet class uses the `@Component` annotation to declare itself an OSGi service of type `Servlet`. It uses OSGi HTTP Whiteboard to respond to requests at URLs matching `http://localhost:8080/o/blade/servlet/*`. Since the component's `osgi.http.whiteboard.context.path` and `osgi.http.whiteboard.servlet.pattern` properties configure the servlet mapping, there's no need to specify one in a `WEB-INF/web.xml` descriptor.

The Portal web application's `WEB-INF/web.xml` defines Liferay's Module Framework Servlet mapping:

```
<servlet-mapping>
    <servlet-name>Module Framework Servlet</servlet-name>
    <url-pattern>/o/*</url-pattern>
</servlet-mapping>
```

The servlet mapping starts at URL pattern `/o/*`. Combined with the `@Component` property `"osgi.http.whiteboard.servlet.pattern=/blade/servlet/*"`, the servlet sample matches URL pattern `/o/blade/servlet/*`.

To develop your own servlet, you can copy and modify all (or part) of the Servlet sample module project or create a servlet in your own module.

Creating a Servlet

Here's how to create your own servlet:

1. Create a module project.
2. Add the necessary dependencies. Here they are for Gradle:

    ```
    compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
    compileOnly group: "org.osgi", name: "org.osgi.service.component.annotations", version: "1.4.0"
    compileOnly group: "org.osgi", name: "org.osgi.service.log", version: "1.4.0"
    ```

3. Create a servlet class that extends `javax.servlet.http.HttpServlet`.
4. Add the following `@Component` annotation.

    ```
    @Component(
        property = {
            "osgi.http.whiteboard.context.path=/",
            "osgi.http.whiteboard.servlet.pattern=/blade/servlet/*"
        },
        service = Servlet.class
    )
    ```

`service = Servlet.class`: Makes the component an OSGi service of type `Servlet`.

5. Set the following `@Component` `property` values to specify a context path and servlet URL pattern:

   `"osgi.http.whiteboard.context.path=/"`: Sets the servlet's context. Replace the value with your servlet's context path.

   `"osgi.http.whiteboard.servlet.pattern=/blade/servlet/*"`: Sets the servlet's mapping pattern. Replace the value with your servlet's pattern.

6. Override `HttpServlet` methods to implement your servlet's behavior.

7. Deploy your module.

Your servlet is up and running. You're well on your way to delivering custom user experiences using servlets.

Related Topics

Servlet Sample
    Servlet Filters
    JAX-RS
    Portlets

148.2 Servlet Filters

Servlet filters can both pre-process requests as they arrive and post-process responses before they go to the client browser. They let you apply functionality to requests and responses for multiple servlets, without the servlets knowing. Here are some common filter use cases:

- Logging
- Auditing
- Transaction management
- Security

You can use patterns in descriptors to map the filters to servlet URLs. When requests arrive at these URLs, your filters process them. Filter chaining lets you apply filters in an order you want. Servlet Filter Hook plugins let you deploy and undeploy filters without modifying the Liferay web application. Here are the steps for creating and deploying a servlet filter:

1. Create a Servlet Filter class

2. Map URLs to your Servlet Filter

3. Create a Liferay plugin descriptor

4. Deploy your plugin

For reference, you can download the example servlet filter project code.

In a traditional web application (`.war`) project, start with creating your servlet filter class.

Note: Portlet filters let you apply functionality to portlet requests and responses. JSP overrides are one way to use portlet filters.

Step 1: Create a Servlet Filter class

Create a class that implements `javax.servlet.Filter`. Here's an example servlet filter class:

```
package com.liferay.sampleservletfilter.hook.filter;

import com.liferay.portal.kernel.util.WebKeys;

import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;

public class SampleFilter implements Filter {

    @Override
    public void destroy() {
        System.out.println("Called SampleFilter.destroy()");
    }

    @Override
    public void doFilter(
            ServletRequest servletRequest, ServletResponse servletResponse,
            FilterChain filterChain)
        throws IOException, ServletException {

        String uri = (String)servletRequest.getAttribute(
            WebKeys.INVOKER_FILTER_URI);

        System.out.println(
            "Called SampleFilter.doFilter(" + servletRequest + ", " +
                servletResponse + ", " + filterChain + ") for URI " + uri);

        filterChain.doFilter(servletRequest, servletResponse);
    }

    @Override
    public void init(FilterConfig filterConfig) {
        System.out.println(
            "Called SampleFilter.init(" + filterConfig + ") where hello=" +
                filterConfig.getInitParameter("hello"));
    }

}
```

Here are the `Filter` methods to implement:

1. `init(FilterConfig)`: Configure the filter and perform any necessary initializations.

   When `SampleFilter` is initialized, for example, its `init(FilterConfig)` method prints the `FilterConfig` object and the `hello` parameter's value:

```
Called SampleFilter.init(com.liferay.portal.kernel.servlet.filters.invoker.InvokerFilterConfig@7c953747) where hello=world
```

2. `doFilter(ServletRequest, ServletResponse, FilterChain)`: Filter on requests and responses here. To apply your filter, invoke `filterChain.doFilter(servletRequest, servletResponse)`.

   When users visit URLs mapped for `SampleFilter`, for example, its `doFilter(...)` method prints the `ServletResponse` object, `FilterChain` object, and the `ServletRequest` URI before passing control to the next filter by invoking `filterChain.doFilter(servletRequest, servletResponse)`.

```
Called SampleFilter.doFilter(org.apache.catalina.connector.RequestFacade@68be71e0, com.liferay.portal.servlet.filters.absoluteredirects.Absol
```

3. `destroy()`: Clean up the filter's unneeded resources.

   When `SampleFilter` is destroyed, its `destroy()` method prints the message: `Called SampleFilter.destroy()`.

It's time to map URLs to your servlet filter.

Step 2: Map URLs to your Servlet Filter

Traditionally, specifying a servlet filter and its filter mapping requires modifying your web application's `web.xml` file. Liferay DXP, however, lets you specify them in your plugin, so you don't need to modify the Liferay DXP web application. Specify your servlet filter mapping in a descriptor file `WEB-INF/liferay-hook.xml`, like this one for Sample Filter:

```
<?xml version="1.0"?>
<!DOCTYPE hook PUBLIC "-//Liferay//DTD Hook 7.1.0//EN" "http://www.liferay.com/dtd/liferay-hook_7_1_0.dtd">

<hook>
    <servlet-filter>
        <servlet-filter-name>Sample Filter</servlet-filter-name>
        <servlet-filter-impl>com.liferay.sampleservletfilter.hook.filter.SampleFilter</servlet-filter-impl>
        <init-param>
            <param-name>hello</param-name>
            <param-value>world</param-value>
        </init-param>
    </servlet-filter>
    <servlet-filter-mapping>
        <servlet-filter-name>Sample Filter</servlet-filter-name>
        <before-filter>SSO Open SSO Filter</before-filter>
        <url-pattern>/group/*</url-pattern>
        <url-pattern>/user/*</url-pattern>
        <url-pattern>/web/*</url-pattern>
        <url-pattern>*.jsp</url-pattern>
        <dispatcher>REQUEST</dispatcher>
        <dispatcher>FORWARD</dispatcher>
    </servlet-filter-mapping>
</hook>
```

Here's how to map URLs to your servlet filter:

1. Create a descriptor file `WEB-INF/liferay-hook.xml`, based on the Liferay Hook DTD.
2. Add a `servlet-filter` element as a sub element of `hook`. Specify your `servlet-filter` sub-elements.

   `servlet-filter-name`: Arbitrary name. (required)

   `servlet-filter-impl`: `Filter` implementation class. (required)

   `init-param` elements: Initialization parameters. (optional)

3. Add a `servlet-filter-mapping` element as a sub element of `hook`.

   `servlet-filter-name`: Match the one used in the `servlet-filter`. (required)

   `after-filter`: Name of a `servlet-filter` for this filter to go after. (optional)

   `before-filter`: Name of a `servlet-filter` for this filter to go before. (optional)

   `url-pattern` elements: URL patterns you want to filter requests and responses for. (required)

   `dispatcher` elements: Specify `Dispatcher` enumerated constants to constrain how the filter is applied to requests. (optional)

Step 3: Create a Liferay plugin descriptor

In a `WEB-INF/liferay-plugin-package.properties` file, specify the versions of Liferay DXP your plugin supports:

```
liferay-versions=7.1.0+
```

Step 4: Deploy your plugin

Specify compile-time dependencies on these artifacts:

- `com.liferay.portal.kernel`
- `javax.servlet-api`

Gradle:

```
compileOnly group: "com.liferay.portal", name: "com.liferay.portal.kernel", version: "3.0.0"
compileOnly group: "javax.servlet", name: "javax.servlet-api", version: "3.0.1"
```

Maven:

```
<dependency>
    <groupId>com.liferay.portal</groupId>
    <artifactId>com.liferay.portal.kernel</artifactId>
    <version>3.0.0</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
</dependency>
```

Build your plugin `.war` file and deploy it by copying it to the `[LIFERAY_HOME]/deploy` folder. Liferay's auto-deployer copies the `.war` to the `[LIFERAY_HOME]/osgi/war` folder. The WAB Generator converts the `.war` to an OSGi Web Application Bundle (WAB) and installs it to Liferay DXP's runtime. The output in your console should look like this:

```
2018-11-03 16:20:09.118 INFO  [fileinstall-C:/workspace_liferay/bundles/osgi/war][BaseAutoDeployListener:43] Copying hook plugin for C:\workspace_li
9.0.6\temp\20181103162009108XCJZAKUY\sample-servlet-filter-hook.war
2018-11-03 16:20:09.390 INFO  [fileinstall-C:/workspace_liferay/bundles/osgi/war][BaseDeployer:880] Deploying sample-servlet-
filter-hook.war
2018-11-03 16:20:10.019 INFO  [fileinstall-C:/workspace_liferay/bundles/osgi/war][BaseAutoDeployListener:50] Hook for C:\workspace_liferay\bundles\t
9.0.6\temp\20181103162009108XCJZAKUY\sample-servlet-filter-hook.war copied successfully
2018-11-03 16:20:10.730 INFO  [Refresh Thread: Equinox Container: 4023060a-c8de-0018-1c3a-ebee784b7a28][BundleStartStopLogger:35] STARTED sample-
servlet-filter-hook_7.1.10.1 [963]
2018-11-03 16:20:11.050 INFO  [Refresh Thread: Equinox Container: 4023060a-c8de-0018-1c3a-ebee784b7a28][HotDeployImpl:226] Deploying sample-
servlet-filter-hook from queue
```

```
2018-11-03 16:20:11.052 INFO  [Refresh Thread: Equinox Container: 4023060a-c8de-0018-1c3a-ebee784b7a28][PluginPackageUtil:1001] Reading plugin packag
servlet-filter-hook
03-Nov-2018 16:20:11.066 INFO [Refresh Thread: Equinox Container: 4023060a-c8de-0018-1c3a-ebee784b7a28] org.apache.catalina.core.ApplicationContext.l
2018-11-03 16:20:11.093 INFO  [Refresh Thread: Equinox Container: 4023060a-c8de-0018-1c3a-ebee784b7a28][HookHotDeployListener:457] Registering hook f
servlet-filter-hook
Called SampleFilter.init(com.liferay.portal.kernel.servlet.filters.invoker.InvokerFilterConfig@7c953747) where hello=world
2018-11-03 16:20:11.134 INFO  [Refresh Thread: Equinox Container: 4023060a-c8de-0018-1c3a-ebee784b7a28][HookHotDeployListener:533] Hook for sample-
servlet-filter-hook is available for use
```

The servlet container calls your filter's `init` method. Deploying `SampleFilter`, for example, invokes its `init` method, which prints this output:

```
Called SampleFilter.init(com.liferay.portal.kernel.servlet.filters.invoker.InvokerFilterConfig@7c953747) where hello=world
```

Visiting the URLs mapped to your servlet filter invokes your filter's `doFilter` method. Since the sample's servlet filter mapping includes the URL `/web/*`, visiting http://localhost:8080/web/guest invokes `SampleFilter.doFilter`, which prints this:

```
Called SampleFilter.doFilter(org.apache.catalina.connector.RequestFacade@68be71e0, com.liferay.portal.servlet.filters.absoluteredirects.AbsoluteRedi
```

Undeploying your servlet filter `.war` invokes its `destroy()` method.

Congratulations on filtering requests to your site's URLs.

Related Topics

Configuring Dependencies

CHAPTER 149

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. Liferay lets you use whatever testing framework you want. There's JUnit for unit testing, Arquillian plus JUnit annotations for integration testing, and more; consult the testing framework's documentation for details.

Here are the ways Liferay facilitates testing:

- Injecting Service Components into Tests: Liferay's `@Inject` annotation allows you to inject service instances into tests.

149.1 Injecting Service Components into Integration Tests

You can use Liferay DXP's `@Inject` annotation to inject service components into an integration test, like you use the `@Reference` annotation to inject service components into an OSGi component.

Note: Arquillian plus JUnit annotations is one way to develop integration tests. Liferay lets you use whatever testing framework you want.

`@Inject` uses reflection to inject a field with a service component object matching the field's interface. Test rule `LiferayIntegrationTestRule` provides the annotation. The annotation accepts `filter` and `type` parameters, which you can use separately or together.

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

CHAPTER 150

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'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 applications, you and your team can enjoy developing your own modules, applications, and customizations in parallel.

150.1 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.

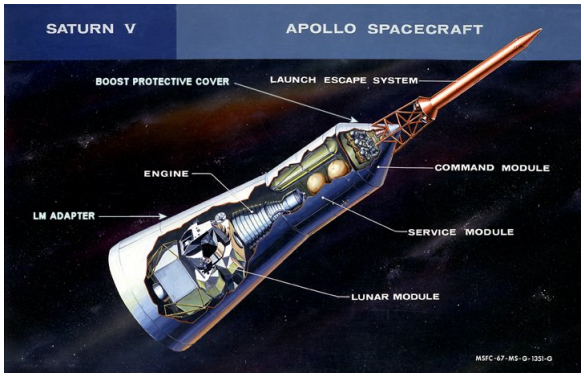


Figure 150.1: The Apollo spacecraft's modules collectively took astronauts to the moon's surface and back to Earth.

- **Dependencies**: Modules can require capabilities other modules satisfy.

  The Apollo modules had these dependencies:

  - 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.

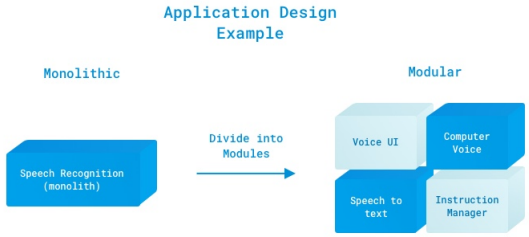


Figure 150.2: The speech recognition application can be implemented in a single monolithic code base or in modules, each focused on a particular function.

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.

For example, the *Instruction manager* and *Computer voice* modules can be *reused* by a navigation app.

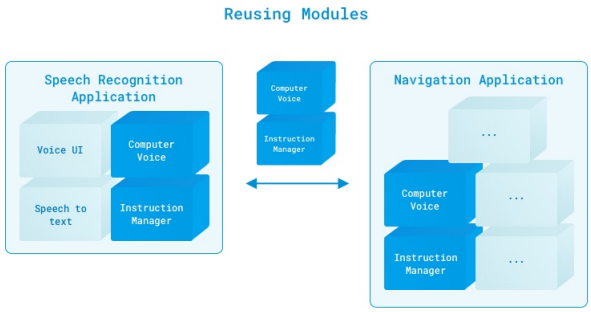


Figure 150.3: The *Instruction manager* and *Computer voice* modules designed for the speech recognition app can be used (or *reused*) by a navigation app.

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.

150.2 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 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'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'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 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
    Planning a Plugin Upgrade to Liferay 7

150.3 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 dependencies facilitate modular development

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

How Dependencies Work

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.

Note: Since Liferay 7.0, 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.

Versioning is independent for each 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 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 and projects created from Liferay project templates) 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 you 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, 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.

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

Liferay DXP'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. Liferay'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.

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 Liferay'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, explore OSGi Services and dependency injection using OSGi Declarative Services.

If you visited this tutorial as a part of the Learning Path From Liferay Portal 6 to 7.1, you can continue with the next topic: OSGi Services and dependency injection using OSGi Declarative Services.

Related Topics

Configuring Dependencies
    Importing Packages
    Exporting Packages

150.4 OSGi Services and Dependency Injection with Declarative Services

In Liferay DXP, 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 components 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 do you use DS to register and bind services? Does it involve creating XML files? No, it's much easier than that. You use 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.

Note: The `@Reference` annotation can only be used in a class that is annotated with `@Component`. That is, only a Declarative Services component can use `@Reference` to bind to an OSGi service.

At build time in modules created from Liferay project templates, 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. You can create and publish service components for other classes to use. You can update the components and even publish alternative component implementations for a service. This kind of dynamism is a powerful part of Liferay DXP.

If you visited this tutorial as a part of the Learning Path From Liferay Portal 6 to 7.1, you can with the next topic: dynamic deployment in OSGi.

150.5 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 `[Liferay Home]/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 the module `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 you manage the module lifecycle. You can install/uninstall modules and start/stop them. You can update a module and notify dependent modules to use the

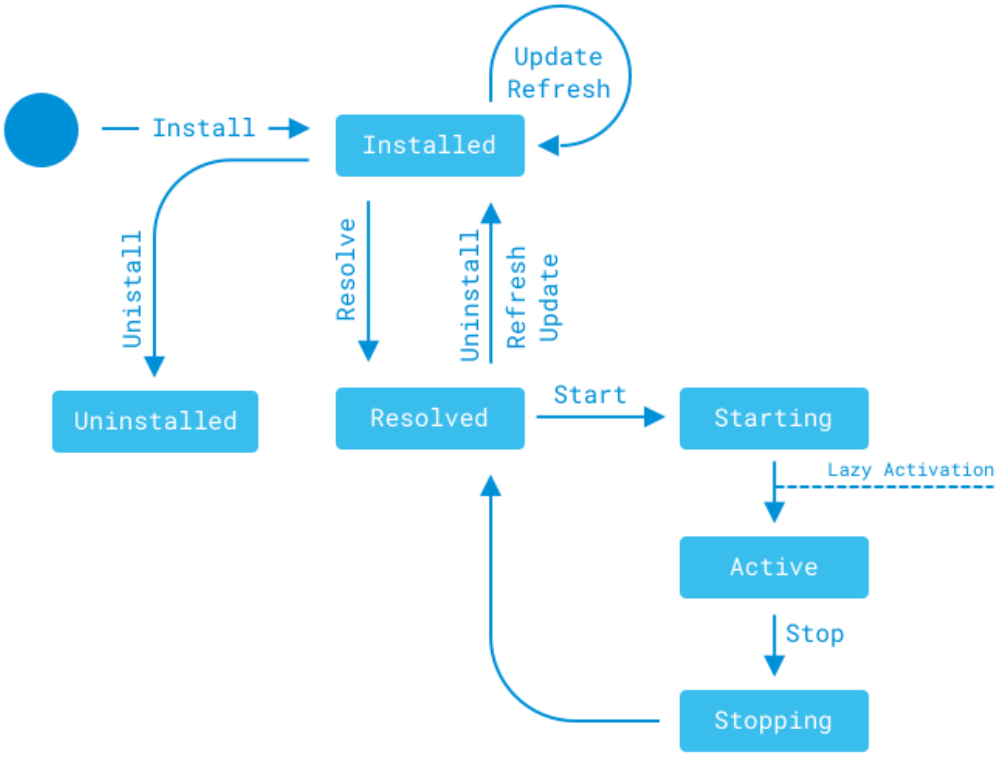


Figure 150.4: This state diagram illustrates the module lifecycle.

update. Liferay's tools, including Liferay Dev Studio DXP, 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,
    ...
)
```

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, you 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 (whose service policy is greedy) 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.

If you visited this tutorial as a part of the Learning Path From Liferay Portal 6 to 7.1, you can go here to continue it.

Related Topics

Starting Module Development
Planning Plugin Upgrades

150.6 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.

If you're ready to dive deep into OSGi, 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 151

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 developing OSGi bundles (modules).

151.1 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 classloaders. All the classloaders from Java's Bootstrap classloader to classloaders for bundle classes and JSPs are part of a hierarchy.

This tutorial explains Liferay'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'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's OSGi module framework classloader which is used to provide controlled isolation for the module framework bundles.

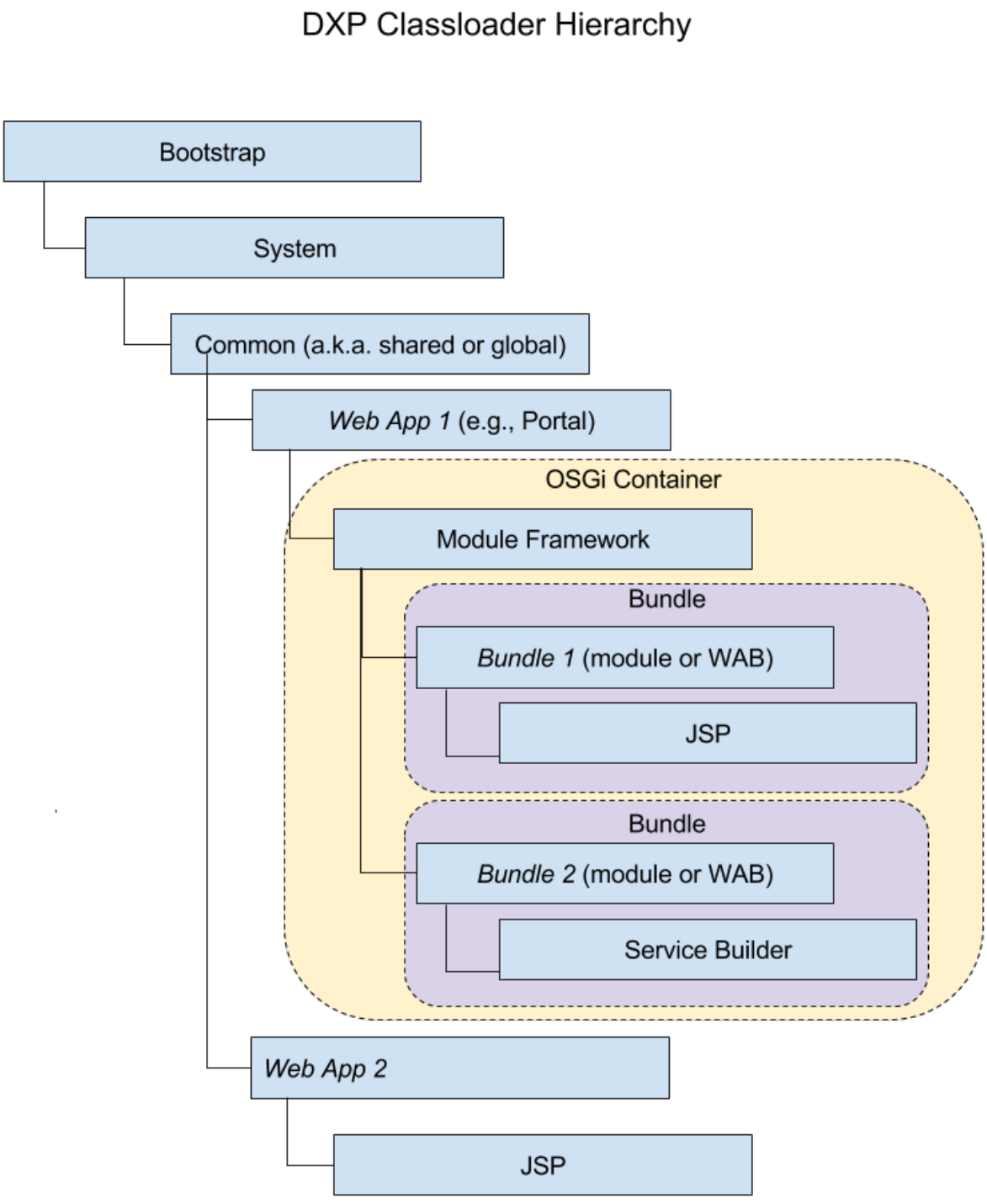


Figure 151.1: 0: Here is Liferay's classloader hierarchy.

- **bundle**: Classes from a bundle's packages or from packages other bundles export.
- **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

151.2 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.

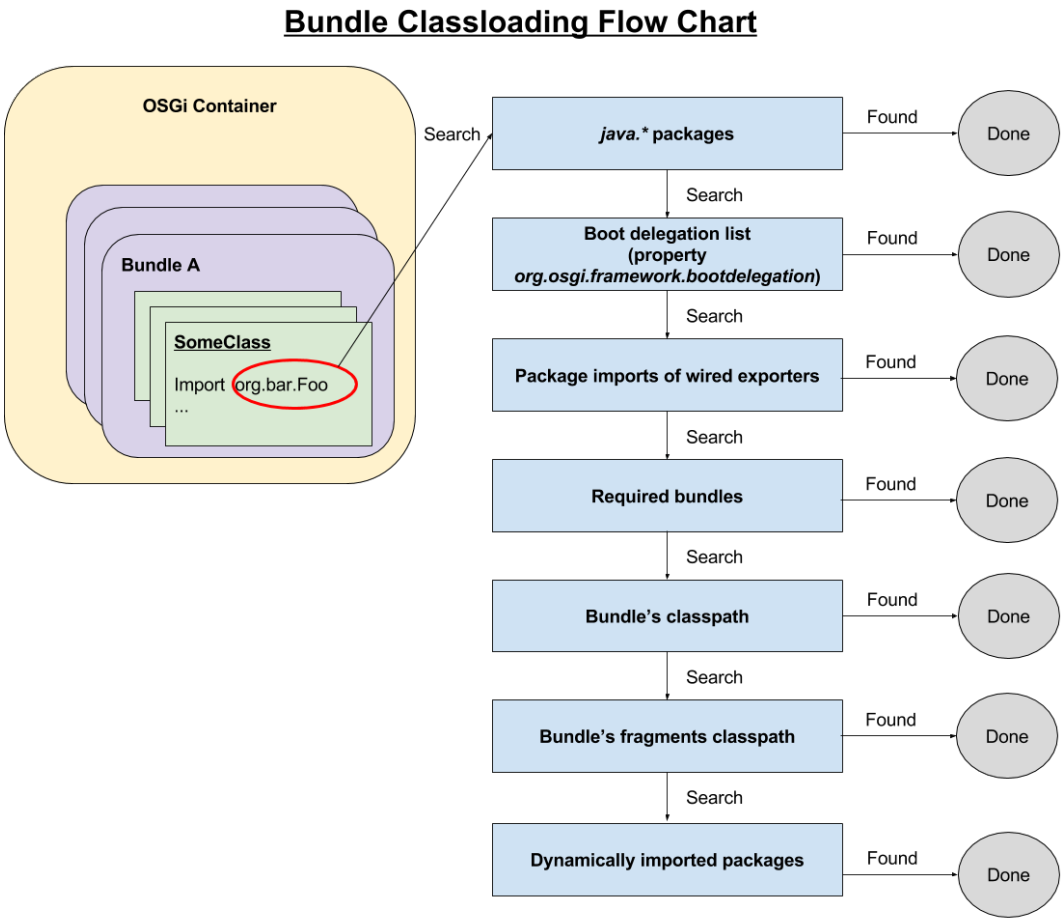


Figure 151.2: 0: This flow chart illustrates classloading in a bundle.

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 for OSGi bundles.

151.3 Importing Packages

Your modules often must 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. Here are the different package import scenarios:

- Automatic Package Import Generation
- Manually Adding Package Imports

Let's explore how package imports are specified in these scenarios.

Automatic Package Import Generation

Gradle and Maven module projects created using Blade CLI, Liferay's Maven archetypes, or Liferay Dev Studio DXP 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.

Note: Packages for Java APIs, such as Java Portlet, aren't semantically versioned but have Portable Java Contracts. Each API's contract specifies the JSR it satisfies. Modules that use these APIs must specify requirements on the API contracts. The contract requirement specifies your module's relationship with the imported API packages. If the system you're running does *not* provide the exact contract, your module does not resolve. Resolving the missing package is better than handling an incompatibility failure during execution.

- **Blade CLI and Liferay Dev Studio DXP module projects** specify Portable Java Contracts automatically! For example, if your Blade CLI or Liferay Dev Studio DXP module uses the Java Portlet API and you compile against the Java Portlet 2.0 artifact, a contract requirement for the package is added to your module's manifest.

- **Module projects that use bnd but are not created using Blade CLI or Liferay Dev Studio DXP** must specify contracts in their `bnd.bnd` file. For example, here are contract instructions for Java Portlet and Java Servlet APIs:

  ```
  -contract: JavaPortlet,JavaServlet
  ```

  At build time, bnd adds the contract instructions to your module's manifest. It adds a requirement for the first version of the API found in your classpath and *removes* version range information from `Import-Package` entries for corresponding API packages—the package version information isn't needed.

- **Projects that don't use bnd** must specify contracts in their module manifest. For example, here's the specified contract for `JavaPortlet 2.0`, which goes in your `META-INF/MANIFEST.MF` file:

```
Import-Package: javax.portlet
Require-Capability: osgi.contract;filter:=(&(osgi.contract=JavaPortlet)(version=2.0))
```

For Portable Java Contract details, see Portable Java Contract Definitions.

Note: Liferay DXP 7.1 GA1 exports the Java Portlet 2.0 API. Until Java Portlet 3.0 API is supported, make sure to use the 2.0 version.

Manually Adding Package Imports

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

151.4 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 Dev Studio DXP 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

151.5 Resolving Third Party Library Package Dependencies

The OSGi framework lets you build applications composed of multiple OSGi bundles (modules). For the framework to assemble the modules into a working system, the modules must resolve their Java package dependencies. In a perfect world, every Java library would be an OSGi module, but many libraries aren't. So how do you resolve the packages your 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` dependency for it in your module. When you deploy your module, the OSGi framework wires the dependency module to your module. If you don't find an 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 module or embed them wholesale, if you must. The rest of the tutorial shows you how to do these things.

Note: Liferay's Gradle plugin `com.liferay.plugin` automates several third party library configuration steps. The plugin is automatically applied to Liferay Workspace Gradle module projects created using Liferay Dev Studio DXP or Liferay Blade CLI.

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 must 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 module more modular. This also keeps other modules that depend on your module from incorporating 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 by name in an `-includeresource:` instruction without having to add them explicitly to the dependency list. 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 might 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 Projects with Blade CLI

151.6 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.ServiceTrackerCustomizer` 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
    Liferay DXP Startup Phases

151.7 Using the WAB Generator

You can create applications for Liferay DXP as Java EE-style Web Application ARchive (WAR) artifacts or as Java ARchive (JAR) OSGi bundle artifacts. Some portlet types, however, limit your 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*.

Liferay DXP 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 project source 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 traditional WAR-style plugin into a WAB during deployment. 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 the 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 that header's list of values.

- Unrecognized descriptor file
- Custom or unrecognized descriptor element or attribute
- Reflection code
- Classloader code

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 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 your plugin's `liferay-plugin-package.properties` file. It's pointless to add a `bnd.bnd` file or a build-time plugin (e.g., `bnd-maven-plugin`) to your WAR plugin, because the generated WAB cannot use of them.

Do you want to try generating a WAB? 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 instance's `osgi/wabs` folder. You can learn more about these properties in the Module Framework Web Application Bundles properties section. Restart Liferay to use these properties.

3. Copy your WAR plugin in your Liferay instance's `deploy` folder.
4. Navigate to your Liferay instance's `osgi/wabs` folder and inspect the generated WAB.

Awesome! You've seen the WAB Generator in action!

Related Topics

Configurable Applications

151.8 Service Trackers

In an OSGi runtime ecosystem, you must consider how your modules can rely on services in other modules for functionality. It's possible for service implementations to be swapped out or removed entirely, and your module must not just survive but thrive in this environment.

If you call services from `@Component` classes, it's easy: you just use another Declarative Services (DS) annotation, `@Reference`, to get a service reference. The component activates when the referenced service is available.

Note: The `@Reference` annotation can only be used in a class that is annotated with `@Component`. That is, only a Declarative Services component can use `@Reference` to bind to an OSGi service.

If you can 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 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. If you use a static class, you might

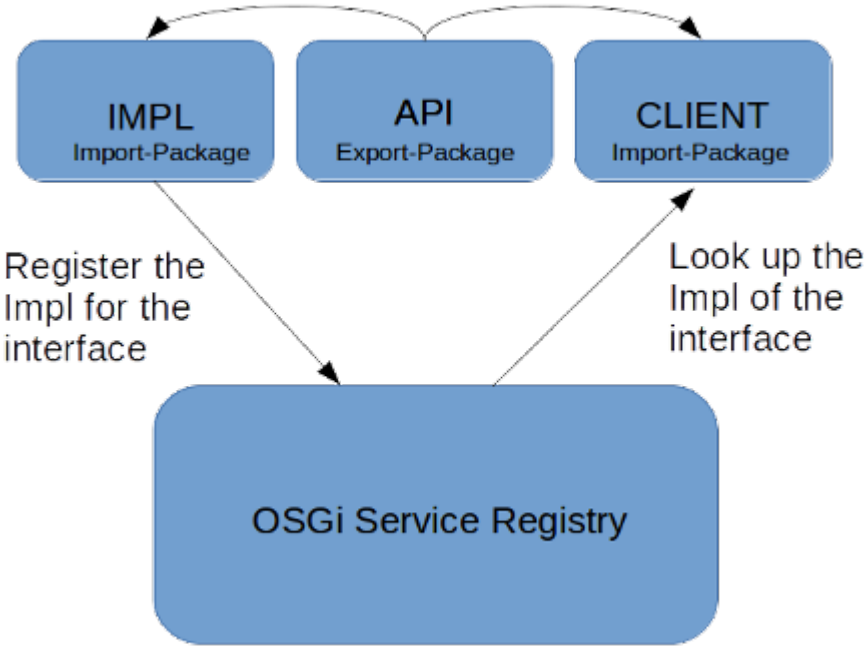


Figure 151.3: Service implementations that are registered in the OSGi service registry can be accessed using Service Trackers.

attempt calling a stopped service or one that hasn't been deployed or started. This could cause unrecoverable runtime errors. Service Trackers, however, help you make OSGi-friendly service calls.

Your non-OSGi application can access any service registered in the OSGi runtime using a Service Tracker, including your own Service Builder services and the services published by Liferay's modules (like the popular `UserLocalService`).

Implementing a Service Tracker

Service Trackers don't give you the luxury of managing your service dependencies with DS, but you can call services from the service registry.

You can implement a service tracker in two ways: 1) In the code where you need it, or 2) In a class that extends `org.osgi.util.tracker.ServiceTracker`.

To create it directly, 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);
```

A better way is to create a class that extends `org.osgi.util.tracker.ServiceTracker`, because this simplifies your code.

1. Create a class like this one that extends `ServiceTracker`:

```
public class SomeServiceTracker
    extends ServiceTracker<SomeService, SomeService> {

    public SomeServiceTracker(Object host) {
        super(
            FrameworkUtil.getBundle(host.getClass()).getBundleContext(),
            SomeService.class, null);
    }
}
```

2. From the initialization part of your logic that uses the service, call your service tracker constructor. The `Object host` parameter obtains your own bundle context and must be an object from your own bundle in order to give accurate results.

```
ServiceTracker<SomeService, SomeService> someServiceTracker =
    new SomeServiceTracker(this);
```

3. When you want to use the service tracker, open it, typically as early as you can.

```
someServiceTracker.open();
```

4. Before attempting to use a service, use the Service Tracker 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.

5. Later when your application is being destroyed or undeployed, close the service tracker.

```
someServiceTracker.close();
```

Service Trackers make it possible to call OSGi services from outside the OSGi runtime.

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 that 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.ServiceTracker`

```
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.

151.9 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.

CHAPTER 152

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 apps and modules, you want to resolve them fast. These frequently asked questions and answers help you troubleshoot and correct problems.

Here are the troubleshooting sections:

- Modules
- Services and Components
- Front-end

Click a question to view the answer.

152.1 Modules

How can I configure dependencies on Liferay artifacts?

```
<p>See <a href="/docs/7-1/tutorials/-/knowledge_base/t/configuring-dependencies">Configuring Dependencies</a>. </p>
```

What are optional package imports and how can I specify them?

```
<p>When developing modules, you can declare <em>optional</em> package imports. An optional package import is one your module can use if it's availabl
1/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's OSGi environment is almost the same as connecting to them from the Java EE
1/tutorials/-/knowledge_base/t/connecting-to-data-sources-using-jndi">use Liferay DXP's class loader to load the application server's JNDI classes</a
```

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 DXP reports an unresolved requirement:<
<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/7-1/tutorials/-/knowledge_base/t/resolving-bundle-requirements">find a module that provides the capabil
```

An IllegalContextNameException reports that my bundle's context name does not follow Bundle-SymbolicName syntax. How can I fix the context name?

Adjust the <code>Bundle-SymbolicName</code> to adhere to the syntax.

Why aren't my module's JavaScript and CSS changes showing?

Incorrect component properties or stale

Why aren't my fragment's JSP overrides showing?

Make sure your <code>Fragment-Host</code>'s bundle version is compatible with the host's bundle version.

Why doesn't the package I use from the fragment host resolve?

Refrain from importing (<code>Import-Package: ...</code>) host packages that the host doesn't export.

The application server and database started, but Liferay DXP failed to connect to the database. What happened and how can I fix this?

Liferay DXP initialization can fail while attempting to connect to a database server that isn't ready. Configuring startup to retry JDBC connections facil

How can I adjust my module's logging?

See Adjusting Module Logging.

How can I implement logging in my module or plugin?

Use Simple Logging Facade for Java (SLF4J) to log messages.

Why did the entity sort order change when I migrated to a new database type?

Your new database uses a different default query re
you should be able to configure a different order.

After creating a relational mapping between Service Builder entities, my portlet is using too much memory. What can I do?

<a href="/docs/7-1/tutorials/-/knowledge_base/t/disabling-cache-for-table-mapper-tables">Disabling the cache related to the entity mapping lowers

152.2 Services and Components

How can I see what's happening in the OSGi container?

```
<p><a href="/docs/7-1/tutorials/-/knowledge_base/t/system-check">Run a System Check.</a>. </p>
```

How can I detect unresolved OSGi components?

```
<p>module components that use Service Builder use Dependency Manager (DM) and most other module components use Declarative Services (DS). <a href="/d
1/tutorials/-/knowledge_base/t/detecting-unresolved-osgi-components">Gogo shell commands and tools help you find and inspect unsatisfied component re
```

What is the safest way to call OSGi services from non-OSGi code?

```
<p>See <a href="/docs/7-1/tutorials/-/knowledge_base/t/service-trackers">Calling Non-OSGi Code that Uses OSGi Services</a>. </p>
```

How can I use files to configure components?

```
<p>See <a href="/docs/7-1/tutorials/-/knowledge_base/t/using-files-to-configure-product-modules">Using Files to Configure Module Components</a>. </p>
```

How can I access OSGi Services from my Ext plugin?

```
<p><a href="/docs/7-1/tutorials/-/knowledge_base/t/using-osgi-services-from-ext-plugins">Use `ServiceTrackers`</a>. </p>
```

152.3 Resolving Bundle Requirements

If one of your bundles needs a package that is not exported by any other bundle in the Liferay OSGi runtime, 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 modul
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"; version="1.0.0"; uses:="org.osgi.framework"
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.lifer
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 line states *could not resolve the bundles*. What follows is a string of requirements that Lferay's OSGi Runtime could not resolve.

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 unsatisfied requirement. 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.

The `com.liferay.petra.io` module's `build.gradle` file lists its dependencies:

```
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 are 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
...
```

The dependency module `org.apache.aries.spifly.dynamic.bundle` is missing from the runtime bundle list. The `org.apache.aries.spifly.dynamic.bundle` 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"
```

This capability `osgi.extender; filter:="(osgi.extender=osgi.serviceloader.processor)"` is the unresolved requirement we identified earlier. Deploying this missing bundle `org.apache.aries.spifly.dynamic.bundle` satisfies the example module's requirement and allows the module to resolve and install.

You can resolve your bundle exceptions by following steps similar to these.

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

152.4 Resolving Bundle-SymbolicName Syntax Issues

Liferay's OSGi Runtime framework sometimes throws an `IllegalContextNameException`. Often, this is because an OSGi bundle's `Bundle-SymbolicName` manifest header has a space in it.

The `Bundle-SymbolicName` uniquely identifies the bundle—along with the `Bundle-Version` manifest header—and cannot contain spaces. To follow naming best practices, use a reverse-domain name in your `Bundle-SymbolicName`. For example, a module with the domain `troubleshooting.liferay.com` would be reversed to `com.liferay.troubleshooting.`.

There are three ways to specify a bundle's `Bundle-SymbolicName`:

1. `Bundle-SymbolicName` header in a bundle'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.

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.
```

However you set your a `Bundle-SymbolicName`, refrain from using spaces.

Related Topics

Using the WAB Generator

152.5 Resolving ClassNotFoundException and NoClassDefFoundError in OSGi Bundles

`ClassNotFoundException` and `NoClassDefFoundError` are common, well known exceptions:

- `ClassNotFoundException` is 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:

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 the package, a `ClassNotFoundException` or `NoClassDefFoundError` occurs.

   Check the package name and make sure the consuming module imports the right package. If the 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:** Modules are changed frequently in OSGi runtime environments. 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 limited options when 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 NoClassDefFoundError?.

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! Proceed only if you have no other solution, and watch out for unintended consequences. There are two ways to export the package:

1. In your `portal-ext.properties` file, use the property `module.framework.system.packages.extra` to specify the packages to export. Preserve the property's current list.

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 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

152.6 Identifying Liferay Artifact Versions for Dependencies

When you're developing an application using Liferay APIs or tools—for example, you might create a Service Builder application or use Message Bus or Asset Framework—you must determine which versions of Liferay artifacts (modules, apps, etc.) you application's modules must specify as dependencies. To learn how to find Liferay artifacts and configure dependencies on them, see Configuring Dependencies.

Related Topics

Configuring Dependencies
    Finding Extension Points

152.7 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.

```
thread.setContextClassLoader(PortalClassLoaderUtil.getClassLoader());
```

It uses JNDI to look up the data source.

```
InitialContext ctx = new InitialContext();
DataSource datasource = (DataSource)
    ctx.lookup("java:comp/env/jdbc/TestDB");
```

After working with the data source, the code reinstates the thread's original classloader.

```
 thread.setContextClassLoader(origLoader);
```

Here are the class imports for the example code:

```
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 Databases

152.8 Adjusting Module Logging

Liferay DXP uses Log4j logging services. Here are the ways to configure logging for module classes and class hierarchies.

- *Log Levels* in 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
- INFO
- WARN
- ERROR
- FATAL
- OFF

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).

That's it for module log configuration. You're all set to print the information you want.

Related Topics

Implementing Logging

152.9 Implementing Logging

7.0 uses the Log4j logging framework, but it may be replaced in the future. 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.

Related Topics

Adjusting Module Logging

152.10 Declaring Optional Import Package Requirements

When developing modules, you can declare *optional* dependencies. An optional dependency is one your module can use if 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 more than one distinct type 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. In your module's `bnd.bnd` file, declare the package your module optionally depends on:

   ```
   Import-Package: com.liferay.demo.foo;resolution:="optional"
   ```

   Note that you can use either an optional or dynamic import. The differences are explained here.

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 be a controller for the first. The second component checks the classloader for the optional class on the classpath. 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).

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 Gogo shell with this command:

```
equinox:refresh [bundle ID]
```

Related Topics

Configuring Dependencies

152.11 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

152.12 Why Aren't JSP overrides I Made Using Fragments Showing?

Important: It's strongly recommended to customize JSPs using Liferay DXP's API. Since overriding a JSP using an OSGi fragment is not based on APIs there's no way to guarantee that it will 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. Overriding a JSP using a fragment should only be used as a last resort.

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"
```

Finding versions of deployed modules is straightforward.

Related Topics

JSP Overrides using Portlet Filters
Customizing JSPs
Configuring Dependencies

152.13 Why doesn't the package I use from the fragment host resolve?

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.

Resolve the issue by explicitly excluding host packages that the host doesn't export.

For example, this fragment bundle's JSP uses classes from the fragment host bundle's internal package `com.liferay.portal.search.web.internal.custom.facet.display.context`:

```
<%@
page import="com.liferay.portal.search.web.internal.custom.facet.display.context.CustomFacetDisplayContext" %><%@
page import="com.liferay.portal.search.web.internal.custom.facet.display.context.CustomFacetTermDisplayContext" %>
```

Since the example host bundle doesn't export the package, the fragment bundle can avoid importing the package by using an OSGi manifest header, like the one below, to explicitly exclude the package from package imports:

```
Import-Package: !com.liferay.portal.search.web.internal.*,*
```

152.14 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.

152.15 Disabling Cache for Table Mapper Tables

Service Builder creates relational mappings between entities. It uses mapping tables to associate the entities. In your `service.xml` file, both entities have a `mapping-table` column attribute of the format `mapping-table="table1_table2"`. For example, a `service.xml` that maps `AssetEntrys` to `AssetCategorys` has an `AssetCategory` entity with this column:

```
<column entity="AssetEntry"
mapping-table="AssetEntries_AssetCategories"
name="entries" type="Collection" />
```

and an `AssetEntry` entity element with this column:

```
<column entity="AssetCategory"
mapping-table="AssetEntries_AssetCategories"
name="categories" type="Collection" />
```

By default, a table mapper cache is associated with each mapping table. The cache optimizes object retrieval. In some cases, however, it's best to disable a table mapper cache.

Why would I want to disable cache on a table mapper?

Super-large entity tables can result in a memory-hogging table mapper cache. For this reason, consider disabling cache on a table mapper.

The `table.mapper.cacheless.mapping.table.names` Portal property disables cache for table mappers associated with the specified mapping tables. Here's the default property setting:

```
##
## Table Mapper
##

  #
  # Set a list of comma delimited mapping table names that will not be using
  # cache in their table mappers.
  #
  table.mapper.cacheless.mapping.table.names=\
    Users_Groups,\
    Users_Orgs,\
    Users_Roles,\
    Users_Teams,\
    Users_UserGroups
```

All of the disabled caches above pertain to the `User` object because the table mappers tend to be much too large to have a useful cache—each `User` can have several entries in each related table.

Potential race conditions retrieving objects from the cache is another reason to disable a table mapper.

For example, LPS-84374 describes a race condition in which a custom entity's table mapper cache can be cleared while in use, causing transactional rollbacks. Publishing `AssetEntry`s clears all associated table mapper caches. If they're published at the same time getter methods are retrieving objects from the `AssetEntries_AssetCategories` mapping table, transaction rollbacks occur.

Disabling a Table Mapper Cache

Adding a mapping table name to the `table.mapper.cacheless.mapping.table.names` Portal property disables the associated table mapper cache.

1. In your `[Liferay_Home]/portal-ext.properties` file, add the current `table.mapper.cacheless.mapping.table.name` property setting. The setting is in your Liferay DXP installation's `portal-impl.jar/portal.properties` file.

2. Append your mapping table name to the list. For example, to disable the cache associated with a mapping table named `AssetEntries_AssetCategories`, add that name to the list.

```
table.mapper.cacheless.mapping.table.names=\
    Users_Groups,\
    Users_Orgs,\
    Users_Roles,\
    Users_Teams,\
    Users_UserGroups,\
    AssetEntries_AssetCategories
```

3. Restart the Liferay DXP instance to delete the table mapper cache.

You've disabled an unwanted table mapper cache.

152.16 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 *DXP's* hood?" And since you get access to DXP Digital Enterprise (DXP)'s source code, you can attach a debugger and 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 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 below on 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 command line 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-9.0.6/webapps/ROOT/
   global.lib.path=../tomcat-9.0.6/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 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

Troubleshooting FAQ

Liferay Dev Studio DXP

152.17 Troubleshooting Front-End Development Issues

Front-end development involves many moving parts. Sometimes it's hard to tell what may be causing the issues you run into along the way. This can be particularly frustrating. These frequently asked questions and answers help you troubleshoot and correct problems arising during front-end development.

Here are the troubleshooting sections:

- CSS
- Modules

- Portlets
- Templates
- Themes

Click a question to view the answer.

CSS

- Why are my CSS templates not applied in my Angular app?
- Why is Liferay Portal's CSS broken in Internet Explorer?

Why are my CSS templates not applied in my Angular app?

```
<p>A known <a href="https://github.com/angular/angular/issues/4974">bug</a> with Angular causes absolute URLs for CSS files not to be recognized.</p>
<p>Due to the nature of portals, a relative URL is not an option either because the app can be placed on any page.</p>
<p>To fix this, you can either provide the CSS with a theme or themelet, or you can specify the path to the CSS file with the <code>com.liferay.portl
portlet-css</code> property in the portlet containing your Angular code.</p>
```

Why is Liferay Portal's CSS broken in Internet Explorer?

```
<p>By default CSS files are minified in the browser. This can cause issues in Internet Explorer. You can disable this behavior by including <code>the
ext.properties</code> file. </p>
```

Modules

- Why does my JQuery module throw an anonymous module error when I try to load it?
- Why are my source maps not showing for my Angular or Typescript module?
- I'm using the liferay-npm-bundler for multiple projects. How can I disable analytics tracking for the entire tool across all my projects?

Why does my JQuery module throw an anonymous module error when I try to load it?

```
<p>If you're using an external library that you host, you must disable the <i>Expose Global</i> option as described in the <a href="https://portal.li
1/tutorials/-/knowledge_base/t/using-external-javascript-libraries#using-libraries-that-you-host">Using External JavaScript Libraries</a> tutorial.</
```

Why are my source maps not showing for my Angular or Typescript module?

```
<p>This is due to <a href="https://issues.liferay.com/browse/LPS-83052">LPS-83052</a>.</p>
<p>To solve this, activate the <a href="https://www.typescriptlang.org/docs/handbook/compiler-options.html"><code>inlineSources</code> compiler optio
```

I'm using the liferay-npm-bundler for multiple projects. How can I disable analytics tracking for the liferay-npm-bundler in my projects?

```
<p>There are a couple options you can use to disable reporting:</p>
<ul>
  <li><p>Use the <code>--no-tracking</code> flag in your <code>package.json</code>'s build script to disable reporting:</p>
  <p><pre><code>liferay-npm-bundler --no-tracking</code></pre></p>
  </li>
  <li>
  <p>Create a <code>.liferay-npm-bundler-no-tracking</code> file in your project's root folder, or any of its ancestors, to disable reporting.</p>
  <p>This equates to answering <code>No</code> to the <code>May liferay-npm-bundler anonymously report usage statistics to improve the tool over time
  </li>
</ul>
```

Portlets

- I want to use a custom router in my Angular/React/Vue portlet. How can I disable the default Senna JS SPA engine in my portlet?

I want to use a custom router in my Angular/React/Vue portlet. How can I disable the default Senna JS SPA engine in my portlet?

```
<p>By default, the <a href="https://portal.liferay.dev/docs/7-1/tutorials/-/knowledge_base/t/automatic-single-page-applications#what-
is-sennajs">Senna JS SPA engine</a> is enabled in your portlets and sites. This disables full page reloads during portlet navigation.</p>
<p>If you want to use a custom router in your portlet instead, follow the <a href="https://portal.liferay.dev/docs/7-1/tutorials/-
/knowledge_base/t/automatic-single-page-applications#disabling-spa">instructions in the SPA documentation</a> to blacklist your portlet from SPA.</p
```

Templates

- Why does my web content break when I refresh the page?

Why does my web content break when I refresh the page?

```
<p>Some taglibs, such as the <code>liferay-map</code> taglib, have limitations when used in a cacheable template (e.g., FreeMarker and Velocity). For
map</code> taglib is used in a cacheable template and the user refreshes the page, the map does not show. </p>
<p>One possible workaround is to disable cache for the template by editing it and unchecking the cacheable option. Alternatively, you can disable cac
<p>As best practice, however, we recommend that you don't use taglibs in cacheable web content. </p>
```

Themes

- How can I use the Classic theme as my base theme?
- How can I include OSGi headers in my theme?
- Why aren't my changes showing up after I redeploy my theme?
- Why is my theme not loading? It returns the default theme instead.
- How can I prevent specific CSS rules from transforming for RTL Languages?

How can I use the Classic theme as my base theme?

```
<p>The Classic theme is already an implementation of an existing base theme and should not be extended. You can use the <a href="/docs/7-
1/tutorials/-/knowledge_base/t/copying-an-existing-themes-files">Gulp kickstart</a> task to copy files from the Classic theme into your theme if you
```

How can I include OSGi headers in my theme?

```
<p>Specify the headers you want to use in your theme's <code>liferay-plugin-package.properties</code> file. Any headers placed in this file are inclu
<p>For example, you can add OSGi dependencies in your theme by importing the exported package with the <code>Import-Package</code> header:</p>
<pre><code>Import-Package:com.liferay.docs.portlet</code></pre>
```

Why aren't my changes showing up after I redeploy my theme?

```
<p>By default CSS, JS, and theme template files are cached in the browser. During development, you can enable <a href="/docs/7-
1/tutorials/-/knowledge_base/t/using-developer-mode-with-themes">Devloper Mode</a> to prevent your theme's files from caching. </p>
```

Why is my theme not loading? It returns the default theme instead.

```
<p>If you receive the warning "No theme found for specified theme id...", you may be referencing an outdated theme ID in your Site. Verify that the t
look-and-feel.xml</code> matches the theme ID in the warning message: "mytheme_WAR_mytheme". If the theme IDs match, there may be pages using the out
```

How can I prevent specific CSS rules from transforming for RTL Languages?

```
<p>You can prevent specific CSS rules from transforming (flipping) with the <code>/* @noflip */</code> decoration. Place the decoration to the left o
  <pre><code>
  /* @noflip */ body {
   margin-left: 20em;
  }
  </pre></code>
<p>You can also use the <code>.rtl</code> CSS selector for rules that exclusively apply to RTL languages.</p>
```

152.18 System Check

During development, all kinds of strange things can happen in the OSGi container. Liferay's `system:check` Gogo shell command can help you see what's happening. You can enable it to run as the last Portal startup step and you can execute it any time in Gogo shell.

`system:check` aggregates these commands:

- `ds:unsatisfied`: Reports unsatisfied Declarative Service components.
- `dm na`: Reports unsatisfied Dependency Manager service components, including Service Builder services.

System checking functionality from future Liferay tools will be added to `system:check`.

Developer mode runs `system:check` automatically on every startup.

You can enable `system:check` to run on startup outside of developer mode by setting this property in your `portal-ext.properties` file:

```
module.framework.properties.initial.system.check.enabled=true
```

As stated previously, you can run the `system:check` command any time in Gogo shell. Enjoy detecting unresolved components and other issues fast using `system:check`.

Related Topics

Detecting Unresolved OSGi Components

Gogo shell

152.19 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 Dependency Manager (DM) components. Both Declarative Services and Dependency Manager are Apache Felix projects.

The unresolved component troubleshooting instructions are divided into these sections:

- Declarative Services Components
  - Declarative Services Unsatisfied Component Scanner
  - ds:unsatisfied Command
- Service Builder Components
  - Unavailable Component Scanner
  - dm na Command
  - `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:

- Enable a Declarative Services Unsatisfied Component Scanner to report unsatisfied references automatically or
- Use the Gogo shell command `ds:unsatisfied` to check for them manually.

Declarative Services Unsatisfied Component Scanner

Here's how to enable the unsatisfied component scanner:

1. Create a file `com.liferay.portal.osgi.debug.declarative.service.internal.configuration.UnsatisfiedComponer`
2. Add the following content:

   ```
   unsatisfiedComponentScanningInterval=5
   ```

3. Copy the file into `[LIFERAY_HOME]/osgi/configs`.

The scanner detects and logs unsatisfied service component references. The log message describes the bundle, the referencing DS component class, and the referenced component.

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 the `com.liferay.blogs.web` bundle's DS component `com.liferay.blogs.web.internal.portlet.action.EditEntryMVCRenderCommand` has an unsatisfied reference to a component of type `ItemSelectorHelper`. The referencing component's ID (SCR ID) is `3333` and its bundle ID is `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 DS components
- `ds:unsatisfied [BUNDLE_ID]` shows the bundle's unsatisfied DS components

To view more detailed information about the unsatisfied DS component, pass the component's ID to the command `scr:info [component ID]`. For example, the following command does this for a component with 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

Service Builder modules are implemented using Spring. Liferay DXP uses the Apache Felix Dependency Manager to manage Service Builder module OSGi components via the Portal Spring Extender module.

When developing a Liferay Service Builder application, you might sometimes have an unresolved Spring-related OSGi component. This can occur 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
- dm na Command
- ServiceProxyFactory

Unavailable Component Scanner

The OSGi Debug Spring Extender module's Unavailable Component Scanner reports missing components in modules that use Service Builder. Here's how to enable the scanner:

1. Create the configuration file `com.liferay.portal.osgi.debug.spring.extender.internal.configuration.Unavail`
2. In the configuration file, set the time interval (in seconds) between scans:

   ```
   unavailableComponentScanningInterval=5
   ```

3. Copy the file into `[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
```

If a component is unavailable, it warns you:

```
11:13:08,851 WARN  [Spring Extender Unavailable Component Scanner][UnavailableComponentScanner:173] Found unavailable component in bundle com.liferay
Component ComponentImpl[null com.liferay.portal.spring.extender.internal.context.ModuleApplicationContextRegistrator@1541eee] is unavailable due to r
```

Component unavailability, such as what's reported above, can occur when DS components and Service Builder components are published and used in the same module. Use separate modules to publish DS components and Service Builder components.

dm na Command

Dependency Manager's Gogo shell command `dm` lists all Service Builder components, their required services, and whether each required service is available.

To list unresolved components only execute this Gogo shell command:

```
dm na
```

The `na` option stands for "not available."

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 component is unresolved.

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.SingleDestinationMessageSende
```

Based on the message above, there's no bundle providing the service `com.liferay.portal.kernel.messaging.sende`

Now you can detect unresolved components, DS and DM components, automatically using scanners, manually using Gogo shell commands, and programmatically using a `ServiceProxyFactory`.

Related Topics

System Check

152.20 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, use `.config` files for their flexibility and ability to use type information. Since `.cfg` files lack type information, if you want to store anything but a `String`, you must use properties utility classes to cast `String`s to intended types (and you must carefully document properties that aren't `String`s). `.config` files eliminate this need by allowing type information. The articles below explain the file formats:

- Understanding System Configuration Files
- Configuration file (`.config`) syntax
- Properties file(`.cfg`) syntax

Naming Configuration Files

Before you create a configuration file, follow these steps to determine whether multiple instances of the component can be created or if the component is intended to be a singleton:

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.

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:

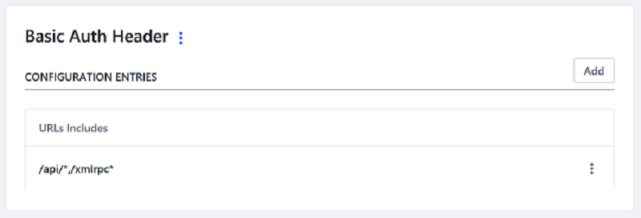


Figure 152.1: You can create multiple instances of components whose System Settings page has a *Configuration Entries* section.

```
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

152.21 Calling Non-OSGi Code that Uses OSGi Services

Liferay DXP's static service utilities (e.g., `UserServiceUtil`, `CompanyServiceUtil`, `GroupServiceUtil`, etc.) are examples of non-OSGi code that use OSGi services. Service Builder generates them for backwards compatibility purposes only. If you're tempted to call a `*ServiceUtil` class or your existing code calls one, access the `*Service` directly instead using one these alternatives:

- If your class is a Declarative Services component, use an `@Reference` annotation to access the `*Service` class.
- If your class isn't a Declarative Services component, use a `ServiceTracker` to access the `*Service` class.

You can check the state of Liferay DXP's services in the Gogo shell. The `scr:list` Gogo shell 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

Detecting Unresolved OSGi Components
Felix Gogo Shell
OSGi Basics For Liferay Development

152.22 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. But there is a way to avoid this situation: database connection retries.

1. Create a `portal-ext.properties` file in your Liferay Home folder.

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

152.23 Using OSGi Services from EXT Plugins

`ServiceTrackers` are the best way for Ext plugins to access OSGi services. They account for the possibility of OSGi services coming and going.

Related Topics

Detecting Unresolved OSGi Components
    Felix Gogo Shell
    OSGi Basics For Liferay Development

CHAPTER 153

DATA UPGRADES

The development process doesn't end when you first release your application. Through your own planning, feature requests, and bug reports, developers improve their applications on a regular basis.

Sometimes, those changes result in changes to the data structure and underlying database. When users upgrade, they need a process that transitions them to improved versions of your application. For this, you must create an upgrade process.

This section shows how to do that.

153.1 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`:

```
Liferay-Require-SchemaVersion: 1.1
```

Specifying the major and minor schema version only (format `major.minor`) gives your module flexibility to use any micro schema version. This lets you disregard new micro schema versions or upgrade to them when you want. You can also revert micro schema versions.

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.api", version: "2.0.3"
```

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. The upgrade framework lets you 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.

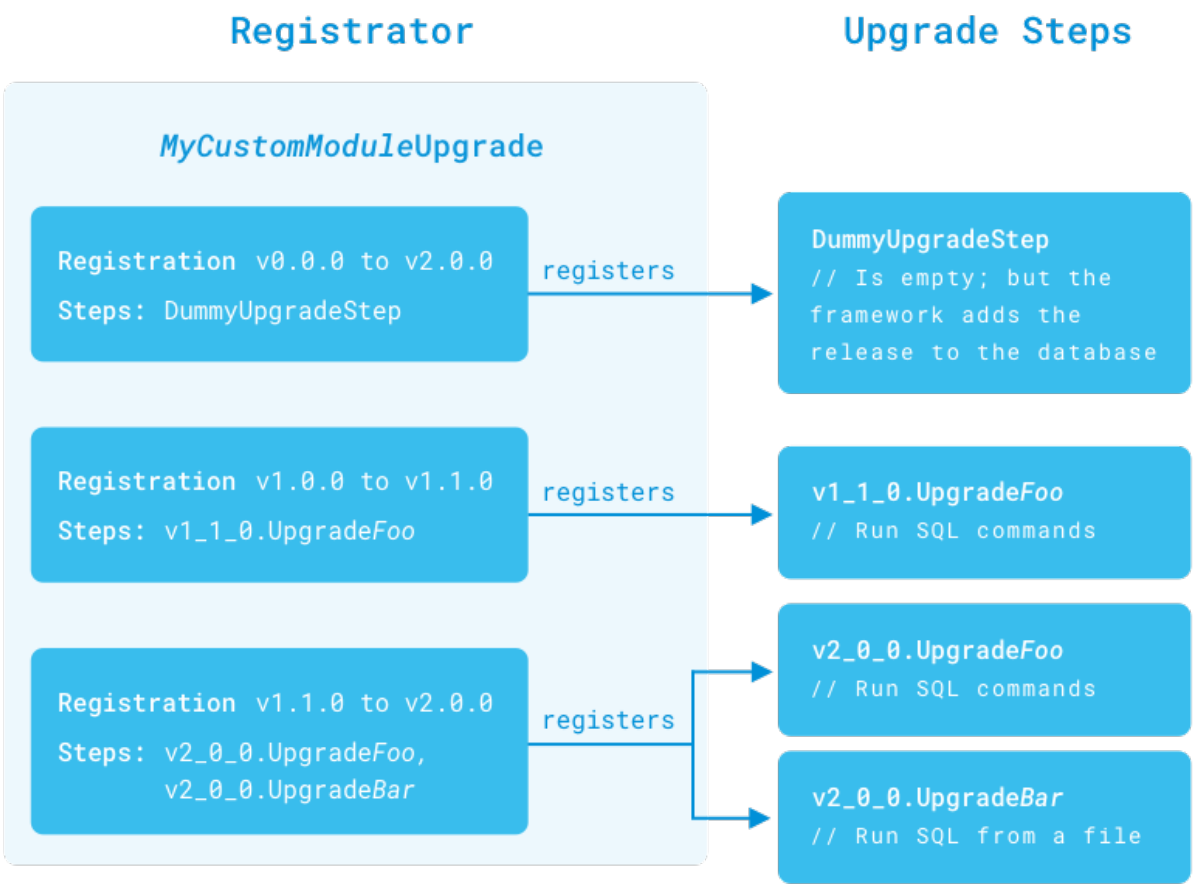


Figure 153.1: In a registrator class, the developer specifies a registration for each schema version upgrade. The upgrade steps handle the database updates.

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_` 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
Configurable Applications
Migrating Data Upgrade Processes to the New Framework for Modules

153.2 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 Liferay DXP's `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

153.3 Meaningful Schema Versioning

Data schema versions can be as arbitrary as you like; but they are most helpful when they provide meaning. Liferay's data schema version convention communicates a schema's compatibility with older versions of the software. It tells you whether a schema's changes maintain or break compatibility with existing software. For example, if a new data schema removes a field your software expects, the schema breaks compatibility. But if a new schema's changes are non-breaking (e.g., adds a new field), the schema is compatible and can be used with existing software. Since Liferay DXP 7.1, Liferay uses a meaningful schema version convention (similar to Semantic Versioning) to define new upgrade steps and support rollback of schema micro versions. The convention is optional but tracks data schema backwards compatibility.

Here's Liferay's schema version convention:

MAJOR.MINOR.MICRO

Each part means something:

MAJOR: Contains breaking schema/data changes that are incompatible with the latest version of existing code.

MINOR: Contains schema/data changes compatible with the latest version of existing code. The changes typically involve supporting new functionality.

MICRO: Contains schema/data changes that are compatible with the latest version of existing code.

Next are some concrete examples of micro, minor, and major changes.

Micro change examples

Here are common micro changes:

- Increasing `VARCHAR` field sizes.
- Modifying DB indexes.
- Modifying data values to adapt to current logic. These include backwards compatible data changes only. These changes commonly occur when data updates are missed for new functionalities.
- Converting a field from a String to a CLOB, as long as the field has few records and isn't used in `DISTINCT` or `GROUP BY` SQL clauses.

Minor change examples

Here are common minor changes:

- Adding a new DB field.
- Creating a new DB table.

Major change examples

Here are common major changes:

- Removing a DB field
- Removing a DB table.
- Altering a column name.
- Decreasing the size of a `VARCHAR` field.
- Converting a field from a String to a CLOB, where the field is has many records or is used in `DISTINCT` or `GROUP BY` SQL clauses.

Now you can ascribe meaningful versions to your module's data schemas.

153.4 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.

The Build Auto Upgrade feature is now in a 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.
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 154

Back-end Frameworks

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
- Portlet Providers
- Data Scopes
- Message Bus

These frameworks and more deliver smashing performances and are stars in their own right.

CHAPTER 155

PORTLET PROVIDERS

Some apps perform the same operations on different entity types. For example, the Asset Publisher lets users browse, add, preview, and view various entities as assets including documents, web content, blogs, and more. The entities vary, while the operations and surrounding business logic stay the same. Apps such as the Asset Publisher rely on the Portlet Providers framework to fetch portlets to operate on the entities. In this way, the framework lets you focus on entity operations and frees you from concern about portlets that carry out those operations. This tutorial shows you how to

- Create and register Portlet Providers
- Retrieve portlets from the Portlet Providers

155.1 Creating PortletProviders

`PortletProviders` are Component classes associated with an entity type. They have methods that return portlet IDs and portlet URLs. Once you've registered a `PortletProvider`, you can invoke the `PortletProviderUtil` class to retrieve the portlet ID or portlet URL from the corresponding `PortletProvider`.

Examine the `WikiPortletProvider` class:

```
@Component(
    immediate = true,
    property = {
        "model.class.name=com.liferay.wiki.model.WikiPage",
        "service.ranking:Integer=100"
    },
    service = {EditPortletProvider.class, ViewPortletProvider.class}
)
public class WikiPortletProvider
    extends BasePortletProvider
    implements EditPortletProvider, ViewPortletProvider {

    @Override
    public String getPortletName() {
        return WikiPortletKeys.WIKI;
    }
```

```
}
```

`WikiPortletProvider` extends `BasePortletProvider`, inheriting its `getPortletURL` methods. It must, however, implement `PortletProvider`'s `getPortletName` method, which returns the portlet's name `WikiPortletKeys.WIKI`.

Note: If you're creating a `PortletProvider` for one of Liferay's portlets, make your `getPortletName` method returns the portlet name from that portlet's `*PortletKeys` class if it has such a class.

`WikiPortletProvider`'s `@Component` annotation specifies these elements and properties:

- `immediate = true` activates the component immediately upon installation.
- `"model.class.name=com.liferay.wiki.model.WikiPage"` specifies the entity type the portlet operates on.
- `"service.ranking:Integer=100"` sets the component's rank to `100`, prioritizing it above all `PortletProviders` that specify the same `model.class.name` value but have a lower rank.
- `service = {EditPortletProvider.class, ViewPortletProvider.class}` reflects the subinterface `PortletProvider` classes this class implements.

Here's how to create your own `PortletProvider`:

1. Create an OSGi module.

2. Create a `PortletProvider` class in your module. Use the recommended class naming convention:

   `[Entity] + [Action] + PortletProvider`

   Example:

   `LanguageEntryViewPortletProvider`

3. Extend `BasePortletProvider` if you want to use its `getPortletURL` method implementations.

4. Implement one or more `PortletProvider` subinterfaces that match your action(s):

   - `BrowsePortletProvider`
   - `EditPortletProvider`
   - `ManagePortletProvider`
   - `PreviewPortletProvider`
   - `ViewPortletProvider`

5. Make the class an OSGi Component by adding an annotation like this one:

```
@Component(
    immediate = true,
    property = {"model.class.name=CLASS_NAME"},
    service = {INTERFACE_1.class, ...}
)
```

The `immediate = true` element specifies that the component should be activated immediately upon installation.

Assign the property `model.class.name` class name of the entity the portlet operates on by replacing `CLASS_NAME` with your entity's fully qualified class name. Here's an example `model.class.name` property:

```
"model.class.name=com.liferay.wiki.model.WikiPage"
```

Assign the `service` element to the `PortletProvider` subinterface(s) you're implementing (e.g., `ViewPortletProvider.class`, `BrowsePortletProvider`). Replace `INTERFACE_1.class, ...` with a list of the subinterface(s) you're implementing.

6. If you're overriding an existing `PortletProvider`, outrank it with your own custom `PortletProvider` by specifying a `service.ranking:Integer` property with a higher integer ranking.

```
property= {"service.ranking:Integer=10"}
```

7. Implement the provider methods you want. Make sure you implement `PortletProvider`'s `getPortletName` method. If you didn't extend `BasePortletProvider`, implement `PortletProvider`'s `getPortletURL` methods too.

8. Deploy your module.

Now your `PortletProvider` is available to return the ID and URL of the portlet that provides the desired behaviors. Using `PortletProviderUtil` to fetch the portlet IDs and URLs is next.

155.2 Retrieving Portlets for Desired Behaviors

The `PortletProviderUtil` class facilitates fetching portlets to execute actions on entities. You can request the ID or URL of a portlet that performs the entity action you want.

The Portlet Provider framework's `PortletProvider.Action` Enums define these action types:

- `ADD`
- `BROWSE`
- `EDIT`
- `MANAGE`
- `PREVIEW`
- `VIEW`

The action type and entity type are key parameters in fetching a portlet's ID or URL.

Fetching a Portlet ID

The Portlet Provider framework's `PortletProviderUtil` class facilitates fetching an ID of a portlet for handling an entity operation. For example, this call gets the ID of a portlet for viewing Recycle Bin entries:

```
String portletId = PortletProviderUtil.getPortletId(
    "com.liferay.portlet.trash.model.TrashEntry",
    PortletProvider.Action.VIEW);
```

`PortletProvider.Action.VIEW` is the operation and `com.liferay.portlet.trash.model.TrashEntry` is the entity type.

Another example is how the Asset Publisher uses the Portlet Provider framework to add a previewed asset to a page—it adds the asset to a portlet and adds that portlet to the page. The Asset Publisher uses the `liferay-asset:asset_display` tag library tag whose `asset_display/preview.jsp` shows an *Add* button for adding the portlet. If the previewed asset is a Blogs entry, for example, the framework returns a blogs portlet ID or URL for adding the portlet to the current page. Here's the relevant code from the `asset_display/preview.jsp`:

```
Map<String, Object> data = new HashMap<String, Object>();

<!-- populate the data map -->

String portletId = PortletProviderUtil.getPortletId(assetEntry.getClassName(), PortletProvider.Action.ADD);

data.put("portlet-id", portletId);

<!-- add more to the data map -->
%>

<c:if test="<%= PortletPermissionUtil.contains(permissionChecker, layout, portletId, ActionKeys.ADD_TO_PAGE) %>">
    <aui:button cssClass="add-button-preview" data="<%= data %>" value="add" />
</c:if>
```

The code above invokes `PortletProviderUtil.getPortletId(assetEntry.getClassName(), PortletProvider.Action.ADD)` to get the ID of a portlet that adds and displays the asset of the underlying entity class.

The JSP puts the portlet ID into the `data` map.

```
data.put("portlet-id", portletId);
```

Then it passes the `data` map to a new *Add* button that adds the portlet to the page.

```
<aui:button cssClass="add-button-preview" data="<%= data %>" value="add" />
```

Fetching a portlet URL is just as easy.

Fetching a Portlet URL

`PortletProviderUtil`'s `getPortletURL` methods return a `javax.portlet.PortletURL` based on an `HttpServletRequest` or `PortletRequest`. They also let you specify a `Group`.

For example, when the Asset Publisher is configured in Manual mode, the user can use an Asset Browser to select asset entries. The `asset-publisher-web` module's `configuration/asset_entries.jsp` file uses `PortletProviderUtil`'s `getPortletURL` method (at the end of the code below) to generate a corresponding Asset Browser URL.

```
List<AssetRendererFactory<?>> assetRendererFactories =
    ListUtil.sort(
        AssetRendererFactoryRegistryUtil.getAssetRendererFactories(
            company.getCompanyId()),
            new AssetRendererFactoryTypeNameComparator(locale));

for (AssetRendererFactory<?> curRendererFactory : assetRendererFactories) {
    long curGroupId = groupId;

    if (!curRendererFactory.isSelectable()) {
        continue;
    }

    PortletURL assetBrowserURL = PortletProviderUtil.getPortletURL(
        request, curRendererFactory.getClassName(),
        PortletProvider.Action.BROWSE);
```

Now you can unleash an arsenal of `PortletProviders` to use in your apps!

155.3 Related Topics

Portlets

Embedding Portlets in Themes

Customizing Liferay Services

CHAPTER 156

DATA SCOPES

Apps can restrict their data to specific scopes. Scopes provide a context for the application's data.

Global: One data set throughout a portal instance.

Site: Separate data sets for each Site it's added to.

Page: Separate data sets for each page it's added to.

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 Widget Scope. To give your applications scope, you must manually add support for it. This tutorial shows you how.

156.1 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. The Service Builder Persistence and Business Logic with Service Builder tutorials show you how.

156.2 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, the Web Content Display Portlet's portlet class sets this component property:

```
@Component(
    immediate = true,
    property = {
        ...
        "com.liferay.portlet.scopeable=true",
        ...,
    },
    service = Portlet.class
)
public class JournalContentPortlet extends MVCPortlet {...
```

That's it! Next, you'll access your app's scope in your code.

156.3 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. Your app's scope is available in these ways:

1. Via the `scopeGroupId` variable that is injected in your JSPs that use the `<liferay-theme:defineObjects />` tag. This variable contains your app's current scope. For example, the Liferay Bookmarks app's `view.jsp` uses its `scopeGroupId` 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.getS
...
```

2. By calling the `getScopeGroupId()` method on the request's `ThemeDisplay` instance. This method returns your app's current scope. For example, the Liferay Blogs app's `EditEntryMVCActionCommand` class 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. The tutorial Understanding `ServiceContext` provides an example and details. If you know your app always needs the portal instance ID, use the `ServiceContext` object's `getCompanyId()` method.

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.

156.4 Accessing the Site Scope Across Apps

There may be times when you must access a different app's Site-scoped data from your app that is 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, exist only in the Site scope. The above techniques return the app's scope, which might not be the Site scope. 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.

156.5 Related Topics

Widget Scope
    Service Builder
    Service Builder Persistence
    Business Logic with Service Builder

Chapter 157

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.

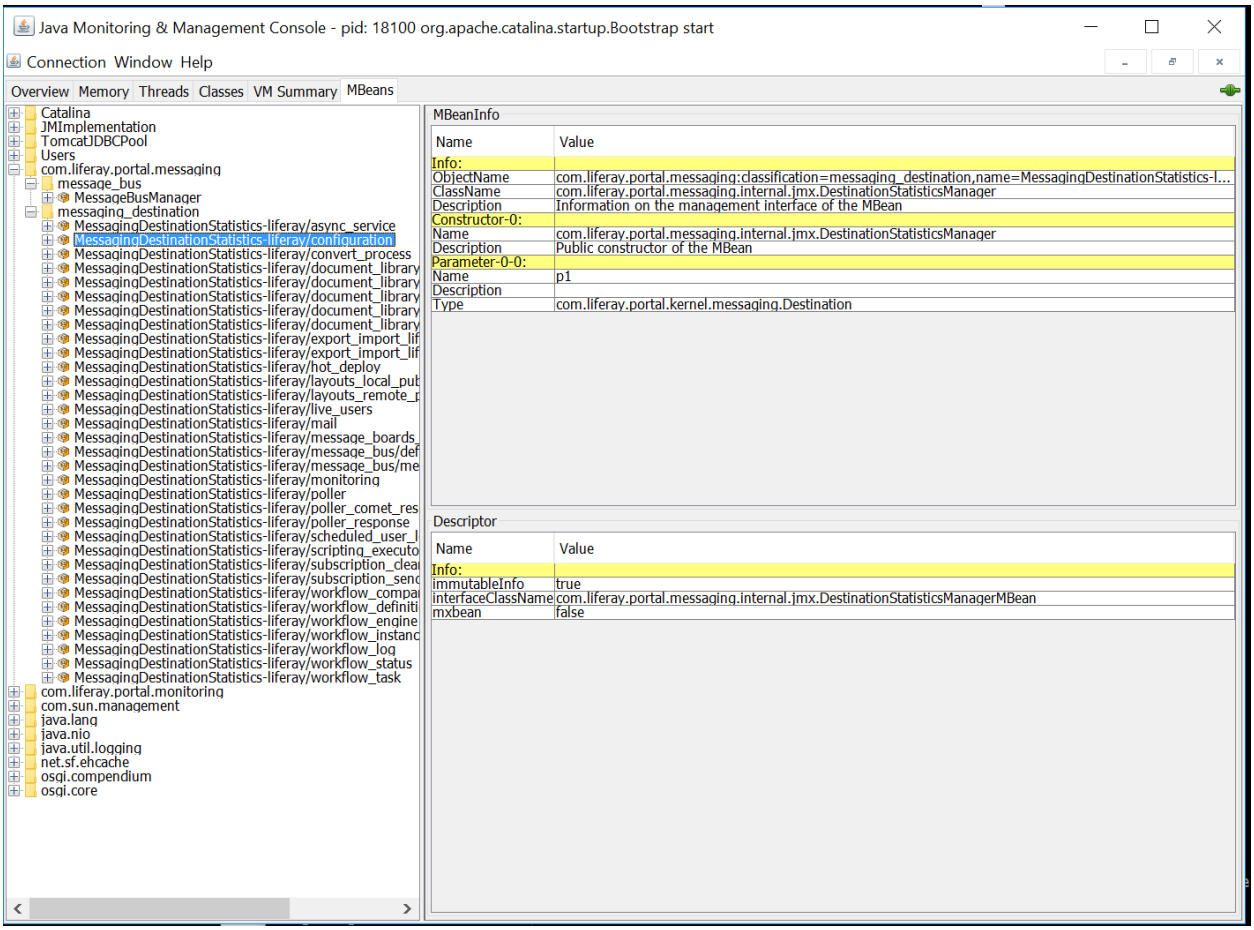


Figure 157.1: JConsole shows statistics on Message Bus messages sent, messages pending, and more.

157.1 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 the 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(
        destinationConfiguration);

     // 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 deactivating the example configurator, 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 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

157.2 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 DXP'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

157.3 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

Part II

Developer Reference

CHAPTER 158

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 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.

158.1 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 Zip 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.

App | Packages | **Announcements:** | com.liferay.announcements.constants | **Apio Architect:** | com.liferay.apio.architect.api | **Application List:** | com.liferay.application.list.api com.liferay.application.list.taglib | **Asset:** | com.liferay.asset.api com.liferay.asset.categories.navigation.api com.liferay.asset.category.property.api com.liferay.asset.display.api com.liferay.asset.display.page.api com.liferay.asset.display.page.item.selector.api com.liferay.asset.entry.rel.api com.liferay.asset.publisher.api com.liferay.asset.tag.stats.api com.liferay.asset.taglib com.liferay.asset.tags.api com.liferay.asset.tags.navigation.api com.liferay.asset.test.util | **Blogs:** | com.liferay.blogs.api com.liferay.blogs.demo.data.creator.api com.liferay.blogs.item.selector.api com.liferay.blogs.recent.bloggers.api com.liferay.blogs.test.util | **Calendar:** | com.liferay.calendar.api | **Captcha:** | com.liferay.captcha.api com.liferay.captcha.taglib | **Comment:** | com.liferay.comment.api com.liferay.comment.demo.data.creator.api com.liferay.comment.taglib | **Configuration Admin:** | com.liferay.configuration.admin.api | **Contacts:** | com.liferay.contacts.api | **Document Library:** | com.liferay.document.library.api com.liferay.document.library.content.api com.liferay.document.library.demo.data.creator.api com.liferay.document.library.file.rank.api com.liferay.document.library.repository.authorization.api com.liferay.document.library.repository.cmis.api com.liferay.document.library.repository.external.api com.liferay.document.library.sync.api com.liferay.document.library.test.util | **Dynamic Data Lists:** | com.liferay.dynamic.data.lists.api | **Dynamic Data Mapping:** | com.liferay.dynamic.data.mapping.api com.liferay.dynamic.data.mapping.taglib com.liferay.dynamic.data.mapping.test.util | **Export Import:** | com.liferay.exportimport.api com.liferay.exportimport.changeset.api com.liferay.exportimport.changeset.taglib com.liferay.exportimport.test.util | **Flags:** | com.liferay.flags.api com.liferay.flags.taglib | **Fragment:** | com.liferay.fragment.api com.liferay.fragment.demo.data.creator.api com.liferay.fragment.item.selector.api | **Friendly URL:** | com.liferay.friendly.url.api | **Frontend Editor:** | com.liferay.frontend.editor.api | **Frontend Image Editor:** | com.liferay.frontend.image.editor.capability | **Frontend JS:** | com.liferay.frontend.js.loader.modules.extender.npm | **HTML Preview:** | com.liferay.html.preview.api | **Invitation:** | com.liferay.invitation.invite.members.api | **Item Selector:** | com.liferay.item.selector.api com.liferay.item.selector.criteria.api com.liferay.item.selector.taglib | **Journal:** | com.liferay.journal.api com.liferay.journal.content.asset.addon.entry.api com.liferay.journal.demo.data.creator.api com.liferay.journal.item.selector.api com.liferay.journal.taglib com.liferay.journal.test.util | **Layout:** | com.liferay.layout.api com.liferay.layout.admin.api com.liferay.layout.item.selector.api com.liferay.layout.page.template.api com.liferay.layout.prototype.api com.liferay.layout.set.prototype.api com.liferay.layout.taglib | **Map:** | com.liferay.map.api com.liferay.map.taglib | **Mentions:** | com.liferay.mentions.api | **Message Boards:** | com.liferay.message.boards.api com.liferay.message.boards.demo.data.creator.api com.liferay.message.boards.test.util | **Mobile Device Rules:** | com.liferay.mobile.device.rules.api | **Organizations:** | com.liferay.organizations.api com.liferay.organizations.item.selector.api | **Password Policies Admin:** | com.liferay.password.policies.admin.constants | **Polls:** | com.liferay.polls.api | **Portal:** | com.liferay.portal.custom.jsp.bag.api com.liferay.portal.dao.orm.custom.sql.api com.liferay.por-

tal.instance.lifecycle.api com.liferay.portal.jmx.api com.liferay.portal.output.stream.container.api com.liferay.portal.spring.extender.api com.liferay.portal.upgrade.api | **Portal Background Task:** | com.liferay.portal.background.task.api | **Portal Cache:** | com.liferay.portal.cache.api com.liferay.portal.cache.ehcache.spi com.liferay.portal.cache.test.util | **Portal Configuration:** | com.liferay.portal.configuration.test.util com.liferay.portal.configuration.upgrade.api | **Portal Instances:** | com.liferay.portal.instances.service | **Portal Lock:** | com.liferay.portal.lock.api | **Portal Reports Engine:** | com.liferay.portal.reports.engine.api | **Portal Remote:** | com.liferay.portal.remote.soap.extender | **Portal Rules:** | com.liferay.portal.rules.engine | **Portal Scripting:** | com.liferay.portal.scripting | **Portal Search:** | com.liferay.portal.search.api com.liferay.portal.search.engine.adapter.api com.liferay.portal.search.spi com.liferay.portal.search.test.util com.liferay.portal.search.web.api | **Portal Security:** | com.liferay.portal.security.exportimport.api com.liferay.portal.security.ldap.api com.liferay.portal.security.permission.api com.liferay.portal.security.service.access.policy.api com.liferay.portal.security.service.access.quota.api | **Portal Security Audit:** | com.liferay.portal.security.audit.api com.liferay.portal.security.audit.event.generators.api com.liferay.portal.security.audit.storage.api | **Portal Security SSO:** | com.liferay.portal.security.sso.cas.api com.liferay.portal.security.sso.facebook.connect.api com.liferay.portal.security.sso.google.api com.liferay.portal.security.sso.ntlm.api com.liferay.portal.security.sso.openid.api com.liferay.portal.security.sso.openid.connect.api com.liferay.portal.security.sso.opensso.api com.liferay.portal.security.sso.token.api | **Portal Settings:** | com.liferay.portal.settings.api | **Portal Template:** | com.liferay.portal.template.soy.api | **Portal URL Builder:** | com.liferay.portal.url.builder | **Portal Workflow:** | com.liferay.portal.workflow.api com.liferay.portal.workflow.kaleo.api com.liferay.portal.workflow.kaleo.definition.api com.liferay.portal.workflow.kaleo.runtime.api | **Portlet Display Template:** | com.liferay.portlet.display.template.api | **Product Navigation:** | com.liferay.product.navigation.control.menu.api com.liferay.product.navigation.product.menu.api com.liferay.product.navigation.simulation.api com.liferay.product.navigation.taglib | **Ratings:** | com.liferay.ratings.page.ratings.constants | **Reading Time:** | com.liferay.reading.time.api com.liferay.reading.time.taglib | **Roles:** | com.liferay.roles.admin.api com.liferay.roles.admin.demo.data.creator.api com.liferay.roles.item.selector.api | **RSS:** | com.liferay.rss.api com.liferay.rss.taglib | **Site:** | com.liferay.site.api com.liferay.site.demo.data.creator.api com.liferay.site.item.selector.api com.liferay.site.taglib | **Social:** | com.liferay.social.activities.api com.liferay.social.activities.taglib com.liferay.social.activity.api com.liferay.social.activity.test.util com.liferay.social.bookmarks.api com.liferay.social.bookmarks.taglib com.liferay.social.user.statistics.api | **Staging:** | com.liferay.staging.api com.liferay.staging.taglib | **Subscription:** | com.liferay.subscription.api | **Text Localizer:** | com.liferay.text.localizer.address.api com.liferay.text.localizer.taglib | **Trash:** | com.liferay.trash.api com.liferay.trash.taglib com.liferay.trash.test.util | **Upload:** | com.liferay.upload | **User Associated Data:** | com.liferay.user.associated.data.api com.liferay.user.associated.data.test.util | **User Groups Admin:** | com.liferay.user.groups.admin.api com.liferay.user.groups.admin.item.selector.api | **Users Admin:** | com.liferay.users.admin.api com.liferay.users.admin.demo.data.creator.api com.liferay.users.admin.item.selector.api com.liferay.users.admin.test.util | **Wiki:** | com.liferay.wiki.api | **XStream:** | com.liferay.xstream.configurator |

For help finding module attributes and configuring dependencies, see Configuring Dependencies.

158.2 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)
- JSTL core
- aui
- liferay-portlet
- portlet
- portlet_1_0
- liferay-security
- liferay-theme
- liferay-ui
- liferay-util

Liferay DXP App Taglibs

Adaptive Media:
- liferay-application-list (Opens New Window)

Application List:
- liferay-application-list (Opens New Window)

Assets:
- liferay-asset (Opens New Window)
- liferay-trash (Opens New Window)

Forms:
- liferay-ddm (Opens New Window)

Foundation:
- liferay-map (Opens New Window)
- liferay-frontend (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.

158.3 JavaScript and CSS

Lexicon: 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.

Clay: The web implementation of Liferay's Lexicon Experience Language.

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 Liferay DXP 7.1.

CHAPTER 159

BACK-END

As you create portlets and customizations, it helps to reference backend APIs, descriptors, and third-party artifacts. These articles provide such references.

159.1 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-ee-sp20 to classes in liferay-dxp-src-7.1.10-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
ArticleResourcePKComparator
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
AutoDeleteFileInputStream
Old: com.liferay.portal.kernel.io New: com.liferay.petra.io
com.liferay.petra.io
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
BannedUserException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
BaseCmisRepository
Old: com.liferay.portal.kernel.repository.cmis New: com.liferay.document.library.repository.cmis
com.liferay.document.library.repository.cmis.api
BaseCmisSearchQueryBuilder
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
BaseDDLExporter
Old: com.liferay.portlet.dynamicdatalists.util New: com.liferay.dynamic.data.lists.internal.exporter
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.api
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
BlockingPortalCache
Old: com.liferay.portal.kernel.cache New: com.liferay.portal.cache
com.liferay.portal.cache.api
BlogsEntry
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay.blogs.api

BlogsEntryFinder
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay.blogs.api
BlogsEntryLocalService
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api
BlogsEntryLocalServiceUtil
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api
BlogsEntryLocalServiceWrapper
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api
BlogsEntryModel
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay.blogs.api
BlogsEntryPersistence
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay.blogs.api
BlogsEntryService
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api
BlogsEntryServiceUtil
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api
BlogsEntryServiceWrapper
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api
BlogsEntrySoap
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay.blogs.api
BlogsEntryUtil
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay.blogs.api
BlogsEntryWrapper
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay.blogs.api
BlogsStatsUser
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay.blogs.api
BlogsStatsUserFinder
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay.blogs.api
BlogsStatsUserLocalService
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api
BlogsStatsUserLocalServiceUtil
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api

BlogsStatsUserLocalServiceWrapper
Old: com.liferay.portlet.blogs.service New: com.liferay.blogs.service
com.liferay.blogs.api
BlogsStatsUserModel
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay.blogs.api
BlogsStatsUserPersistence
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay.blogs.api
BlogsStatsUserSoap
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay.blogs.api
BlogsStatsUserUtil
Old: com.liferay.portlet.blogs.service.persistence New: com.liferay.blogs.service.persistence
com.liferay.blogs.api
BlogsStatsUserWrapper
Old: com.liferay.portlet.blogs.model New: com.liferay.blogs.model
com.liferay.blogs.api
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
CMISBetweenExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISConjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISContainsExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISContainsNotExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISContainsValueExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISCriterion
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISDisjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISFullTextConjunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISInFolderExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api

CMISInTreeExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISJunction
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISNotExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISParameterValueUtil
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISRepositoryHandler
Old: com.liferay.portal.kernel.repository.cmis New: com.liferay.document.library.repository.cmis
com.liferay.document.library.repository.cmis.api
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.api
CMISSimpleExpression
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CMISSimpleExpressionOperator
Old: com.liferay.portal.kernel.repository.cmis.search New: com.liferay.document.library.repository.cmis.search
com.liferay.document.library.repository.cmis.api
CharPool
Old: com.liferay.portal.kernel.util New: com.liferay.petra.string
com.liferay.petra.string
CharsetDecoderUtil
Old: com.liferay.portal.kernel.nio.charset New: com.liferay.petra.nio
com.liferay.petra.nio
CharsetEncoderUtil
Old: com.liferay.portal.kernel.nio.charset New: com.liferay.petra.nio
com.liferay.petra.nio
ClassLoaderPool
Old: com.liferay.portal.kernel.util New: com.liferay.petra.lang
com.liferay.petra.lang

ClassResolverUtil
Old: com.liferay.portal.kernel.util New: com.liferay.petra.lang
com.liferay.petra.lang
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.internal
com.liferay.portal.compound.session.id
Condition
Old: com.liferay.portlet.dynamicdatamapping.storage.query New: com.liferay.adaptive.media.image.media.query
com.liferay.adaptive.media.image.api
ContactConverterKeys
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap
com.liferay.portal.security.ldap.api
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
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

DLContent

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay.document.library.content.api

DLContentDataBlobModel

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay.document.library.content.api

DLContentLocalService

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.content.service

com.liferay.document.library.content.api

DLContentLocalServiceUtil

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.content.service

com.liferay.document.library.content.api

DLContentLocalServiceWrapper

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.content.service

com.liferay.document.library.content.api

DLContentModel

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay.document.library.content.api

DLContentPersistence

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.content.service.persistence

com.liferay.document.library.content.api

DLContentSoap

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay.document.library.content.api

DLContentUtil

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.content.service.persistence

com.liferay.document.library.content.api

DLContentVersionComparator

Old: com.liferay.portlet.documentlibrary.util.comparator New: com.liferay.document.library.content.service.util.comparator

com.liferay.document.library.content.service

DLContentWrapper

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.content.model

com.liferay.document.library.content.api

DLFileRank

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.file.rank.model

com.liferay.document.library.file.rank.api

DLFileRankFinder

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.file.rank.service.persistence

com.liferay.document.library.file.rank.api

DLFileRankLocalService

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.file.rank.service

com.liferay.document.library.file.rank.api

DLFileRankLocalServiceUtil

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.file.rank.service

com.liferay.document.library.file.rank.api

DLFileRankLocalServiceWrapper

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.file.rank.service

com.liferay.document.library.file.rank.api

DLFileRankModel

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.file.rank.model

com.liferay.document.library.file.rank.api

DLFileRankPersistence

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.file.rank.service.persistence

com.liferay.document.library.file.rank.api

DLFileRankSoap

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.file.rank.model

com.liferay.document.library.file.rank.api

DLFileRankUtil

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.file.rank.service.persistence

com.liferay.document.library.file.rank.api

DLFileRankWrapper

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.file.rank.model

com.liferay.document.library.file.rank.api

DLSyncConstants

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.constants

com.liferay.document.library.sync.api

DLSyncEvent

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.model

com.liferay.document.library.sync.api

DLSyncEventLocalService

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.sync.service

com.liferay.document.library.sync.api

DLSyncEventLocalServiceUtil

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.sync.service

com.liferay.document.library.sync.api

DLSyncEventLocalServiceWrapper

Old: com.liferay.portlet.documentlibrary.service New: com.liferay.document.library.sync.service

com.liferay.document.library.sync.api

DLSyncEventModel

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.model

com.liferay.document.library.sync.api

DLSyncEventPersistence

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.sync.service.persistence

com.liferay.document.library.sync.api

DLSyncEventSoap

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.model

com.liferay.document.library.sync.api

DLSyncEventUtil

Old: com.liferay.portlet.documentlibrary.service.persistence New: com.liferay.document.library.sync.service.persistence

com.liferay.document.library.sync.api

DLSyncEventWrapper

Old: com.liferay.portlet.documentlibrary.model New: com.liferay.document.library.sync.model

com.liferay.document.library.sync.api

Database

Old: com.liferay.portal.kernel.util New: com.liferay.portal.tools.db.upgrade.client

com.liferay.portal.tools.db.upgrade.client

DefaultAttributesTransformer

Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.internal

com.liferay.portal.security.ldap.impl

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
DeleteFileFinalizeAction
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay.petra.memory
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.api
DummyDirContext
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.dummy
com.liferay.portal.security.ldap.api
DummyFinalizeAction
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay.petra.memory
DuplicateArticleIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
DuplicateFeedIdException
Old: com.liferay.portlet.journal New: com.liferay.journal.exception
com.liferay.journal.api
DuplicateLDAPServerNameException
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap
com.liferay.portal.security.ldap.api
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
EntryDisplayDateComparator
Old: com.liferay.portlet.blogs.util.comparator New: com.liferay.blogs.util.comparator
com.liferay.blogs.api
EntryModifiedDateComparator

Old: com.liferay.portlet.bookmarks.util.comparator New: com.liferay.bookmarks.util.comparator

com.liferay.bookmarks.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

EntrySmallImageNameException

Old: com.liferay.portlet.blogs New: com.liferay.blogs.exception

com.liferay.blogs.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

EqualityWeakReference

Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory

com.liferay.petra.memory

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
FinalizeAction
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay.petra.memory
FinalizeManager
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay.petra.memory
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.internal.messaging
com.liferay.flags.service
GroupConverterKeys
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap
com.liferay.portal.security.ldap.api
ImportFilesException
Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception
com.liferay.wiki.api
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

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.api
LDAPGroup
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay.portal.security.ldap.api
LDAPServerNameException
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap
com.liferay.portal.security.ldap.api
LDAPToPortalConverter
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay.portal.security.ldap.api
LDAPUser
Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport
com.liferay.portal.security.ldap.api
LDAPUtil
Old: com.liferay.portal.kernel.ldap New: com.liferay.portal.security.ldap.util
com.liferay.portal.security.ldap.api
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
LockedThreadException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
MBBan
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api

MBBanLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBBanLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBBanLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBBanModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBBanPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBBanService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBBanServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBBanServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBBanSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBBanUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBBanWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBCategory
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBCategoryConstants
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.constants
com.liferay.message.boards.api
MBCategoryDisplay
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.web.internal.display
com.liferay.message.boards.web
MBCategoryFinder
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence

com.liferay.message.boards.api
MBCategoryLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBCategoryLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBCategoryLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBCategoryModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBCategoryPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBCategoryService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBCategoryServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBCategoryServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBCategorySoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBCategoryUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBCategoryWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBDiscussion
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBDiscussionLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBDiscussionLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBDiscussionLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api

MBDiscussionModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBDiscussionPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBDiscussionSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBDiscussionUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBDiscussionWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBMailingList
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBMailingListLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMailingListLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMailingListLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMailingListModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBMailingListPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBMailingListSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBMailingListUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBMailingListWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBMessage
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model

com.liferay.message.boards.api
MBMessageConstants
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.constants
com.liferay.message.boards.api
MBMessageDisplay
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBMessageFinder
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBMessageLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMessageLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMessageLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMessageModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBMessagePersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBMessageService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMessageServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMessageServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBMessageSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBMessageUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBMessageWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBStatsUser
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model

com.liferay.message.boards.api
MBStatsUserLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBStatsUserLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBStatsUserLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBStatsUserModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBStatsUserPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBStatsUserSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBStatsUserUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBStatsUserWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBThread
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBThreadConstants
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.constants
com.liferay.message.boards.api
MBThreadFinder
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBThreadFlag
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBThreadFlagLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBThreadFlagLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBThreadFlagLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service

com.liferay.message.boards.api
MBThreadFlagModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBThreadFlagPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBThreadFlagSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBThreadFlagUtil
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBThreadFlagWrapper
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBThreadLocalService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBThreadLocalServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBThreadLocalServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBThreadModel
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBThreadPersistence
Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence
com.liferay.message.boards.api
MBThreadService
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBThreadServiceUtil
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBThreadServiceWrapper
Old: com.liferay.portlet.messageboards.service New: com.liferay.message.boards.service
com.liferay.message.boards.api
MBThreadSoap
Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model
com.liferay.message.boards.api
MBThreadUtil

Old: com.liferay.portlet.messageboards.service.persistence New: com.liferay.message.boards.service.persistence

com.liferay.message.boards.api

MBThreadWrapper

Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model

com.liferay.message.boards.api

MBTreeWalker

Old: com.liferay.portlet.messageboards.model New: com.liferay.message.boards.model

com.liferay.message.boards.api

MBeanRegistry

Old: com.liferay.portal.kernel.jmx New: com.liferay.portal.jmx

com.liferay.portal.jmx.api

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.web.internal.security.permission.resource
com.liferay.mobile.device.rules.web
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.web.internal.security.permission.resource
com.liferay.mobile.device.rules.web
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.web.internal.security.permission.resource
com.liferay.mobile.device.rules.web
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
MailingListEmailAddressException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
MailingListInServerNameException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
MailingListInUserNameException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
MailingListOutEmailAddressException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
MailingListOutServerNameException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
MailingListOutUserNameException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
MemoryReportDesignRetriever
Old: com.liferay.portal.kernel.bi.reporting New: com.liferay.portal.reports.engine
com.liferay.portal.reports.engine.api
MessageBodyException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.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
MessageCreateDateComparator
Old: com.liferay.portlet.messageboards.util.comparator New: com.liferay.message.boards.util.comparator
com.liferay.message.boards.api
MessageSubjectException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
MessageThreadComparator

Old: com.liferay.portlet.messageboards.util.comparator New: com.liferay.message.boards.util.comparator

com.liferay.message.boards.api

Modifications

Old: com.liferay.portal.security.ldap New: com.liferay.portal.security.ldap.exportimport

com.liferay.portal.security.ldap.api

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

NoSuchBanException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay.message.boards.api

NoSuchChoiceException

Old: com.liferay.portlet.polls New: com.liferay.polls.exception

com.liferay.polls.api

NoSuchContentException

Old: com.liferay.portlet.dynamicdatamapping New: com.liferay.dynamic.data.mapping.exception

com.liferay.dynamic.data.mapping.api

NoSuchContentSearchException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception

com.liferay.journal.api

NoSuchDiscussionException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay.message.boards.api

NoSuchFeedException

Old: com.liferay.portlet.journal New: com.liferay.journal.exception

com.liferay.journal.api

NoSuchFileRankException

Old: com.liferay.portlet.documentlibrary New: com.liferay.document.library.file.rank.exception

com.liferay.document.library.file.rank.api

NoSuchMailingListException

Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception

com.liferay.message.boards.api

NoSuchNodeException

Old: com.liferay.portlet.wiki New: com.liferay.wiki.exception

com.liferay.wiki.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
NoSuchStatsUserException
Old: com.liferay.portlet.blogs New: com.liferay.blogs.exception
com.liferay.blogs.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
NoSuchThreadException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
NoSuchThreadFlagException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.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
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.api
PageTitleComparator
Old: com.liferay.portlet.wiki.util.comparator New: com.liferay.wiki.util.comparator
com.liferay.wiki.api
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.api
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
PoolAction
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay.petra.memory
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.api
PortletDisplayTemplate
Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.portlet.display.template
com.liferay.portlet.display.template.api
PortletDisplayTemplateConstants
Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.portlet.display.template.constants
com.liferay.portlet.display.template.api

PortletDisplayTemplateUtil
Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.roles.admin.web.internal.util
com.liferay.roles.admin.web
PortletDisplayTemplateUtil
Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.roles.admin.web.internal.util
com.liferay.roles.admin.web
PortletDisplayTemplateUtil
Old: com.liferay.portlet.portletdisplaytemplate.util New: com.liferay.roles.admin.web.internal.util
com.liferay.roles.admin.web
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.messaging
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
RequiredMessageException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.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
SearchUtil
Old: com.liferay.portal.kernel.search.util New: com.liferay.portal.search.web.internal.util
com.liferay.portal.search.web
ServletContextReportDesignRetriever
Old: com.liferay.portal.kernel.bi.reporting.servlet New: com.liferay.portal.reports.engine.servlet
com.liferay.portal.reports.engine.api

SoftReferencePool
Old: com.liferay.portal.kernel.memory New: com.liferay.petra.memory
com.liferay.petra.memory
SortFactoryImpl
Old: com.liferay.portal.kernel.search New: com.liferay.portal.search.internal
com.liferay.portal.search
SplitThreadException
Old: com.liferay.portlet.messageboards New: com.liferay.message.boards.exception
com.liferay.message.boards.api
Statistics
Old: com.liferay.portal.kernel.monitoring.statistics New: com.liferay.portal.monitoring.internal.statistics
com.liferay.portal.monitoring
StatsUserLastPostDateComparator
Old: com.liferay.portlet.blogs.util.comparator New: com.liferay.blogs.util.comparator
com.liferay.blogs.api
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.internal.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
ThreadLastPostDateComparator
Old: com.liferay.portlet.messageboards.util.comparator New: com.liferay.message.boards.util.comparator
com.liferay.message.boards.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.api
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

Chapter 160

Front-End

Front-end development involves multiple frameworks and tools. Keeping track of all the moving pieces in your project can be a daunting task. This section of reference docs provides the following helpful information for front-end development:

- Understanding the liferay-npm-bundler
- The CKEditor plugins available for use in your custom AlloyEditor configurations.
- AlloyEditor button reference guide
- Fully qualified portlet IDs
- FreeMarker taglib macros
- Setting up your npm environment
- Liferay JS Generator

CHAPTER 161

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).

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.

This section of reference docs covers how Portal supports npm-based portlet projects with the liferay-npm-bundler.

Note: You can also find information for the liferay-npm-bundler in the project's Wiki.

161.1 How the Liferay npm Bundler Works Internally

The liferay-npm-bundler takes a 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 reference.

The liferay-npm-bundler uses the process below to create 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 the project:
    a. Run the source files, specified in the `.npmbundlerrc` configuration, through the rules.
    b. Pre-process the project's package with any configured plugins.
    c. Run Babel with configured plugins for each `.js` file inside the project.
    d. Post-process the project package with any configured plugins.

4. 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 a plain *bundle-name$package@version* format, rather than the standard node_modules tree format. To determine what is copied, the bundler invokes a plugin to filter the package file list.

    b. Run rules on the package files.

    c. Pre-process the npm package with any configured plugins.

    d. Run Babel with configured plugins for each `.js` file inside the npm package.

    e. Post-process the npm 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).

Note: that the pre, post, and Babel phases were designed for the old mode of operation (See the Migrating Your Project to Use the New Mode for more information) and they will gradually be replaced with rules for the new mode.

161.2 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 article explains the `.npmbundlerrc` file's structure. See the default preset reference to learn how the default preset configures the liferay-npm-bundler. See the Creating JavaScript Portlets with JavaScript Tooling tutorial to learn how to use the liferay-npm-bundler to create JavaScript portlets.

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.

Standard Configuration Options

Below are the standard configuration options for the `.npmbundlerrc` file:

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.

dump-report: Sets whether to generate a debugging report. If `true`, a `liferay-npm-bundler-report.html` file is generated in the project directory with information such as what the liferay-npm-bundler is doing with each package. Note that you can also pass this as the build flag `$ liferay-npm-bundler --dump-report` or `$ liferay-npm-bundler -r`. The default value is `false`.

no-tracking: whether to send usage analytics to our servers. Note that you can also pass this as a build flag with the CLI argument `$ liferay-npm-bundler --no-tracking`, or by creating a marker file called `.liferay-npm-bundler-no-tracking` in the project's root folder or any of its ancestors, or by setting the environment variable `LIFERAY_NPM_BUNDLER_NO_TRACKING=''`. The default value is `false`.

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. Note that the dependency npm packages are placed in a `node_modules` folder inside the build folder. Note if `create-jar` is set, the default output folder is `build`.

preset: specifies the `liferay-npm-bundler` preset to use as a base configuration. Note that if a `.npmbundlerrc` file is not provided, the default `liferay-npm-bundler-preset-standard` preset is used.

verbose: Sets whether to output detailed information about what the tool is doing to the console. The default value is `false`.

Package Processing Options

"/": plugins' configuration for the project's package.

"": plugins' configuration for dependency packages.

(asterisk): Defines the default plugin configuration for all npm packages. It contains four values identified by a corresponding key. Keys `copy-plugins`, `plugins` and `post-plugins` identify arrays of `liferay-npm-bundler` plugins to apply in the copy, pre and post process steps. Key `.babelrc` identifies an object specifying the configuration to use in the Babel step and has the same structure of a standard `.babelrc` file.

exclude: defines glob expressions of files to exclude from bundling from all or specific packages. Each list is an array identified by one of the following keys: * (any package), `{package name}` (any version of the package), or `{package name}@{version}` (a specific version of a package). Below is an example configuration:

```
{
  "exclude": {
    "*": ["__tests__/**/*"],
    "is-object": ["test/**/*"],
    "is-array@1.0.1": ["test/**/*", "Makefile"]
  }
}
```

ignore: skips processing the specified JavaScript files with Babel for the project. An example configuration is shown below:

```
{
  "ignore": ["lib/legacy/**/*.js"]
}
```

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).

max-parallel-files: Defines the maximum number of files to process in parallel to avoid EMFILE errors (especially on Windows). The default value is `128`.

packages: defines plugin configuration for npm packages, per package.

process-serially: removed since v2.7.0. Replaced with `max-parallel-files`.

`rules`: defines rules to apply to the projects source files with the loader. Rules must have a `use` array property, which defines the loader to use, which may be specified by just a package name or an object with `loader` and `options` properties if applicable, and one or more of the properties below:

- `test`: defines a regular expression to filter files in the `sources` folders to determine whether to apply rules to them. The project-relative path of each eligible file is compared against the regular expression and files that match are processed by the loaders.
- `exclude`: refines the `test` expression by specifying files to exclude.
- `include`: refines the `test` expression by specifying files to include.

An example configuration is shown below:

```
{
  "rules": [
    {
      "test": "\\.js$",
      "exclude": "node_modules",
      "use": [
        {
          "loader": "babel-loader",
          "options": {
            "presets": ["env", "react"]
```

```
        }
      }
    ]
  },
  {
    "test": "\\.css$",
    "use": ["style-loader"]
  },
  {
    "test": "\\.json$",
    "use": ["json-loader"]
  }
 ]
}
```

`sources`: defines the folders in the project that contain the source files to apply rules to. Folders can be nested (e.g. `/src/main/resources/`) and must be written using POSIX path separators (i.e. use / instead of \ on Win32 systems). Note that rules are automatically applied to package dependency files of the project.

An example configuration is shown below:

```
{
  "sources": ["src", "assets"]
}
```

OSGi Bundle Creation Options

Since version 2.2.0, the liferay-npm-bundler can create portlet OSGi bundles for you. See the Creating and Bundling JavaScript Portlets with JavaScript Tooling tutorial for complete instructions. The configuration options for OSGi bundle creation are shown below:

- **create-jar**: Creates an OSGi bundle when set to a truthy value. When set to `true`, all sub-options take default values. When an object is passed, as shown in the example above, each sub-option can be configured individually. Note that you can also pass this as a build flag: `$ liferay-npm-bundler --create-` or `$ liferay-npm-bundler -j`. The default value is `false`.

  { “create-jar”: true }

- **create-jar.auto-deploy-portlet**: **Note** that this option is deprecated. Use the `create-jar.features.js-extender` option instead.

- **create-jar.features.configuration**: specifies the file describing the system (OSGi) and widget instance (portlet preferences, as defined in the Portlet spec) configuration to use. (see Configuring System Settings and Instance Settings for Your JavaScript Portlets for more information on the required settings configuration). The default value is `features/configuration.json` if that file exists, otherwise the default is `undefined`.

  { “create-jar”: { “features”: { “configuration”: “features/configuration.json” } } }

- **create-jar.output-dir:** specifies where to place the final JAR

  { “create-jar”: { “features”: { “configuration”: “features/configuration.json” } } }

- **create-jar.features.js-extender:** controls whether to process the OSGi bundle with the JS Portlet Extender CE App

DXP App. You can also specify the minimum required version of the Extender to use for the bundle. This can be useful if you want to use advanced features in your bundle, but you want it to be deployable in older versions of the Extender. Pass the string "any" to let the bundle deploy in any version of the Extender. If true, the liferay-npm-bundler automatically determines the minimum version of the Extender required for the features used in the bundle. the default value is true. An example configuration is shown below:

```
{
  "create-jar": {
    "features": {
      "js-extender": "1.1.0"
    }
  }
}
```

- **create-jar.features.web-context:** specifies the context path to use for publishing bundle's static resources. The default value is /{project name}-{project version}.

  { "create-jar": { "features": { "web-context": "/my-project" } } }

- **create-jar.features.localization:** specifies the L10N file to be used by the bundle (see the Creating JS Portlets with JS Tooling tutorial for more information on using localization in your portlet. The default value is features/localization/Language if a properties file with that base name exists, otherwise the default is undefined.

  { "create-jar": { "features": { "localization": "features/localization/Language" } } }

- **create-jar.features.settings:** specifies the JSON file describing the configuration structure (see the Creating JS Portlets with JS Tooling tutorial for more information on the required settings configuration). The default value is features/settings.json if that file exists, otherwise the default is undefined.

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 you know the structure of the .npmbundlerrc file!

161.3 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"
        ]
    }
}
```

The configuration above states that for all npm packages (*) the pre-process phase (plugins) must run the replace-browser-modules plugin. Setting this to post-plugins would run it during the post phase instead.

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: Replaces the server side files for modules listed under browser/unpkg/jsdelivr section of package.json with their browser counterparts.
- 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-normalize-requires: Normalize AMD `require()` calls.
- 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.
- babel-plugin-minify-dead-code-elimination: Inline bindings when possible. Tries to evaluate expressions and prunes unreachable as a result.
- babel-plugin-wrap-modules-amd: Wrap modules inside an AMD `define()` module.
- babel-plugin-name-amd-modules: Name AMD modules based on package name, version, and module path.
- 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-namespace-amd-define: Add a prefix to AMD `define()` calls (by default `Liferay.Loader.`).

Now you know the available configuration presets for `.npmbundlerrc` and how they work.

161.4 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 how the liferay-npm-bundler creates OSGi bundles for npm packages!

161.5 How the Liferay npm Bundler Publishes npm Packages

When you deploy an OSGi bundle with the specified structure, as explained in The Structure of OSGi Bundles Containing NPM Packages reference, 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 article 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.jsisarray@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.

161.6 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 article references the OSGi structure below as an example. You can learn more about this structure in The Structure of OSGi Bundles Containing NPM Packages reference.

- `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.

Now you have a better understanding of how liferay-npm-bundler formats JavaScript modules for AMD!

161.7 Understanding How Liferay AMD Loader Configuration is Exported

NOTE: This article 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_loader_modules` URL.

The OSGi bundle shown below is used for reference in this article:

- `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 reference, 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.

161.8 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.

Now that you understand what changed between version 1.x and 2.x of the liferay-npm-bundler, you can follow the steps in the Migrating a liferay-npm-bundler Project from 1.x to 2.x tutorial to migrate your 1.x projects to 2.x.

161.9 Understanding liferay-npm-bundler's Loaders

liferay-npm-bundler's mechanism is inspired by webpack. Like webpack, the liferay-npm-bundler processes files with a set of rules, which includes loaders that transform a project's source files before producing the final output.

Note: While webpack creates a single JS bundle file, liferay-npm-bundler targets an AMD loader, so webpack and liferay-npm-bundler loaders are not compatible.

Loaders are npm packages that export a function in their main module, which receives source files and returns modified files, and optionally new files, based on the loader's configuration. For example, the babel-loader receives ES6+ JavaScript files, runs Babel on them, and returns transpiled ES5 files along with a generated source map. You can use this simple pattern to create custom loaders. A few example loader functions are shown below:

- Pass JS files through Babel or TSC
- Convert CSS files into JS modules that dynamically inject the CSS into the HTML page

- Process CSS files with SASS
- Create tools that generate code based on IDL files

Loaders are configured via the project's `.npmbundlerrc` file. A loader's configuration is specified with two key options: `sources` (the folders that contain the sources files to process) and `rules` (the loaders, options (if applicable), and regular expressions that determine which files to process). See Understanding the `.npmbundlerrc`'s Structure for more information on the configuration requirements and options.

Loaders can be chained. Files are processed by the loaders in the order in which they are listed in the `use` property. The files are passed to the first loader, processed, sent to the next loader, and so on and so forth, until the files are processed by the rules. This lets you run complex processes, such as converting a SASS file into CSS with the sass-loader, and then convert it into a JavaScript module with the style-loader. Once the rules are applied, the liferay-npm-bundler continues with the pre, post, and babel phases of the bundler plugins.

161.10 Default liferay-npm-bundler Loaders

Several loaders are available for the liferay-npm-bundler by default. These loaders are listed below:

- `babel-loader`: processes source files with Babel. This avoids an extra build step before the bundler.
- `copy-loader`: copies source files (static assets) to the output folder.
- `css-loader`: converts a CSS file into a JavaScript module that's inserted into the DOM once it's loaded.
- `json-loader`: generates JavaScript modules that export the contents of a JSON file as an object. This lets you include JSON files with the `require()` call.
- `sass-loader`: runs `node-sass` or `sass` on source files. This lets you generate static CSS files. It can be chained before `style-loader`.
- `style-loader`: converts a CSS file into a JavaScript module that directly inserts the CSS contents into the DOM once it's loaded. This lets you include CSS files with a `require()` call.

See the liferay-js-toolkit loaders showcase for an example use case of the liferay-npm-bundler's loaders. If the default loaders don't meet your requirements, you can follow the instructions in Creating Custom Loaders for the Bundler to create your own loaders.

161.11 CKEditor Plugin Reference Guide

This reference guide provides a list of the default CKEditor plugins bundled with Liferay DXP's AlloyEditor. You can use these existing CKEditor plugins in your custom AlloyEditor configurations. Each plugin below links to its `plugin.js` file for reference, specifying the plugin's name and buttons if applicable:

- 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, but 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 to set, 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
```

161.12 AlloyEditor Button Reference Guide

This reference guide provides additional information that you may find helpful while creating new buttons for AlloyEditor. In this guide provides useful information on the following topics:

- Mixins

Mixins

When creating a new button for the Alloy Editor, several mixins are available that make it easy to provide additional functionality. The available mixins, along with a brief description and a link to their API docs, are listed below:

- ButtonActionStyle: provides applying style implementation for a button based on the `applyStyle` and `removeStyle` API of CKEDITOR

- ButtonCommandActive: provides an `isActive` method to determine if a context-aware command is currently in an active state.
- ButtonCommand: executes a command via CKEDITOR's API
- ButtonKeystroke: provides a `keystroke` prop that allows configuring a function of the instance to be invoked upon the keystroke activation. https://docs-old.ckeditor.com/ckeditor_api/symbols/CKEDITOR.dom.event.html#getKeystroke
- ButtonCfgProps: provides a style prop and some methods to apply the resulting style and checking if it is present in a given path or selection.
- ButtonStateClasses: decorates the domElement of a component with different CSS classes based on the current state of the element.
- ButtonStyle: provides a style prop and some methods to apply the resulting style and checking if it is present in a given path or selection.
- ToolbarButtons: provides a list of buttons which have to be displayed on the current toolbar depending on user preferences and given state.

161.13 Fully Qualified Portlet IDs

Below is a listing of the portlet IDs for the default portlets in Liferay DXP. You can use these IDs to embed portlets in your theme's sitemap.

Collaboration

Portlet	ID
Blogs	`com_liferay_blogs_web_portlet_BlogsPortlet`
Blogs Aggregator	`com_liferay_blogs_web_portlet_BlogsAgreggatorPortlet`
Calendar	`com_liferay_calendar_web_portlet_CalendarPortlet`
Dynamic Data Lists Display	`com_liferay_dynamic_data_lists_web_portlet_DDLDisplayPortlet`
Form	`com_liferay_dynamic_data_mapping_form_web_portlet_DDMFormPor`
Invite Members	`com_liferay_invitation_invite_members_web_portlet_InviteMemb`
Message Boards	`com_liferay_message_boards_web_portlet_MBPortlet`
Recent Bloggers	`com_liferay_blogs_recent_bloggers_web_portlet_RecentBloggers`

Community

Portlet	ID
My Sites	`com_liferay_site_my_sites_web_portlet_MySitesPortlet`
Page Comments	`com_liferay_comment_page_comments_web_portlet_PageCommentsPo`
Page Flags	`com_liferay_flags_web_portlet_PageFlagsPortlet`
Page Ratings	`com_liferay_ratings_page_ratings_web_portlet_PageRatingsPort`

Content Management

Portlet	ID
Asset Publisher	com_liferay_asset_publisher_web_portlet_AssetPublisherPortle
Breadcrumb	com_liferay_site_navigation_breadcrumb_web_portlet_SiteNaviga
Categories Navigation	com_liferay_asset_categories_navigation_web_portlet_AssetCat
Documents and Media	com_liferay_document_library_web_portlet_DLPortlet
Highest Rated Assets	com_liferay_asset_publisher_web_portlet_HighestRatedAssetsPo
Knowledge Base Article	com_liferay_knowledge_base_web_portlet_ArticlePortlet
Knowledge Base Display	com_liferay_knowledge_base_web_portlet_DisplayPortlet
Knowledge Base Search	com_liferay_knowledge_base_web_portlet_SearchPortlet
Knowledge Base Section	com_liferay_knowledge_base_web_portlet_SectionPortlet
Media Gallery	com_liferay_document_library_web_portlet_IGDisplayPortlet
Most Viewed Assets	com_liferay_asset_publisher_web_portlet_MostViewedAssetsPort
Navigation Menu	com_liferay_site_navigation_menu_web_portlet_SiteNavigationM
Nested Applications	com_liferay_nested_portlets_web_portlet_NestedPortletsPortle
Polls Display Portlet	com_liferay_polls_web_portlet_PollsDisplayPortlet
Related Assets	com_liferay_asset_publisher_web_portlet_RelatedAssetsPortlet
Site Map	com_liferay_site_navigation_site_map_web_portlet_SiteNavigat
Sites Directory	com_liferay_site_navigation_directory_web_portlet_SitesDirec
Tag Cloud	com_liferay_asset_tags_navigation_web_portlet_AssetTagsCloud
Tags Navigation	com_liferay_asset_tags_navigation_web_portlet_AssetTagsNaviga
Web Content Display	com_liferay_journal_content_web_portlet_JournalContentPortle

News

Portlet	ID
Alerts	com_liferay_announcements_web_portlet_AlertsPortlet
Announcements	com_liferay_announcements_web_portlet_AnnouncementsPortlet
Recent Content Portlet	com_liferay_asset_publisher_web_portlet_RecentContentPortlet

Sample

Portlet	ID
Hello World	com_liferay_hello_world_web_portlet_HelloWorldPortlet
IFrame	com_liferay_iframe_web_portlet_IFramePortlet

Search

Portlet	ID
Category Facet	com_liferay_portal_search_web_category_facet_portlet_Categor

Portlet	ID
Custom Facet	com_liferay_portal_search_web_custom_facet_portlet_CustomFac
Folder Facet	com_liferay_portal_search_web_folder_facet_portlet_FolderFac
Modified Facet	com_liferay_portal_search_web_modified_facet_portlet_Modifie
Search Bar	com_liferay_portal_search_web_search_bar_portlet_SearchBarPo
Search Insights	com_liferay_portal_search_web_search_insights_portlet_Search
Search Options	com_liferay_portal_search_web_search_options_portlet_SearchO
Search Results	com_liferay_portal_search_web_search_results_portlet_SearchR
Site Facet	com_liferay_portal_search_web_site_facet_portlet_SiteFacetPo
Suggestions	com_liferay_portal_search_web_suggestions_portlet_Suggestion
Tag Facet	com_liferay_portal_search_web_tag_facet_portlet_TagFacetPort
Type Facet	com_liferay_portal_search_web_type_facet_portlet_TypeFacetPo
User Facet	com_liferay_portal_search_web_user_facet_portlet_UserFacetPo

Social

Portlet	ID
Activities	com_liferay_social_activities_web_portlet_SocialActivitiesPo
Contacts Center	com_liferay_contacts_web_portlet_ContactsCenterPortlet
Members	com_liferay_social_networking_web_members_portlet_MembersPor
My Contacts	com_liferay_contacts_web_portlet_MyContactsPortlet
Profile	com_liferay_contacts_web_portlet_ProfilePortlet

Tools

Portlet	ID
Language Selector	com_liferay_site_navigation_language_web_portlet_SiteNavigat
Search	com_liferay_portal_search_web_portlet_SearchPortlet
Sign In	com_liferay_login_web_portlet_LoginPortlet

Wiki

Portlet	ID
Page Menu	com_liferay_wiki_navigation_web_portlet_WikiNavigationPageMe
Tree Menu	com_liferay_wiki_navigation_web_portlet_WikiNavigationTreeMe
Wiki	com_liferay_wiki_web_portlet_WikiPortlet
Wiki Display	com_liferay_wiki_web_portlet_WikiDisplayPortlet

161.14 FreeMarker Taglib Macros

Liferay DXP's taglibs are mapped to FreeMarker macros, so you can use them in your FreeMarker templates. See the Taglib tutorials for more information on using each taglib in your theme templates. The taglib macros are defined in `taglib-mappings.properties` files. For convenience, these macros are listed in the table below:

Macro	Taglib	TLD
liferay_aui	liferay-aui	liferay-aui.tld
liferay_portlet	liferay-portlet	liferay-portlet-ext.tld
liferay_security	liferay-security	liferay-security.tld
liferay_theme	liferay-theme	liferay-theme.tld
liferay_ui	liferay-ui	liferay-ui.tld
liferay_util	liferay-util	liferay-util.tld
portlet	portlet	liferay-portlet.tld
liferay_frontend	liferay-frontend	liferay-frontend.tld
clay	clay	liferay-clay.tld
liferay_map	liferay-map	liferay-map.tld
liferay_rss	liferay-rss	liferay-rss.tld
liferay_flags	liferay-flags	liferay-flags.tld
liferay_expando	liferay-expando	

liferay-expando.tld
liferay_journal
liferay-journal
liferay-journal.tld
liferay_social_bookmarks
liferay-social-bookmarks
liferay-social-bookmarks.tld
liferay_site
liferay-site
liferay-site.tld
liferay_comment
liferay-comment
liferay-comment.tld
liferay_social_activities
liferay-social-activities
liferay-social-activities.tld
liferay_asset
liferay-asset
liferay-asset.tld
liferay_trash
liferay-trash
liferay-trash.tld
liferay_item_selector
liferay-item-selector
liferay-item-selector.tld
liferay_layout
liferay-layout
liferay-layout.tld
liferay_editor
liferay-editor
liferay-editor.tld
liferay-fragment
liferay-fragment
liferay-fragment.tld
liferay_reading_time
liferay-reading-time
liferay-reading-time.tld
liferay_site_navigation
liferay-site-navigation
liferay-site-navigation.tld
adaptive_media_image
liferay-adaptive-media
liferay-adaptive-media.tld
liferay_product_navigation
liferay-product-navigation
liferay-product-navigation.tld

161.15 Setting up Your npm Environment

If you're using npm for development in Liferay DXP, you should set up your npm environment to avoid potential permissions issues. Follow these steps to configure your npm environment:

1. Create an `.npmrc` file in your user's home directory. This helps bypass npm permission-related issues.

2. In the `.npmrc` file, specify a `prefix` property based on your user's home directory, like the one shown below. This value specifies where to install global npm packages:

   ```
   prefix=/Users/[username]/.npm-packages
   ```

3. Set the `NPM_PACKAGES` system environment variable to the `prefix` value you just specified:

   ```
   NPM_PACKAGES=/Users/[username]/.npm-packages (same as prefix value)
   ```

4. 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
   ```

161.16 Liferay JS Generator

This reference section covers these topics for the Liferay JS Generator:

- Understanding the JS Portlet Extender's Configuration
- A reference list of available commands for the Liferay JS Generator
- Configuration JSON options

161.17 Understanding the JS Portlet Extender Configuration

Bundles generated with the Liferay JS Generator require specific method signatures, MANIFEST headers, and configuration within their `package.json` file to use the JS Portlet Extender. This configuration is provided by default. For reference, this configuration is covered in detail below.

Manifest Header

The OSGi bundle is identified with the MANIFEST header shown below, which specifies to process it with the JS Portlet Extender:

```
Require-Capability: osgi.extender;filter:="(osgi.extender=liferay.npm.portlet)"
```

Main Entry Point

The main module of your JavaScript widget must export a JavaScript function with the signature below. Bundles created with the Liferay JS Generator have this out-of-the-box:

```
function({portletNamespace, contextPath, portletElementId, configuration}) {
  ...
}
```

The entry point function receives one object parameter with four fields:

- `portletNamespace`: the unique namespace of the widget as defined in the Portlet specification.
- `contextPath`: the URL path that can be used to retrieve bundle resources from the browser (it doesn't contain the protocol, host, or port, just the absolute path).
- `portletElementId`: the DOM identifier of the widget's `<div>` node that can be used to render HTML.
- `configuration` (optional): since JS Portlet Extender version 1.1.0, this field contains the system (OSGi) and portlet instance (preferences as described in the Portlet spec) configuration for the widget. It has two subfields:
  - **system:** contains the system level configuration (defined in Control Panel → System Settings)
  - **portletInstance:** contains the per-widget configuration (defined in the Configuration menu option of the widget)

Note that all values are received as strings, no matter what their type is in OSGi configuration store.

The JavaScript-based widget's main `index.js` file configuration is shown below for reference. Note that system settings and localization are enabled in the example below:

```
export default function main({portletNamespace, contextPath, portletElementId, configuration}) {

    const node = document.getElementById(portletElementId);

    node.innerHTML =`
        <div>
            <span class="tag">${Liferay.Language.get('porlet-namespace')}:</span>
            <span class="value">${portletNamespace}</span>
        </div>
        <div>
            <span class="tag">${Liferay.Language.get('context-path')}:</span>
            <span class="value">${contextPath}</span>
        </div>
        <div>
            <span class="tag">${Liferay.Language.get('portlet-element-id')}:</span>
            <span class="value">${portletElementId}</span>
        </div>

        <div>
            <span class="tag">${Liferay.Language.get('configuration')}:</span>
            <span class="value">
                ${JSON.stringify(configuration, null, 2)}
            </span>
        </div>
```

```
    `;
}
```

The JavaScript file containing the main entry point function is specified in the `main` entry of the `package.json` file. Below is the `main` entry for the *JavaScript based portlet*:

```
"main": "index.js"
```

161.18 Liferay JS Generator Commands

The npm commands shown below are available for the Liferay JS Generator:

- `npm run build`: Places the output of liferay-npm-bundler in the designated output folder. The standard output is a JAR file that can be deployed manually to Liferay DXP.
- `npm run deploy`: Deploys the bundle to the configured app server
- `npm run start`: Tests the application in a local webpack installation instead of a Liferay DXP server. This speeds up development because you can see live changes without any need to deploy. Note, however, that because this is separate from a Liferay instance, you don't have access to Liferay's APIs.

Note: By default, the webpack server uses port 8080. You can point the webpack server to a different port by setting the `port` key in `.npmbuildrc`:

```
"webpack": {
  "port": 2070
}
```

- `npm run translate`: Runs the translation features for your bundle. Note that this feature requires Microsoft Translator credentials. See Using Translation Features in Your widget for more information.

161.19 Configuring System Settings for OSGi Bundles Created with the liferay-npm-bundler

If you're creating an OSGi bundle with the Liferay JS Generator and want to provide system settings for your widget, you must provide a `configuration.json` file. This reference guide lists the available configuration options for `configuration.json` along with example code.

JSON Format

The `configuration.json` must follow the basic pattern shown below:

```
{
  "system": {
    "category": "{category identifier}",
    "name": "{name of configuration}",
    "fields": {
      "{field id 1}": {
        "type": "{field type}",
        "name": "{field name}",
        "description": "{field description}",
        "default": "{default value}",
        "options": {
          "{option id 1}": "{option name 1}",
          "{option id 2}": "{option name 2}",

          "{option id n}": "{option name n}"
        }
      },
      "{field id 2}": {},

      "{field id n}": {}
    }
  },
  "portletInstance": {
    "name": "{name of configuration}",
    "fields": {
      "{field id 1}": {
        "type": "{field type}",
        "name": "{field name}",
        "description": "{field description}",
        "default": "{default value}",
        "options": {
          "{option id 1}": "{option name 1}",
          "{option id 2}": "{option name 2}",

          "{option id n}": "{option name n}"
        }
      },
      "{field id 2}": {},

      "{field id n}": {}
    }
  }
}
```

The available options are described in the table below:

Option	Value
`{category identifier}`	Describes the identifier of the configuration category where the settings must be placed. It's equivalent to the category field of the `@ExtendedObjectClassDefinition` annotation explained here. The category field of `configuration.json` is optional and, when not set, the project's name specified in `package.json` is used. You need JS Portlet Extender 1.1.0+ for this feature to work. Otherwise, the system configuration will show up under *Platform → Third Party* in System Settings.
`{name of configuration}`	the configuration's name as a string or a localization key. If no value is given, the bundler falls back to the project's name, then description given in `package.json`.
`{field id}`	the field's name as a string or a localization key
`{field type}`	specifies the field's type, which can be one of the following types: - `number`: an integer number - `float`: a floating point number - `string`: a string - `boolean`: true or false - `password`: a password (string)
`{field name}`	the field's name as a string or a localization key
`{field description}`	an optional string or a localization key that describes the field's purpose and appears as hint text near it
`{default value}`	an optional default value for the field
`options`	an optional section that defines a fixed set of values for the field
`{option id}`	a string that defines the option's ID
`{option name}`	the option's name as a string or a localization key

An example configuration is shown below:

```
{
  "system": {
    "category": "third-party",
    "name": "My project",
    "fields": {
      "a-number": {
        "type": "number",
        "name": "A number",
        "description": "An integer number",
        "default": "42"
      },
      "a-string": {
        "type": "string",
        "name": "A string",
        "description": "An arbitrary length string",
        "default": "this is a string"
```

```
      },
      "a-password": {
        "type": "password",
        "name": "A password",
        "description": "A secret string",
        "default": "s3.cr3t"
      },
      "a-boolean": {
        "type": "boolean",
        "name": "A boolean",
        "description": "A true|false value",
        "default": true
      },
      "an-option": {
        "type": "string",
        "name": "An option",
        "description": "A restricted values option",
        "required": true,
        "default": "A",
        "options": {
          "A": "Option a",
          "B": "Option b"
        }
      }
    }
  },
  "portletInstance": {
    "name": "Widget configuration",
    "fields": {
      "a-float": {
        "type": "float",
        "name": "A float",
        "description": "A floating point number",
        "default": "1.1"
      }
    }
  }
}
```

CHAPTER 162

SCREENLETS IN LIFERAY SCREENS

Liferay Screens includes several Screenlets that you can use in your mobile apps. Screenlets are ready-to-use components that contain a complete UI and the code necessary to call Liferay DXP's remote services for tasks like logging in, displaying portal content, submitting forms, and much more.

This section contains each Screenlet's reference documentation in separate sections for Android and iOS. Each document in these sections lists a Screenlet's features, compatibility, available Views, attributes, listener methods, and more:

- Screenlets in Liferay Screens for Android
- Screenlets in Liferay Screens for iOS

Note: This section only contains Screenlet reference documentation. For instructional information on installing, using, and customizing Liferay Screens and its Screenlets, see the Screens tutorials for Android and iOS.

CHAPTER 163

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 DXP instance.
- **Asset List Screenlet:** Shows a list of assets managed by the 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 the Documents and Media Library.
- **Video Display Screenlet:** Shows a single video file from the Documents and Media Library.
- **Audio Display Screenlet:** Shows a single audio file from the Documents and Media Library.
- **PDF Display Screenlet:** Shows a single PDF file from the 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.

163.1 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

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 OAuth 2.

- **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.

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. For more information, see the User Guide's authentication section.

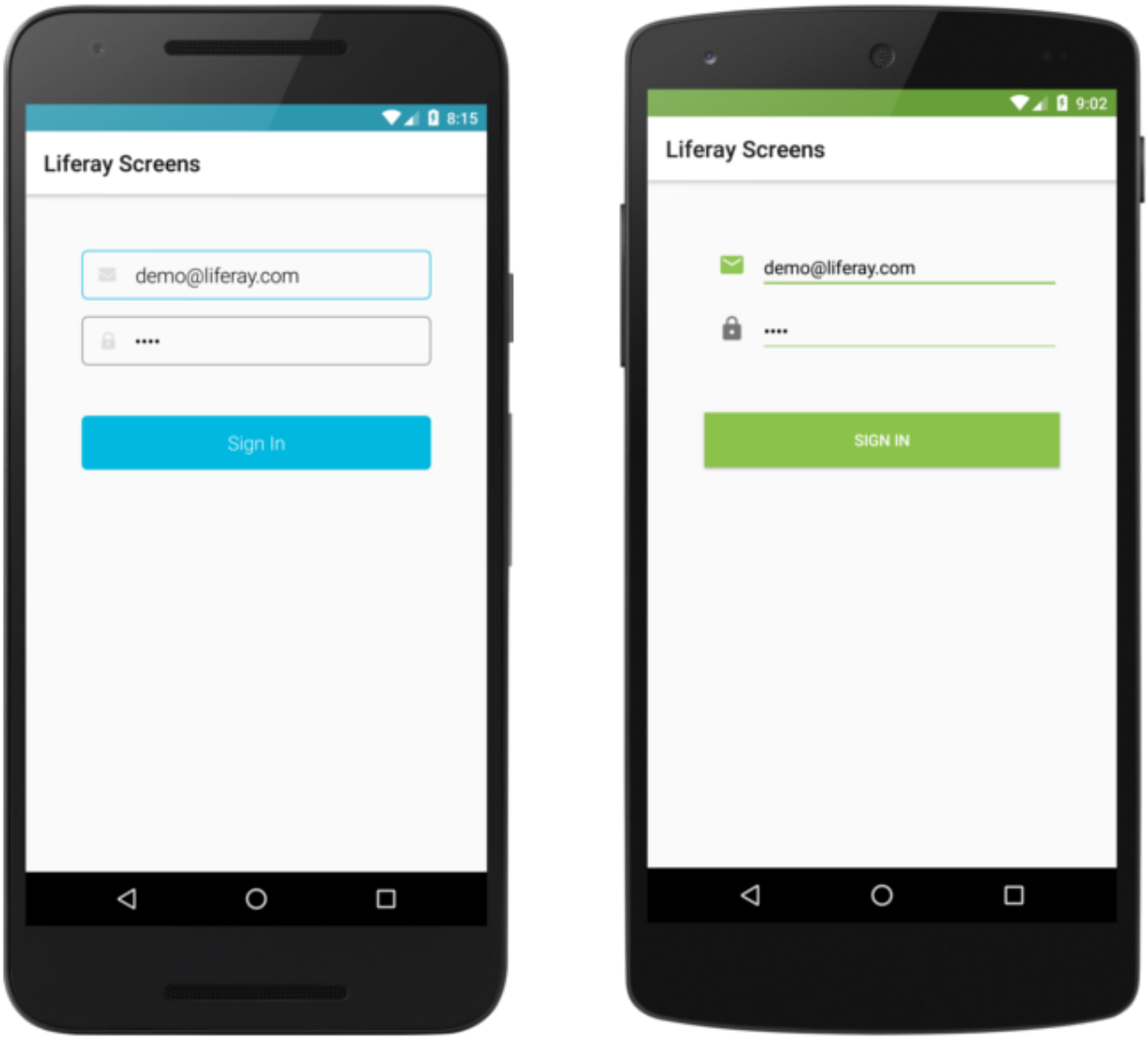


Figure 163.1: The Login Screenlet using the Default (left) and Material (right) Viewsets.

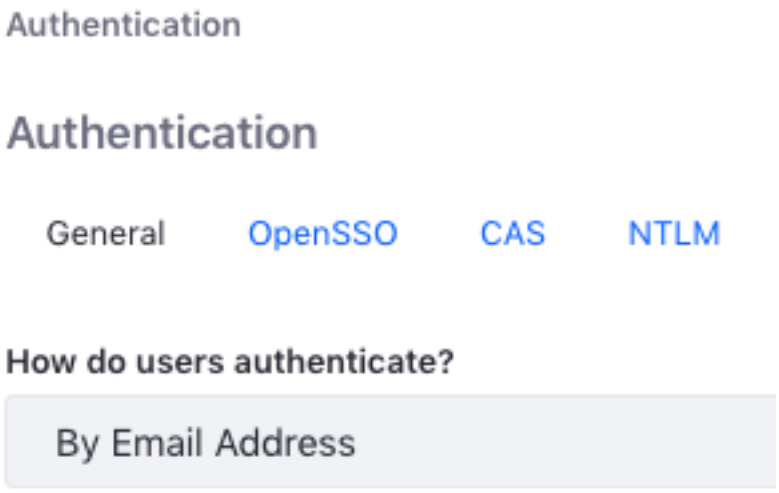


Figure 163.2: Set the authentication method in your Liferay DXP instance.

OAuth Authentication

For instructions on using OAuth with Login Screenlet, see the tutorial on using OAuth 2 with Liferay Screens.

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`, `cookie`, `oauth2Redirect`, or `oauth2UsernameAndPassword`. 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 one of the OAuth values or `cookie`, the Screenlet defaults to basic authentication with the `email` option. | `oauth2Redirect` |

`string` | The URL that the mobile browser will redirect the user to after successful login. You must configure this in the portal's OAuth 2 Admin portlet, and associate the URL with the Android app. | `oauth2ClientId` | `string` | The ID of the OAuth 2 application in the portal. You can find this value in the portal's OAuth 2 Admin portlet. | `oauth2ClientSecret` | `string` | The client secret of the OAuth 2 application in the portal. You can find this value in the portal's OAuth 2 Admin portlet. | `oauth2Scopes` | `string` | The portal permissions to request. You can define a set of permissions associated with an OAuth 2 application in the portal's OAuth 2 Admin portlet. Use this attribute to request a subset of those permissions. Separate multiple scopes with a space (e.g., `"scope1 scope2 scope3"`). | `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;
    }
}
```

163.2 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

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

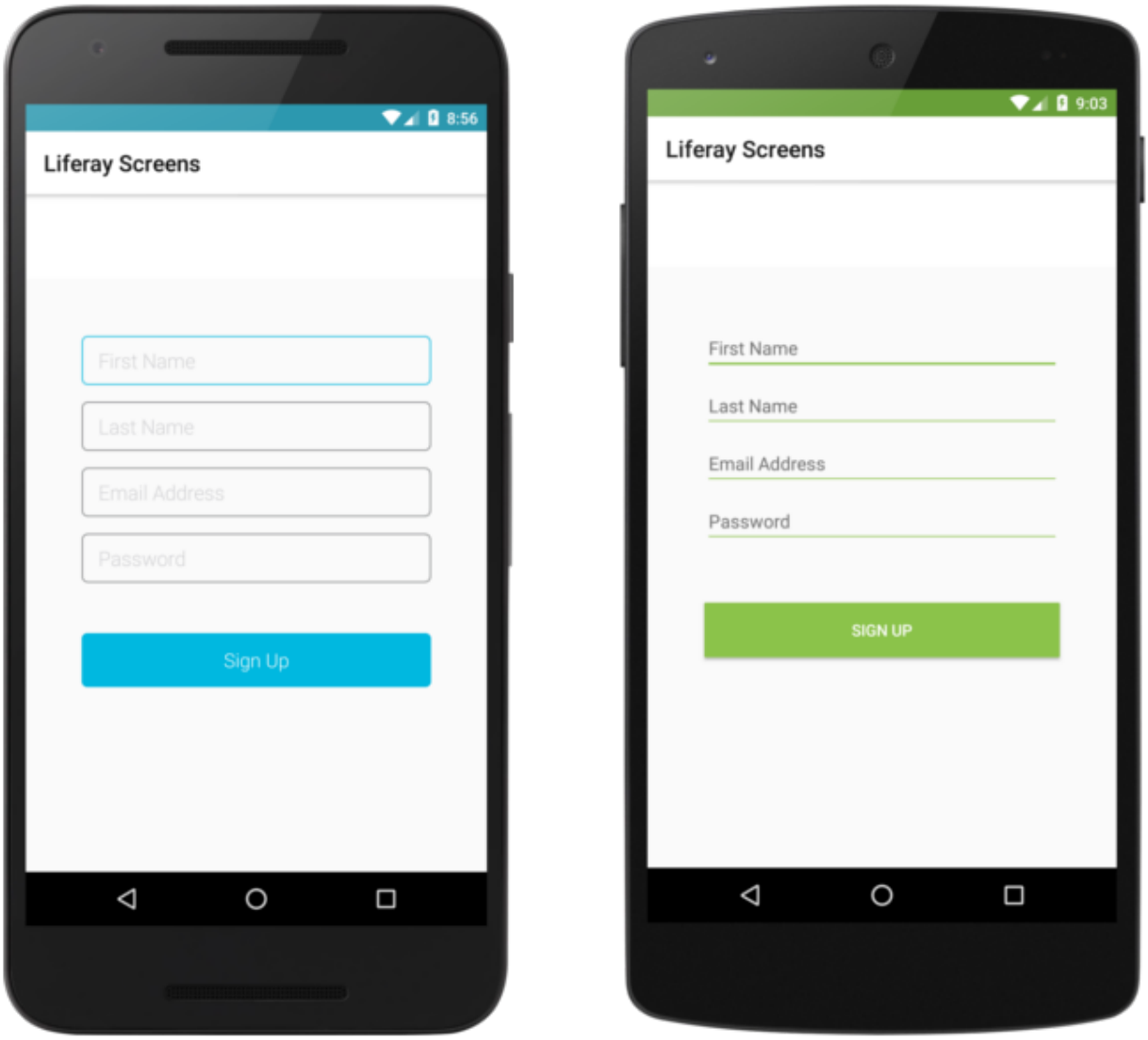


Figure 163.3: The Sign Up Screenlet with the Default (left) and Material (right) Viewsets.

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.

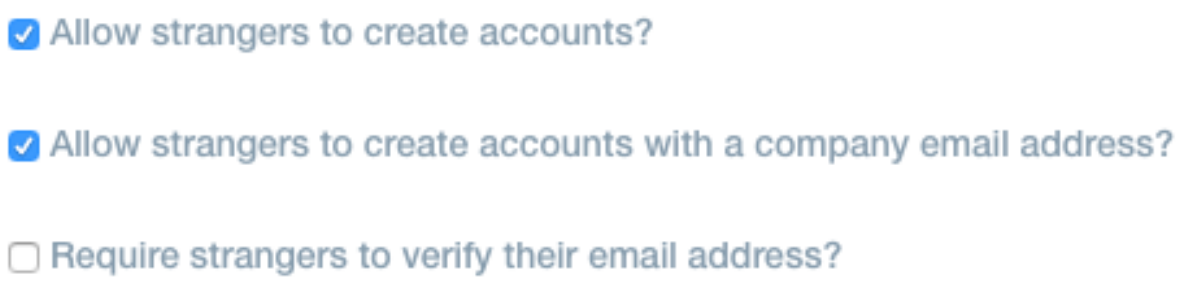


Figure 163.4: The Liferay instance's authentication settings.

For more details, see the Authentication 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, as defined in the portal's User entity.
- onSignUpFailure(Exception e): Called when an error occurs in the process.

163.3 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

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, see the Authentication 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.

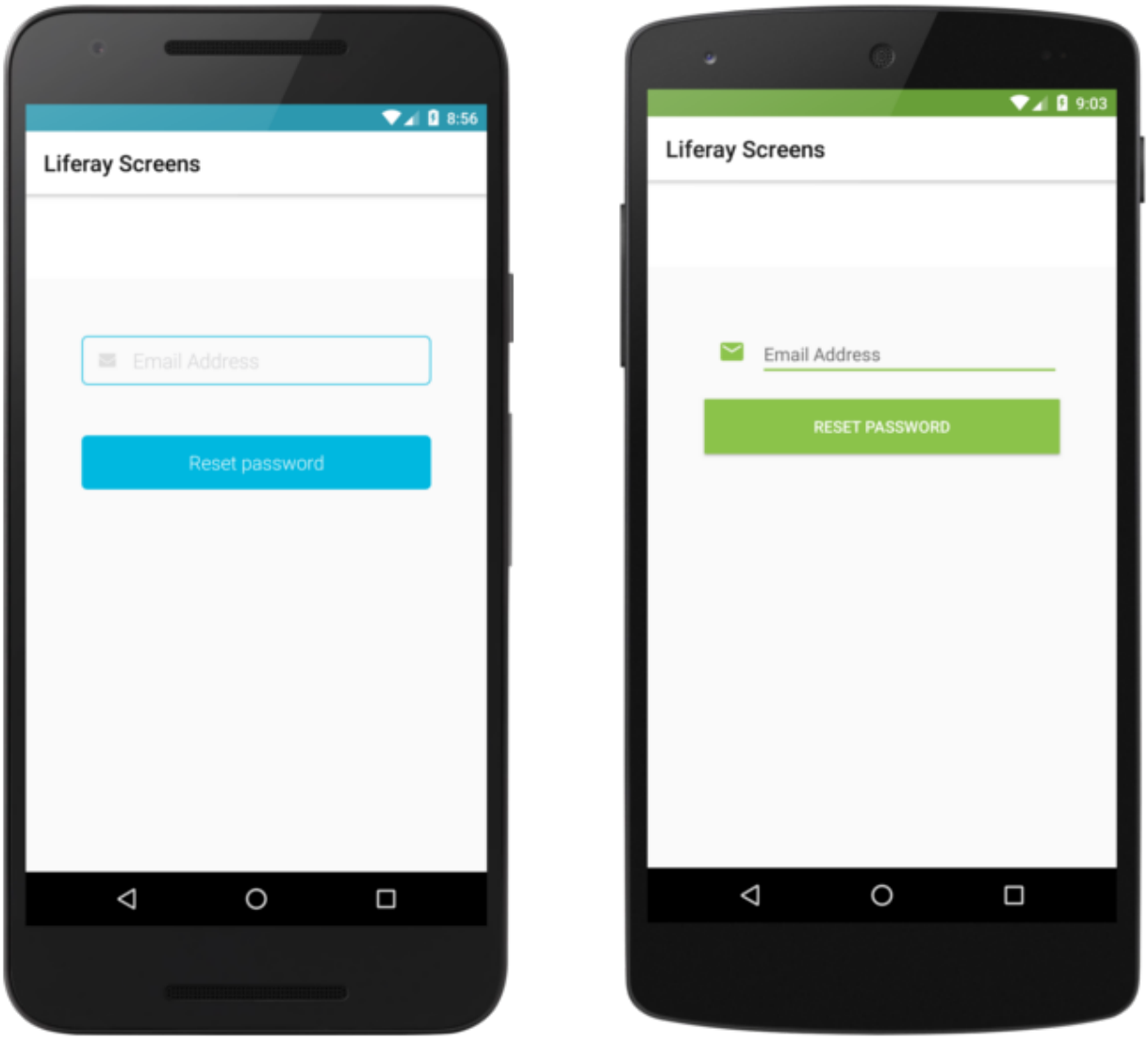


Figure 163.5: The Forgot Password Screenlet with the Default (left) and Material (right) Viewsets.

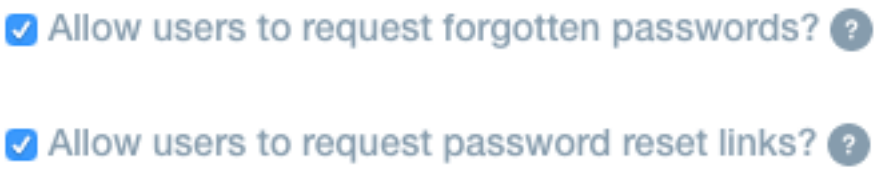


Figure 163.6: Checkboxes for the password recovery features in your Liferay instance.

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 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.

163.4 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
- 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

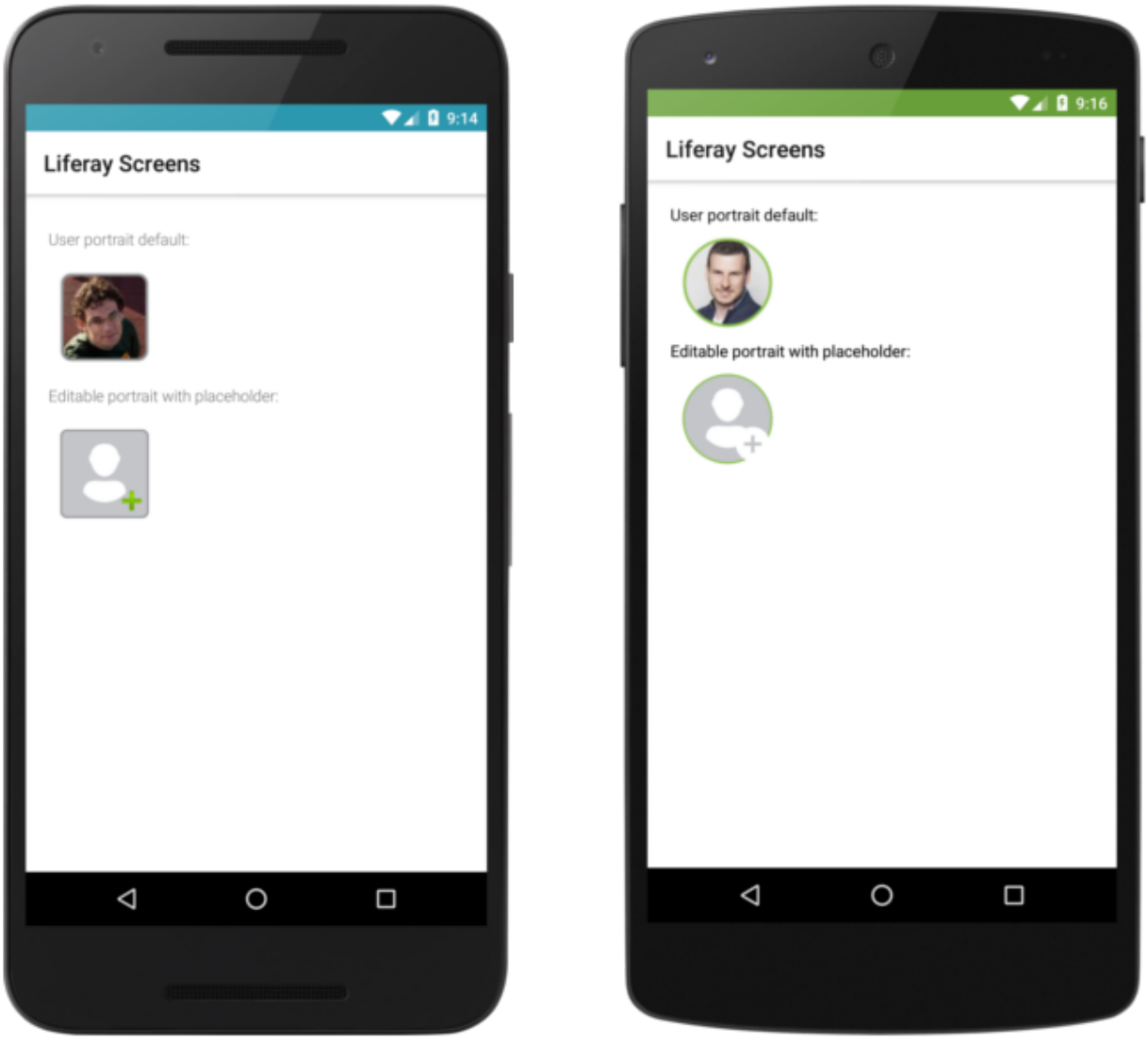


Figure 163.7: The User Portrait Screenlet using the Default (left) and Material (right) Views.

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 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 `portraitId`and `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.

163.5 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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
`Screensddlrecord Service` (Screens compatibility plugin)	`getDDMStructureVersion`	Load form
`Screensddlrecord Service` (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.

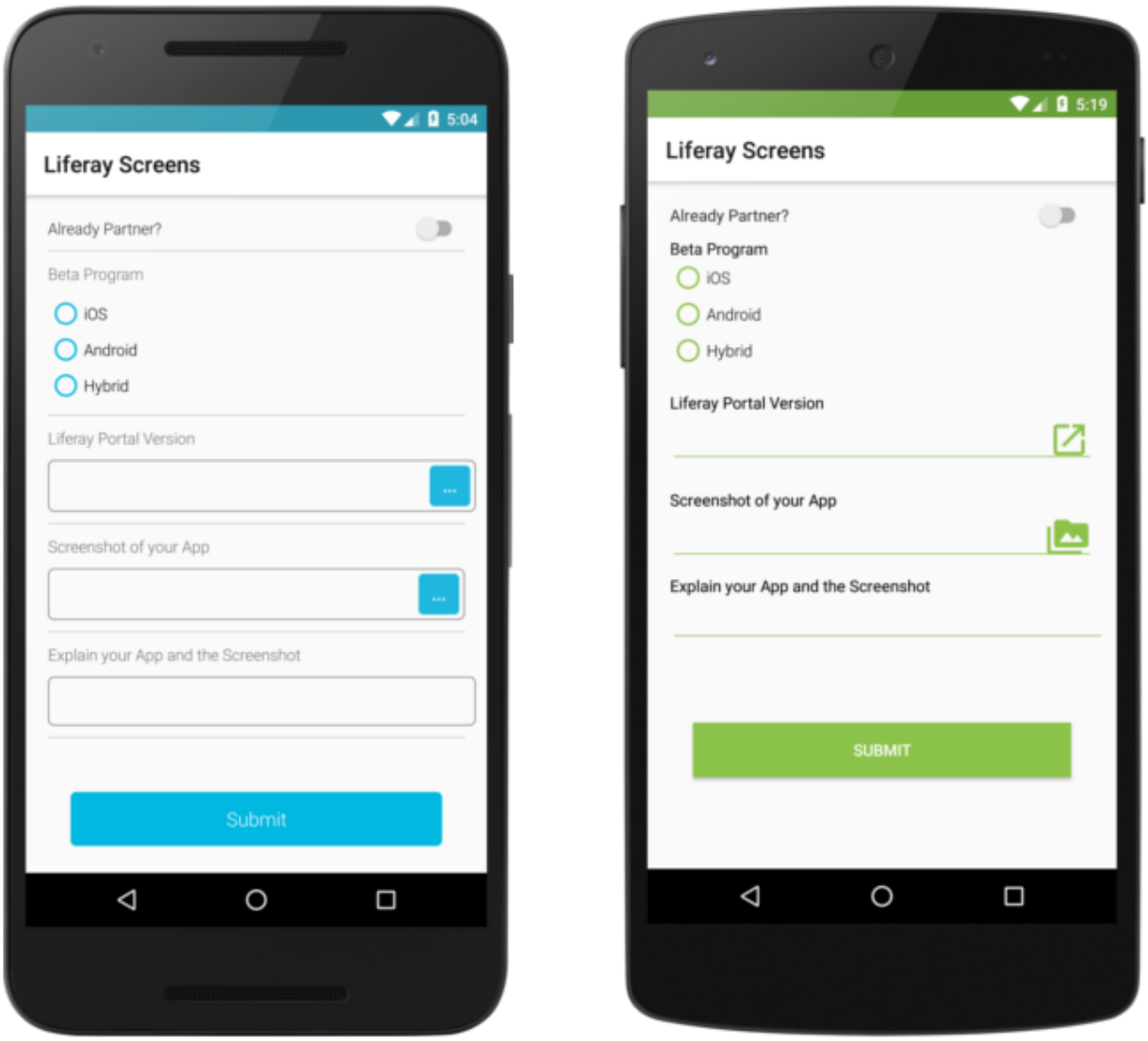


Figure 163.8: DDL Form Screenlet's Default (left) and Material (right) Views.

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

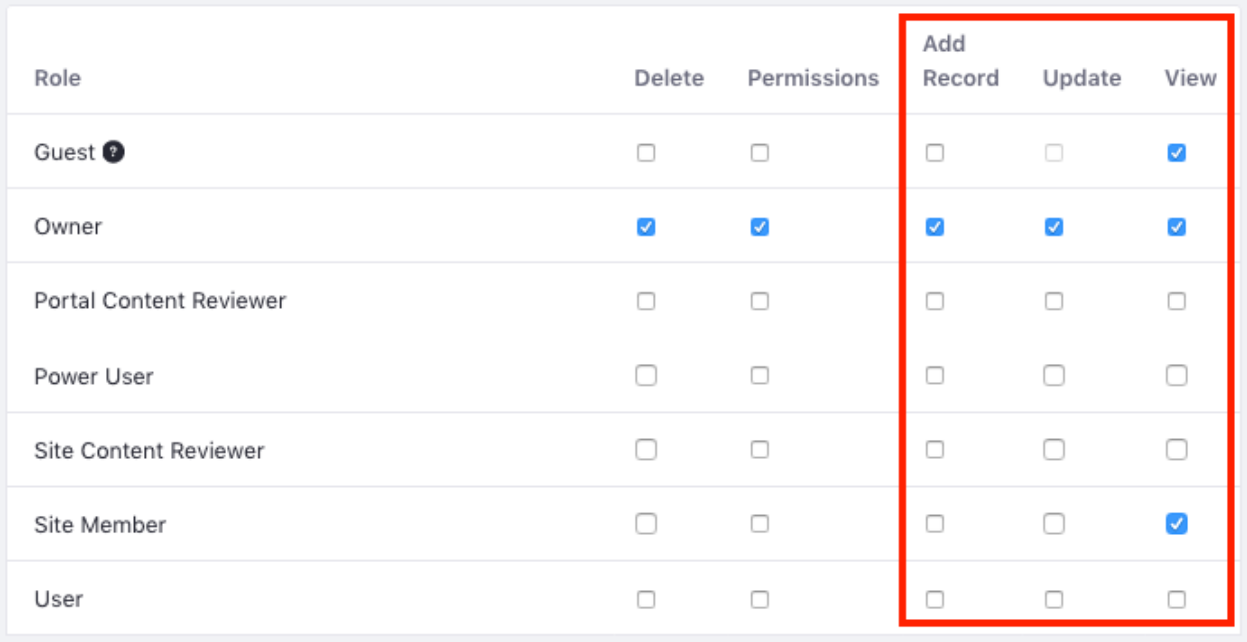


Figure 163.9: The permissions for adding, viewing, and editing DDL records.

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.

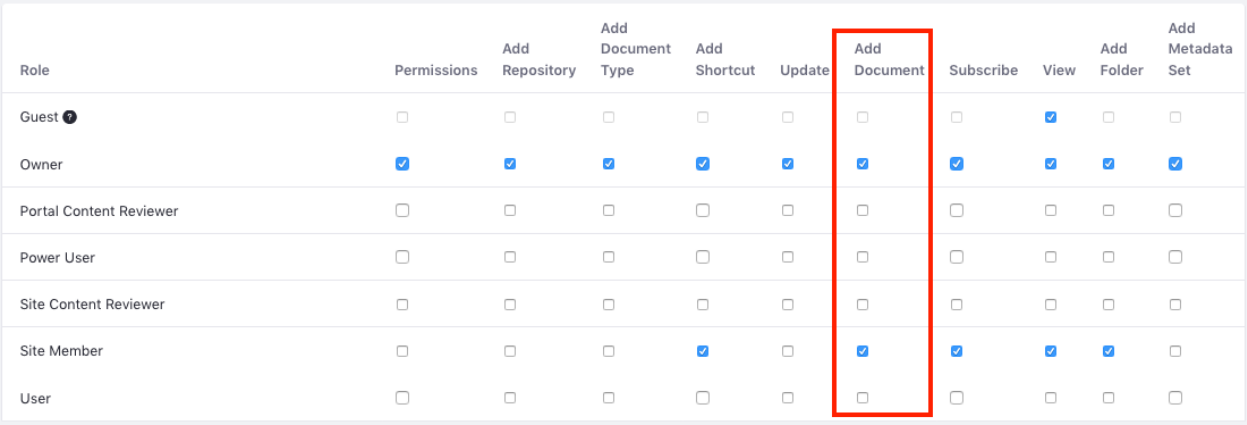


Figure 163.10: The permission for adding a document to a Documents and Media folder.

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:

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.

163.6 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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
`Screensddlrecord Service` (Screens compatibility plugin)	`getDdlRecords`	With `ddlRecordSetId`, or `ddlRecordSetId` and `userId`
`Screensddlrecord Service` (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.

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

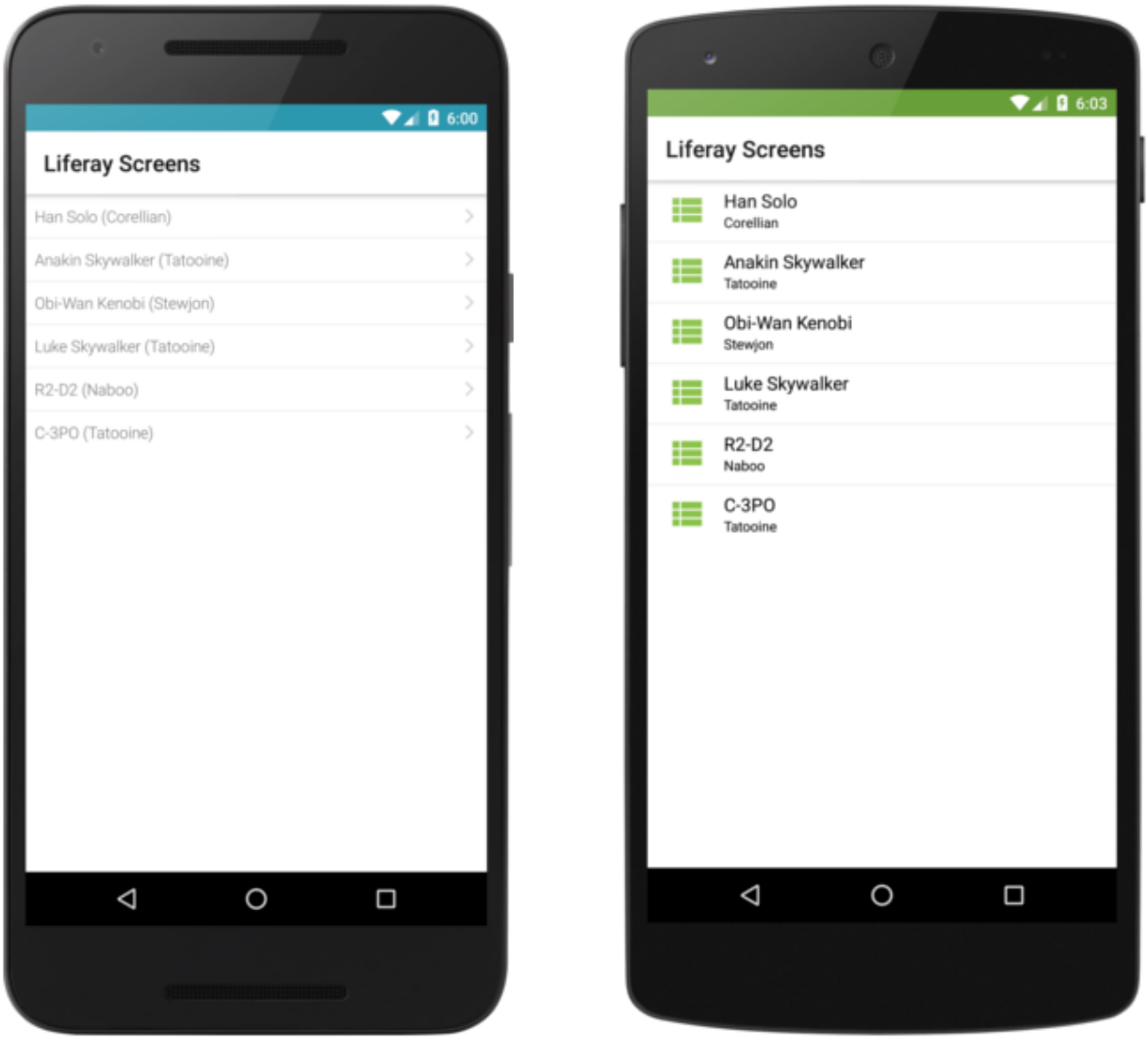


Figure 163.11: The DDL List Screenlet using the Default and Material Views.

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.

163.7 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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
Screensddlrecord Service (Screens compatibility plugin)	getAssetEntries	With entryQuery
Screensddlrecord Service (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.

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.

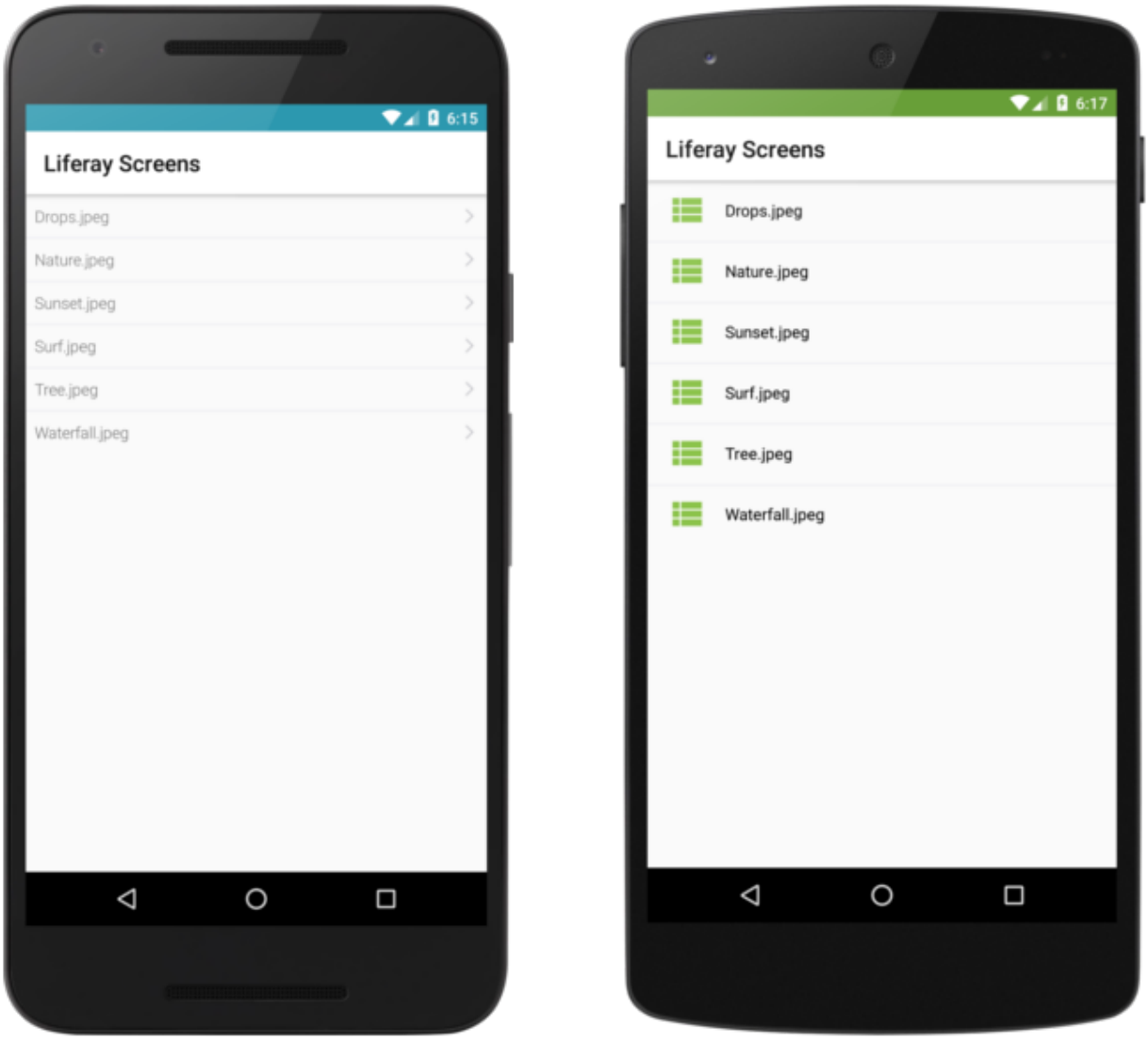


Figure 163.12: Asset List Screenlet using the Default (left) and Material (right) Views.

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.

163.8 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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 web content section of the 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. |

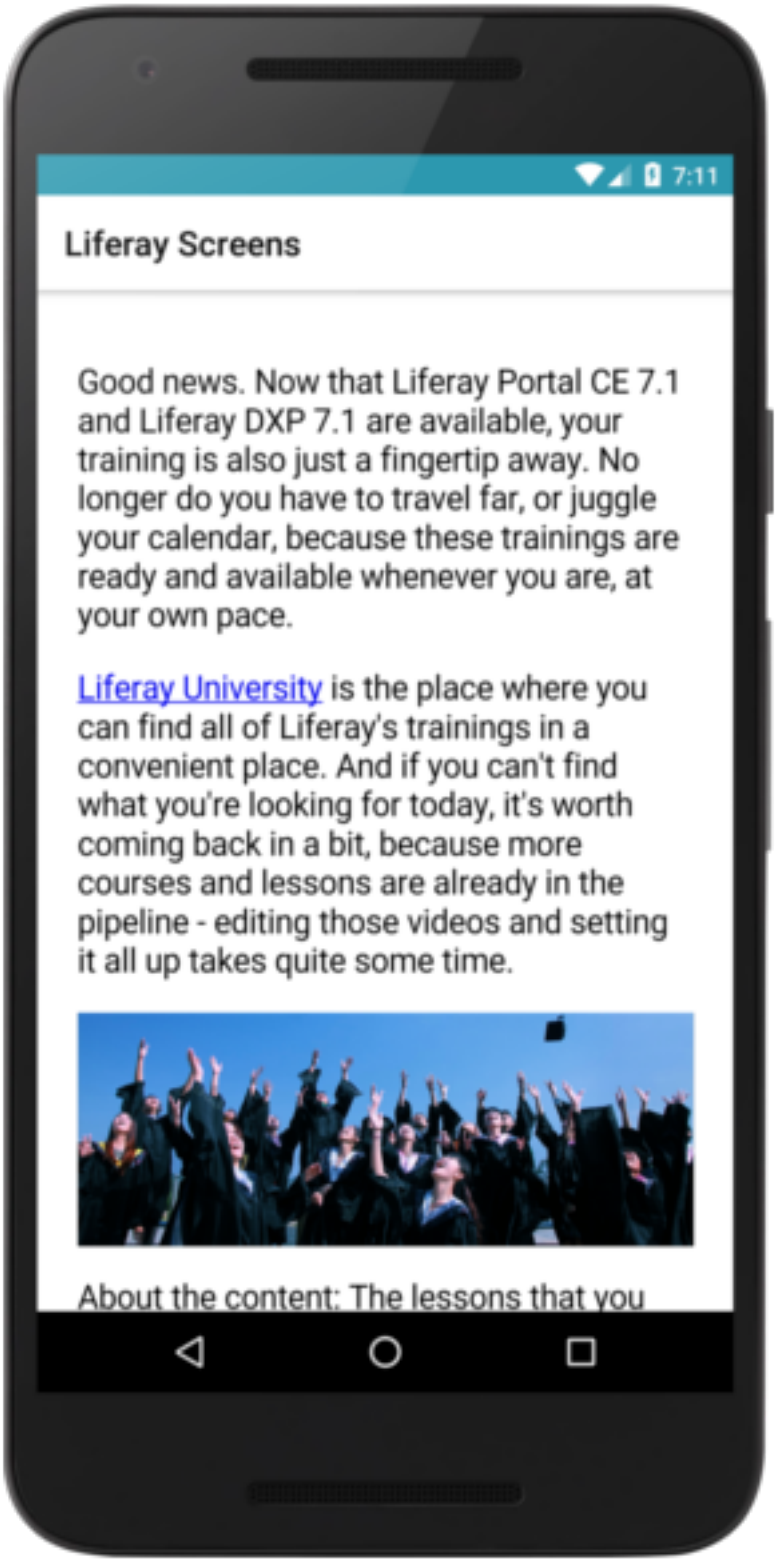


Figure 163.13: Web Content Display Screenlet using the Default View.

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 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.

163.9 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

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` |

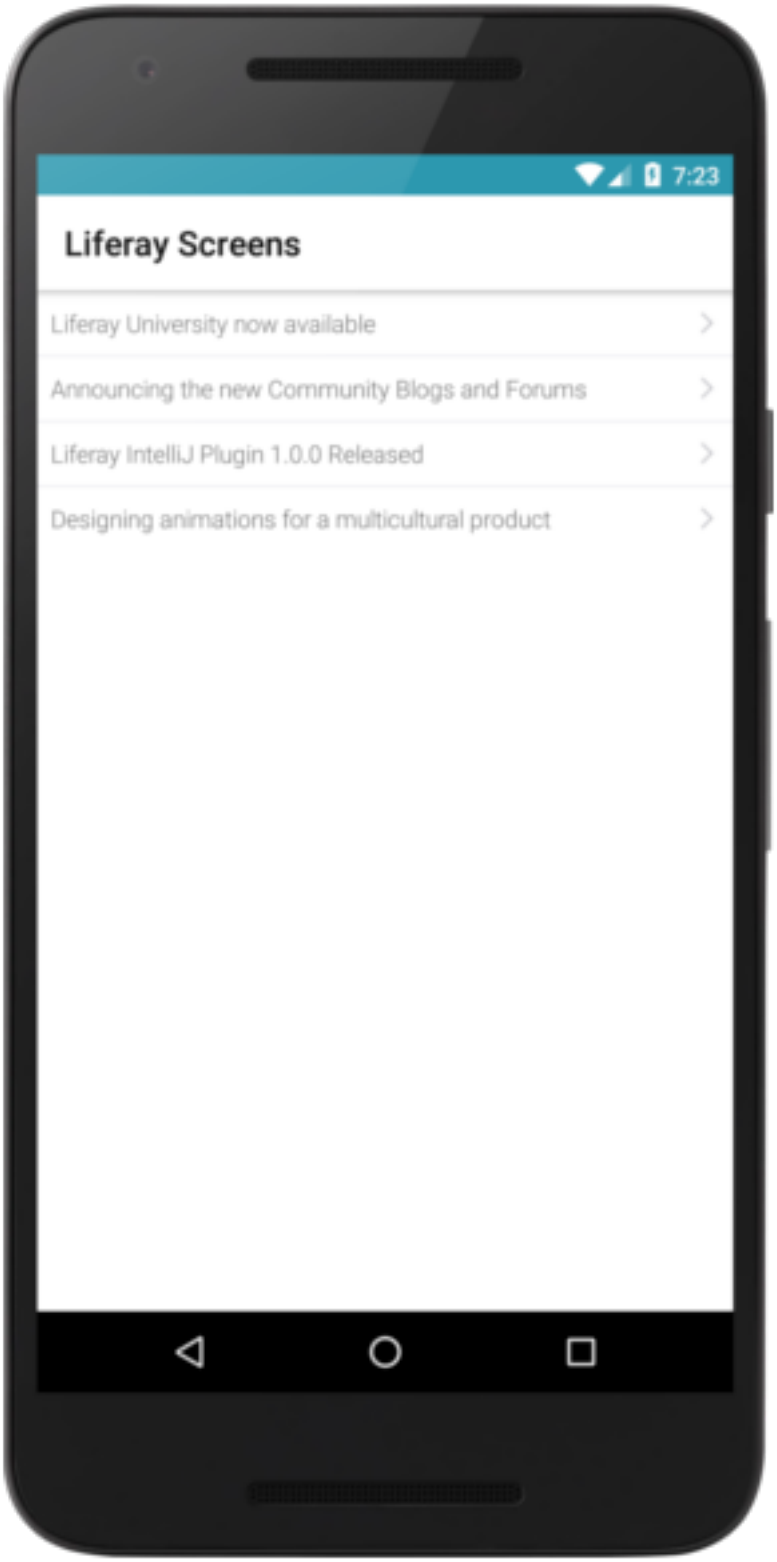


Figure 163.14: The Web Content List Screenlet using the Default View.

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 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).

163.10 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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:

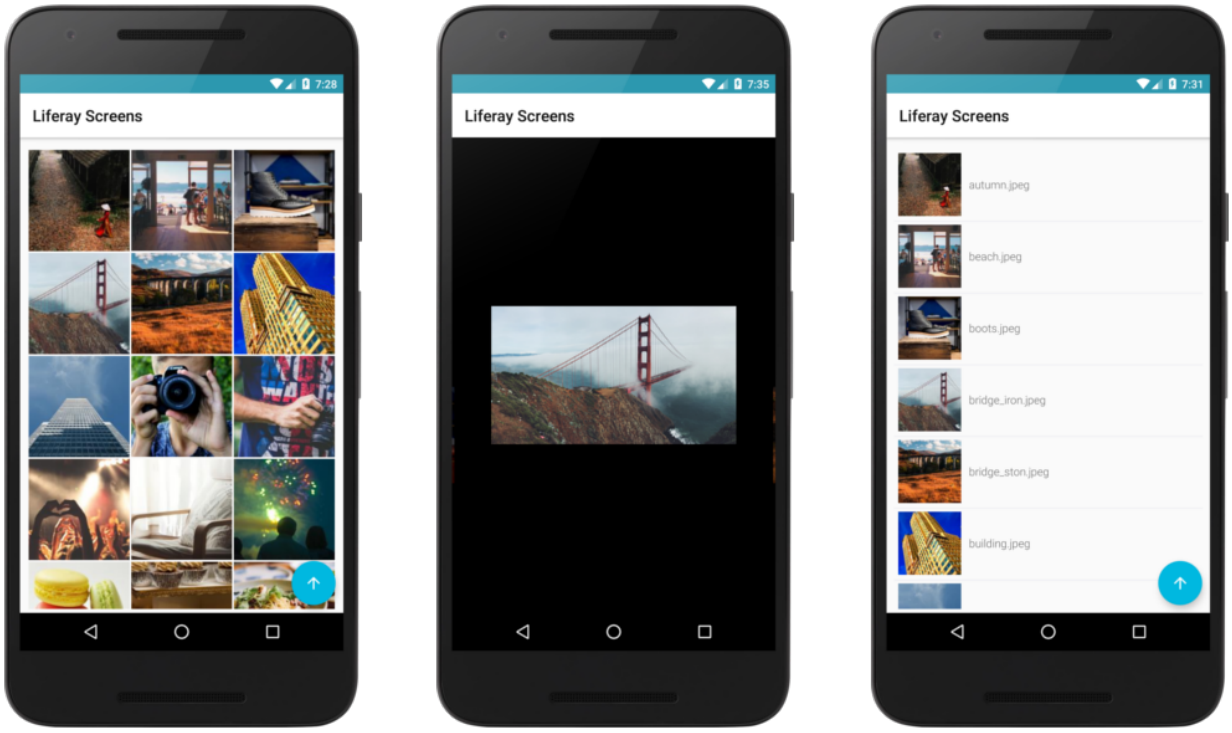


Figure 163.15: Image Gallery Screenlet using the Grid, Slideshow, and List Views.

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 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.

163.11 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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
`Screensratingsentry`Service (Screens compatibility plugin)	`getRatingsEntries`	With `entryId`
`Screensratingsentry`Service (Screens compatibility plugin)	`getRatingsEntries`	With `classPK` and `className`
`Screensratingsentry`Service (Screens compatibility plugin)	`updateRatingsEntry`	
`Screensratingsentry`Service (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 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`

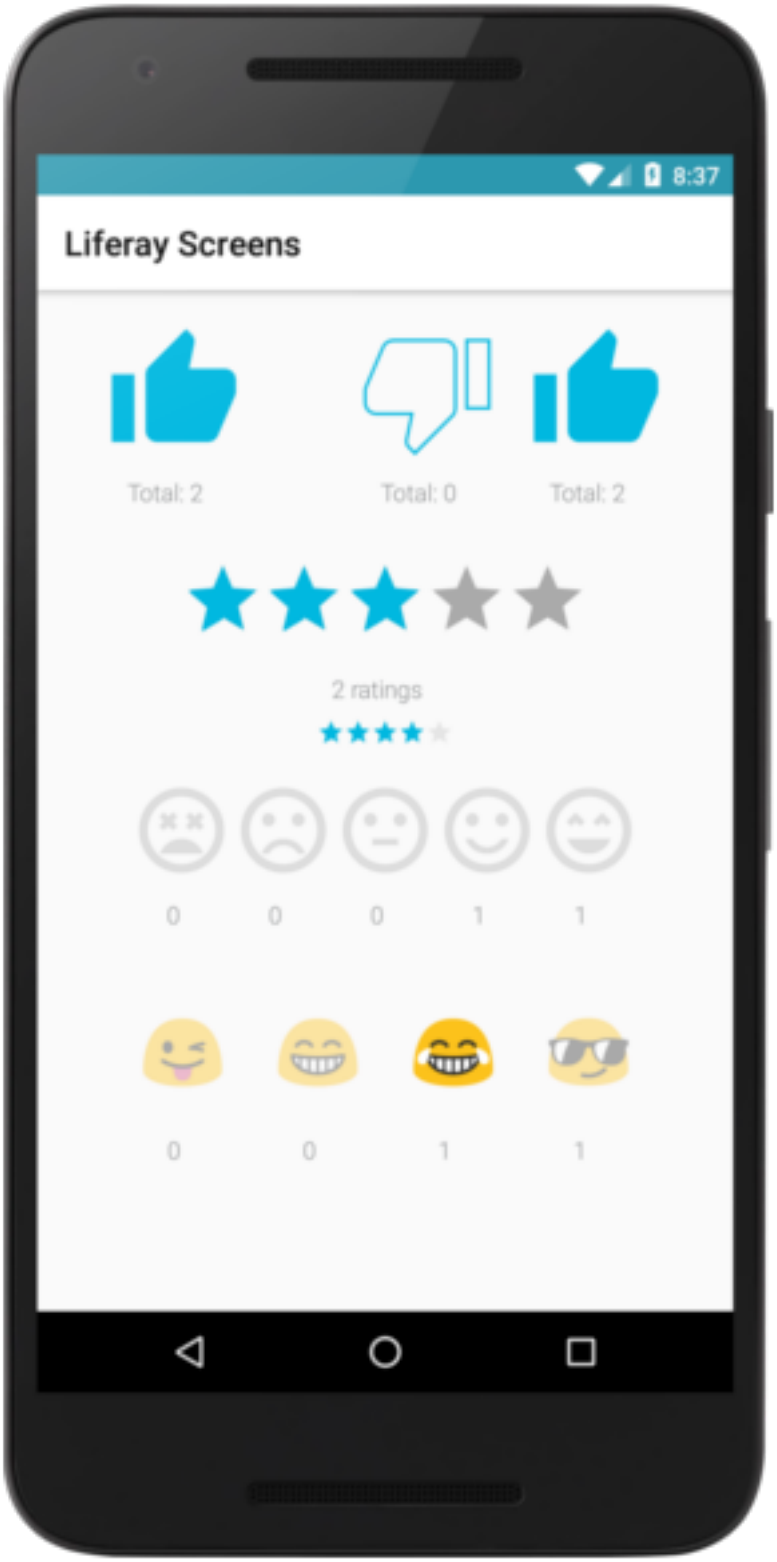


Figure 163.16: Rating Screenlet's different Views.

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.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.

163.12 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

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. |

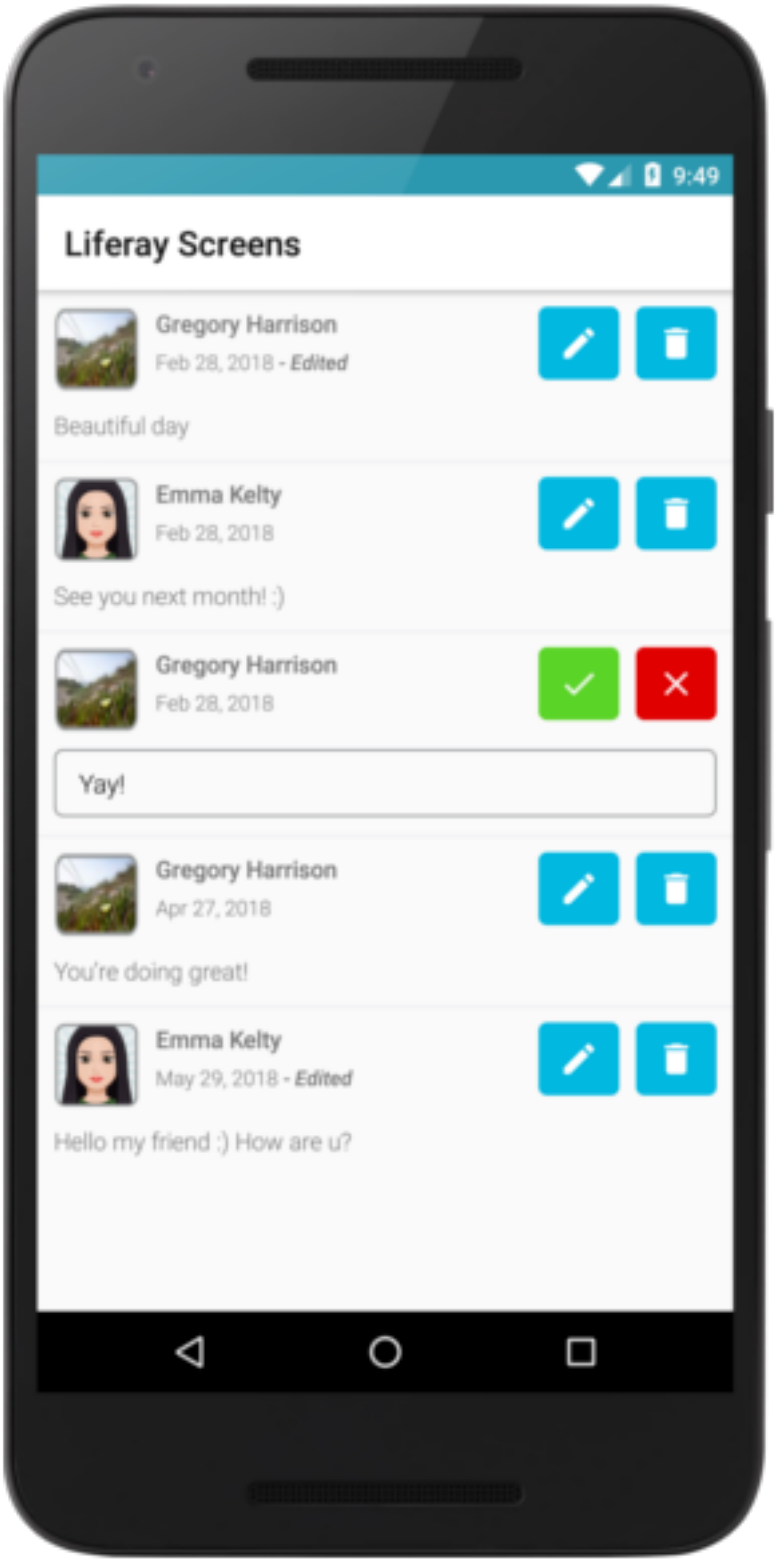


Figure 163.17: Comment List Screenlet using the Default View.

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.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.

163.13 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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`	
`Commentmanagerjsonws Service`	`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. |

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. |

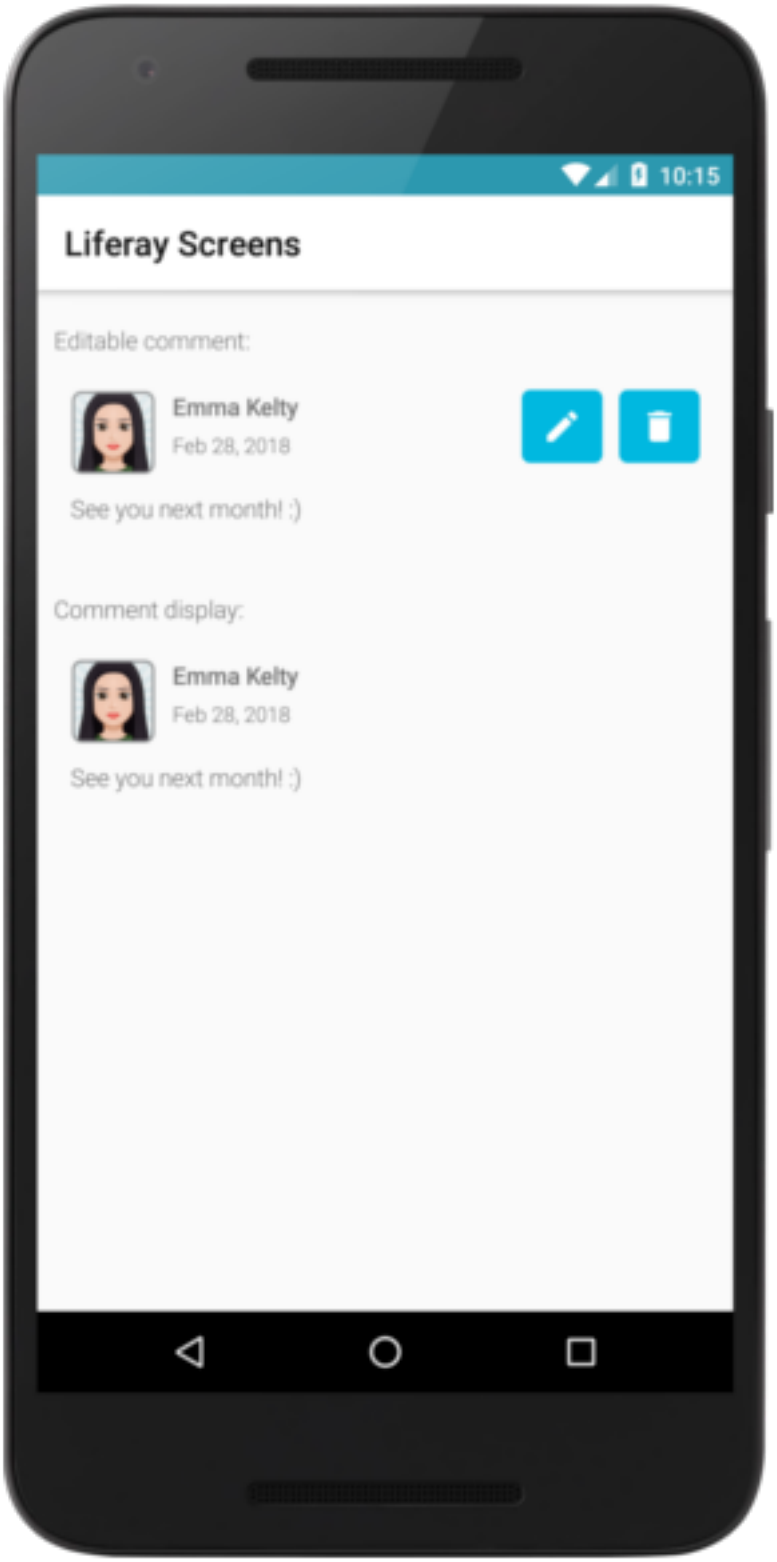


Figure 163.18: Comment Display Screenlet using the Default View.

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.

163.14 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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 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.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. |

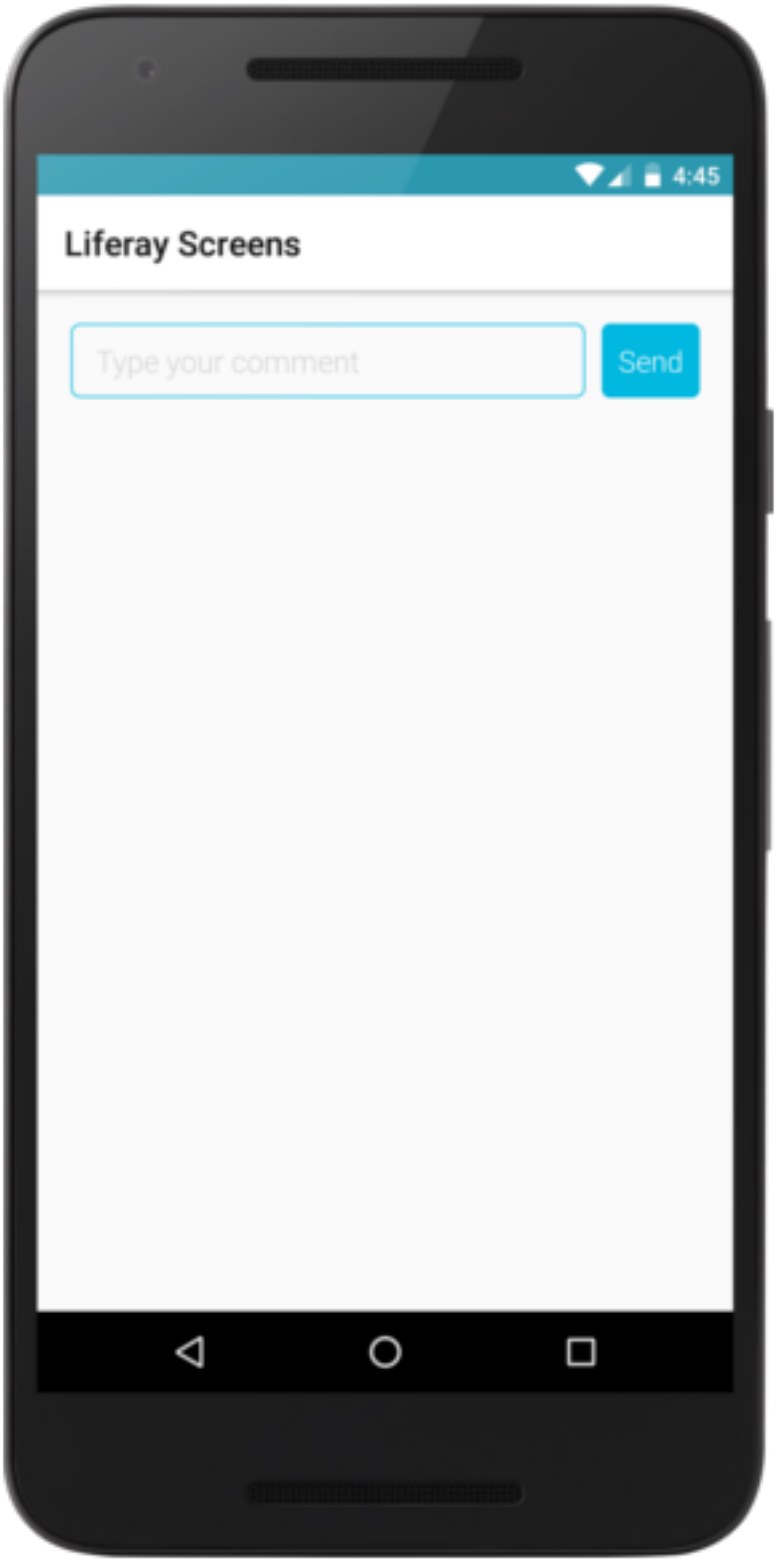


Figure 163.19: Comment Add Screenlet using the Default View.

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.

163.15 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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`

Service	Method	Notes
`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.

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,

Figure 163.20: Asset Display Screenlet using the Default View.

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.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;
}
```

163.16 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

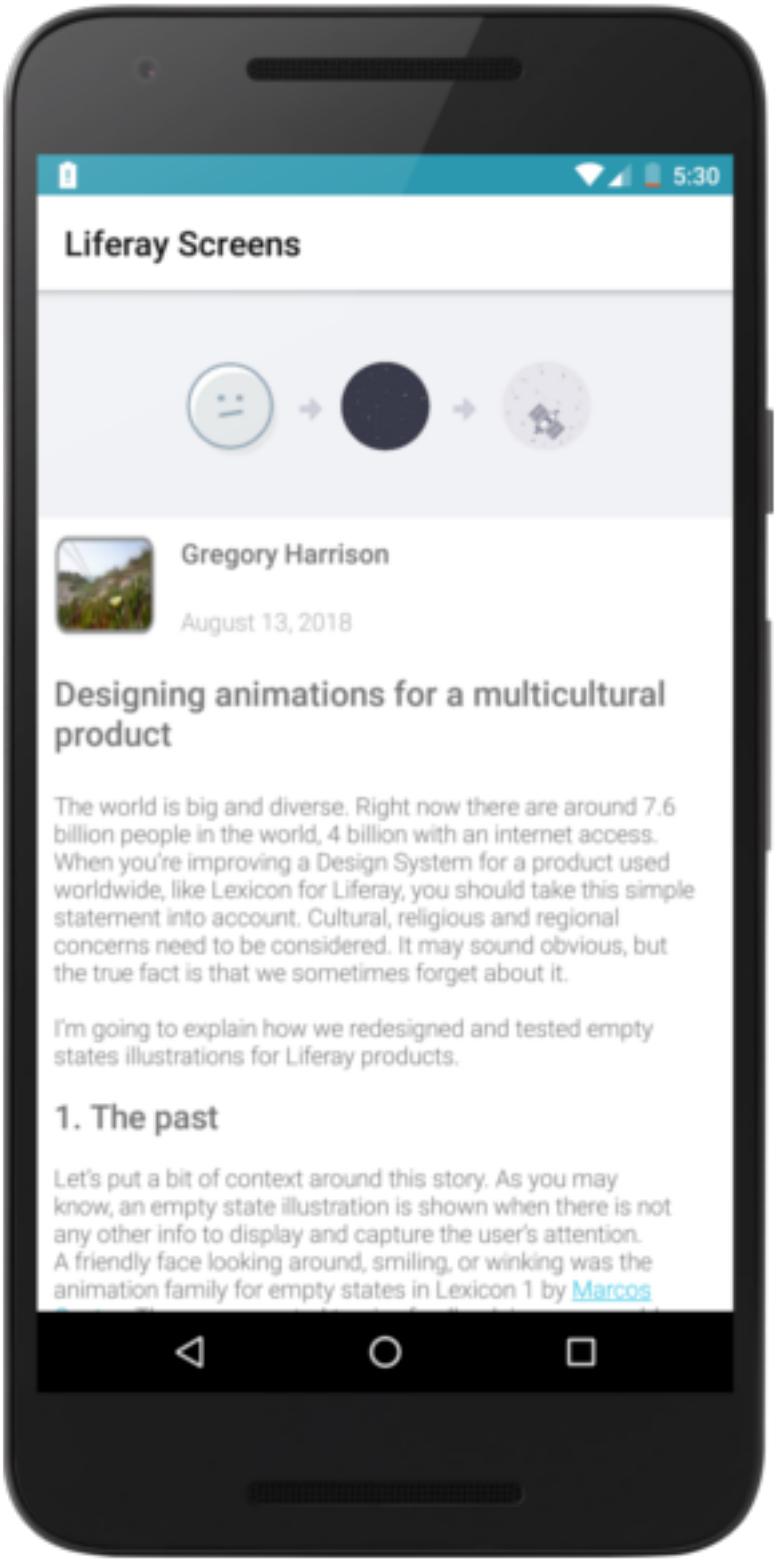


Figure 163.21: Blogs Entry Display Screenlet using the Default View.

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 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.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` 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.

163.17 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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`
`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. | `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`. |

Figure 163.22: Image Display Screenlet using the Default View.

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.

163.18 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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`

Service	Method	Notes
`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`

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

Figure 163.23: Video Display Screenlet using the Default View.

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.

163.19 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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

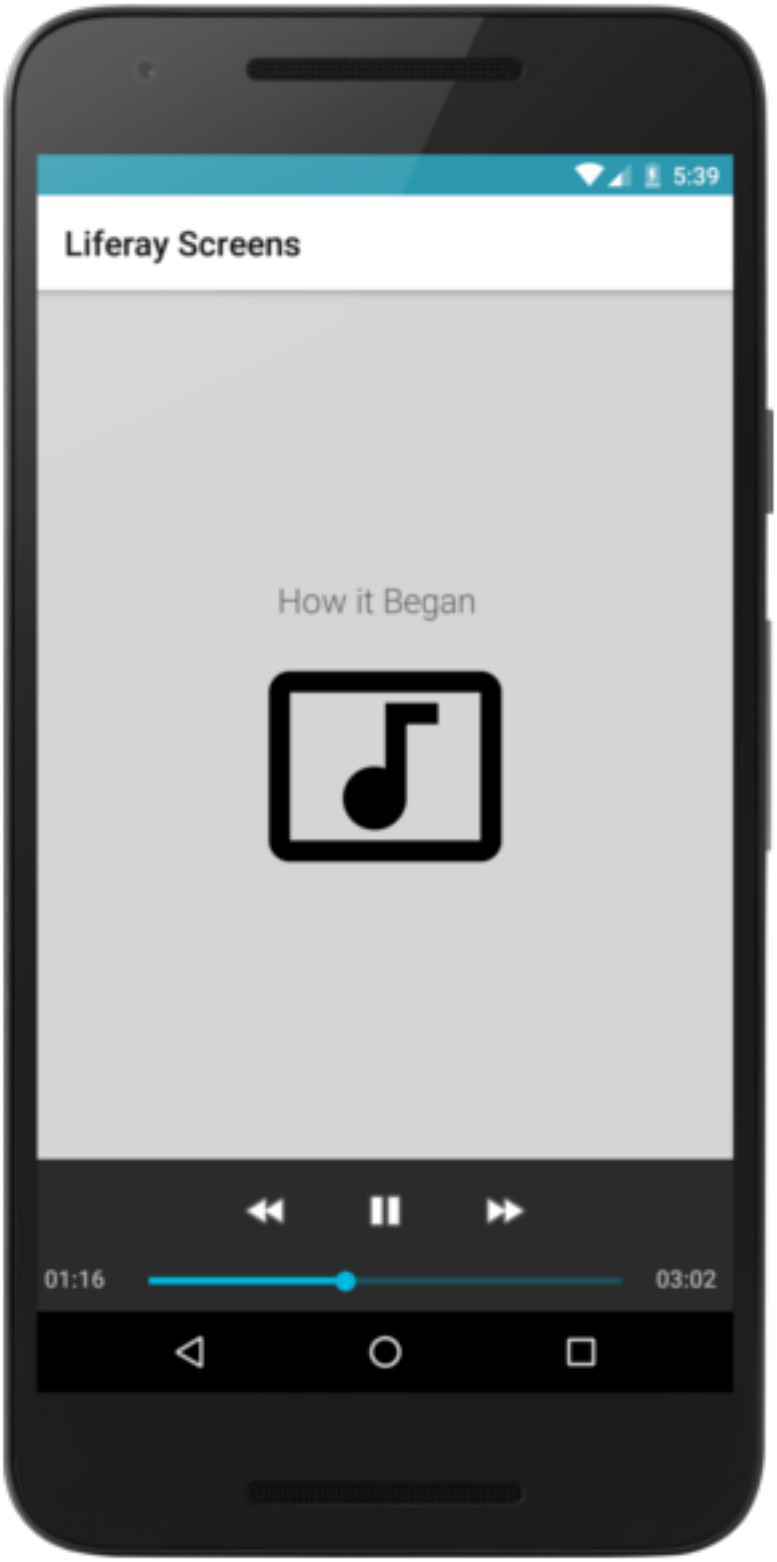


Figure 163.24: Audio Display Screenlet using the Default View.

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 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.

163.20 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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

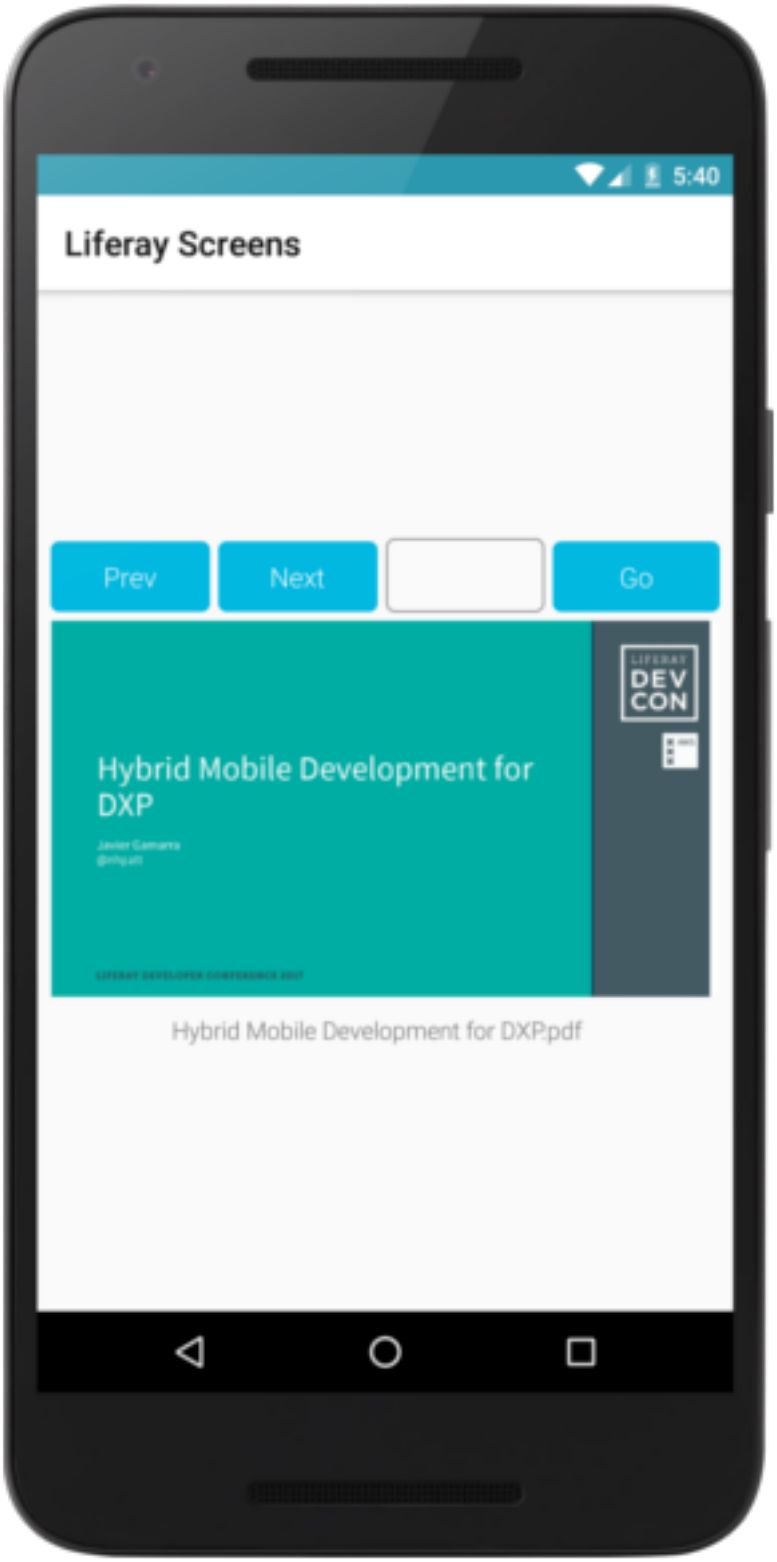


Figure 163.25: PDF Display Screenlet using the Default View.

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 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.

163.21 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

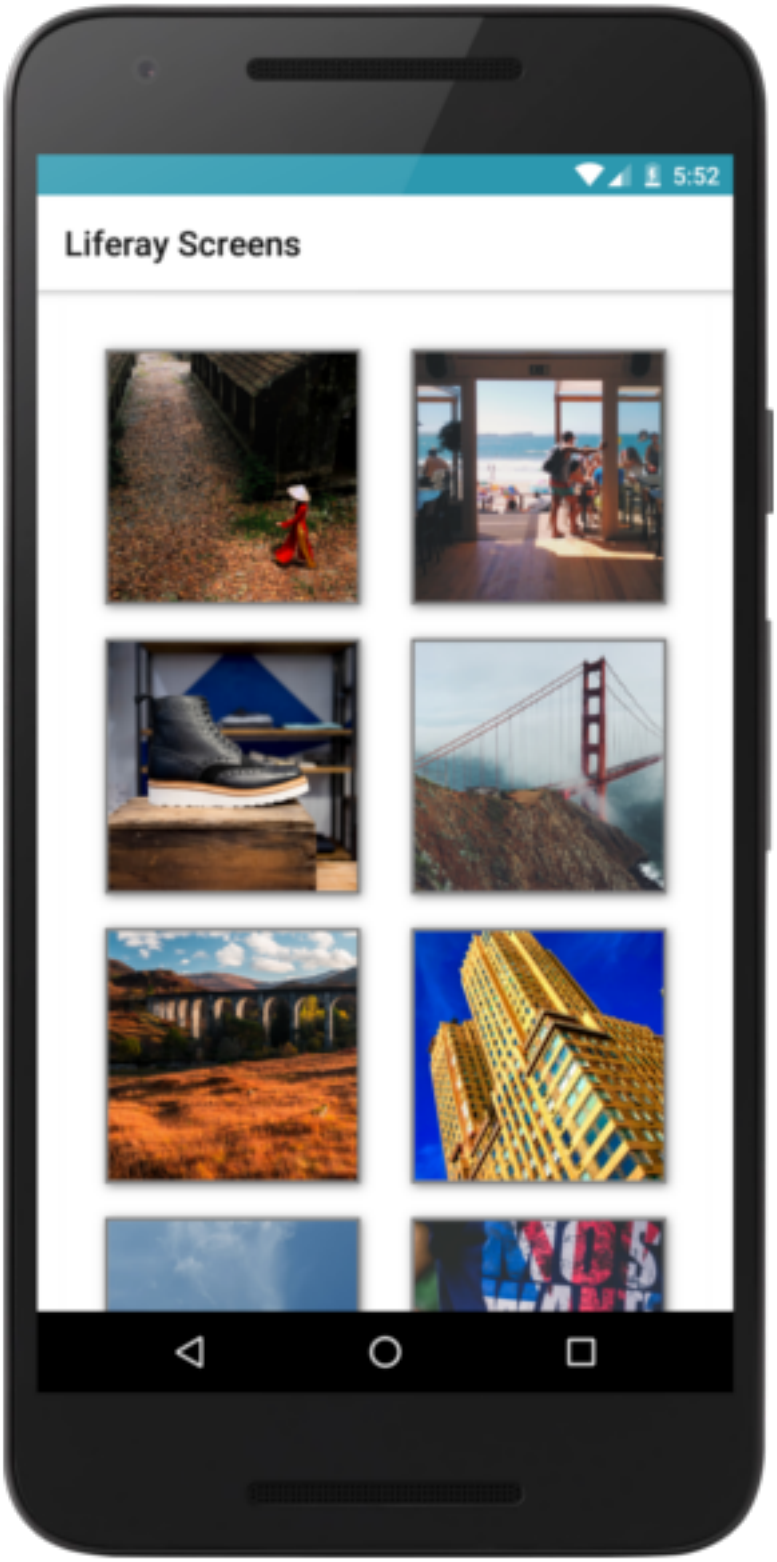


Figure 163.26: The Web Screenlet with the Default View Set.

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`.

- **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.

163.22 DDM Form Screenlet for Android

Requirements

- Android SDK 4.1 (API Level 21) or above
- Liferay DXP 7.1 SP2+
- Liferay Hypermedia REST APIs. These APIs are installed but disabled by default. To enable them, follow the instructions in the tutorial Enabling Hypermedia REST APIs.

Compatibility

- Android SDK 4.1 (API Level 21) or above

Xamarin Requirements

- Visual Studio 7.2
- Mono .NET framework 5.4.1.6

Features

DDM Form Screenlet shows a set of fields that can be filled in by the user. The fields can contain initial or existing values. The following fields are supported:

Paragraph: Add a title and/or text in your form.

Text Field: A single or multiline text area.

Single Selection: Select one item with a radio button.

Select From List: Choose one or more items in a list.

Multiple Selection: Select multiple items via a checkbox.

Date: Select a date from a date picker.

Grid: Select items in a matrix.

Numeric: Enter an integer or decimal number.

Upload: Upload files via Documents and Media.

DDM Form Screenlet also supports the following features:

Element Sets: Reuse pre-existing element sets in your form.

Multiple Pages: Use multi-page forms.

Success Page: Show friendly feedback at the end of your form. **Autosave:** Automatically save any change in form values to a draft.

Restore Previous Draft: Automatically restore the last draft when opening the form, independent of platform.

Rules: Create complex rules in your form. For example, you can show or hide fields depending on the input of other fields.

Workflow: Form submission can trigger a specific workflow.

Required Values: Require specific values and/or validate form fields.

Internationalization: Support i18n in record values and labels.

Module

- DDM

Views

- Default
- Lexicon
- Material

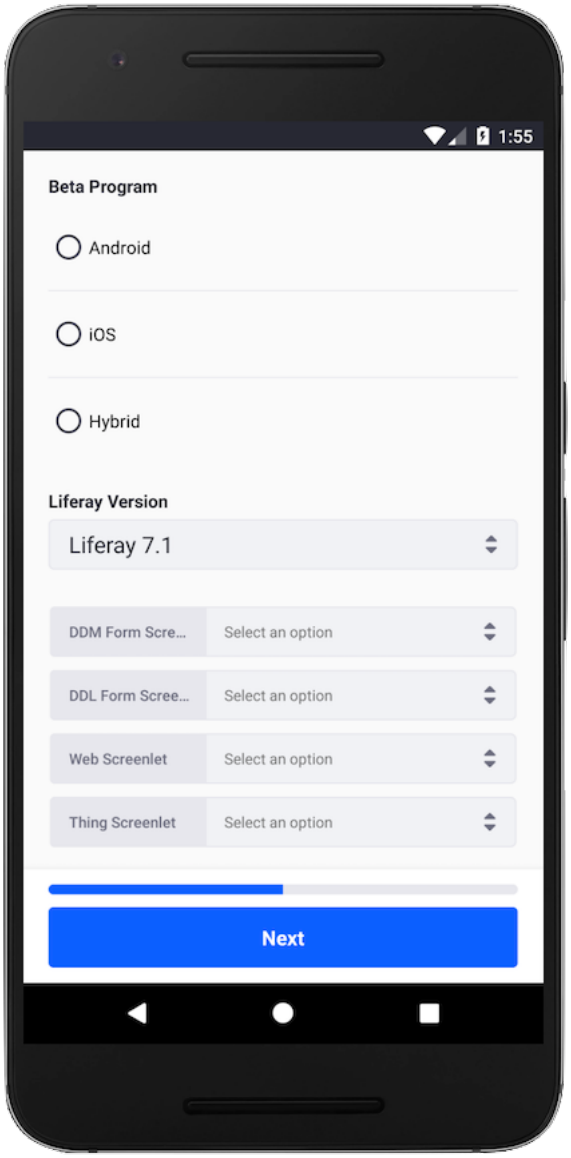


Figure 163.27: The DDM Form Screenlet with the Lexicon View Set.

Custom Layouts

To create custom layouts for a field, create the new layout following the naming pattern `<field_editor_id>_<view_name>`. The Screenlet automatically loads such layouts.

For example, this table lists the filename you should use when creating custom layouts for each field type, for the Lexicon View. Note that because some DDM fields inherit from DDL, they are referenced as DDL.

Editor Type
Field Editor ID
Example Using Lexicon View
Checkbox
ddlfield_checkbox
ddlfield_checkbox_lexicon.xml
Checkbox Multiple
ddmfield_checkbox
ddmfield_checkbox_multiple.xml
Date
ddlfield_date
ddlfield_date_lexicon.xml
Number
ddlfield_number
ddlfield_number_lexicon.xml
Integer
ddlfield_number
ddlfield_number_lexicon.xml
Decimal
ddlfield_number
ddlfield_number_lexicon.xml
Radio
ddlfield_radio
ddlfield_radio_lexicon.xml
Text
ddlfield_text
ddlfield_text_lexicon.xml
Select
ddlfield_select
ddlfield_select_lexicon.xml
Text Area
ddlfield_text_area
ddlfield_text_area_lexicon.xml
Paragraph
ddmfield_paragraph
ddmfield_paragraph_lexicon.xml
Document
ddlfield_document
ddlfield_document_lexicon.xml
Grid
ddmfield_grid
ddmfield_grid_lexicon.xml
Repeatable
ddmfield_repeatable
ddmfield_repeatable_lexicon.xml

Application Configuration

DDM Form Screenlet needs the following user permissions:

```
<uses-permission android:name="android.permission.CAMERA"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

The Documents and Media fields use both to take a picture/video and store it locally before uploading it to the portal.

Portal Configuration

Before using DDM Form Screenlet, ensure that the following exist in the portal:

- A form for the Screenlet to display. For instructions on this, see the article Creating and Managing Forms.
- If your form uses it, workflow must be configured. See the Workflow section of the user guide for instructions on configuring and using workflow.

Required Attributes

- `formInstanceId`

Attributes

Attribute
Data Type
Explanation
formInstanceId
number
The ID of the form to display in the Screenlet. To find the IDs for your data definitions in the portal, select the site to work in and click Content → Forms. The table that lists the site's forms also lists each form's ID.
layoutId
@layout
The layout to use to show the View.
autoloadDraftEnabled
boolean
Sets whether the form loads the last draft for the current user when the Screenlet is shown. The default value is `true`.
autosaveDraftEnabled
boolean
Sets whether the form should autosave a draft for the current user. The default value is `true`.
syncFormTimeout
number
Time in milliseconds to start synchronize the form (save and evaluate form rules). The default value is 500.

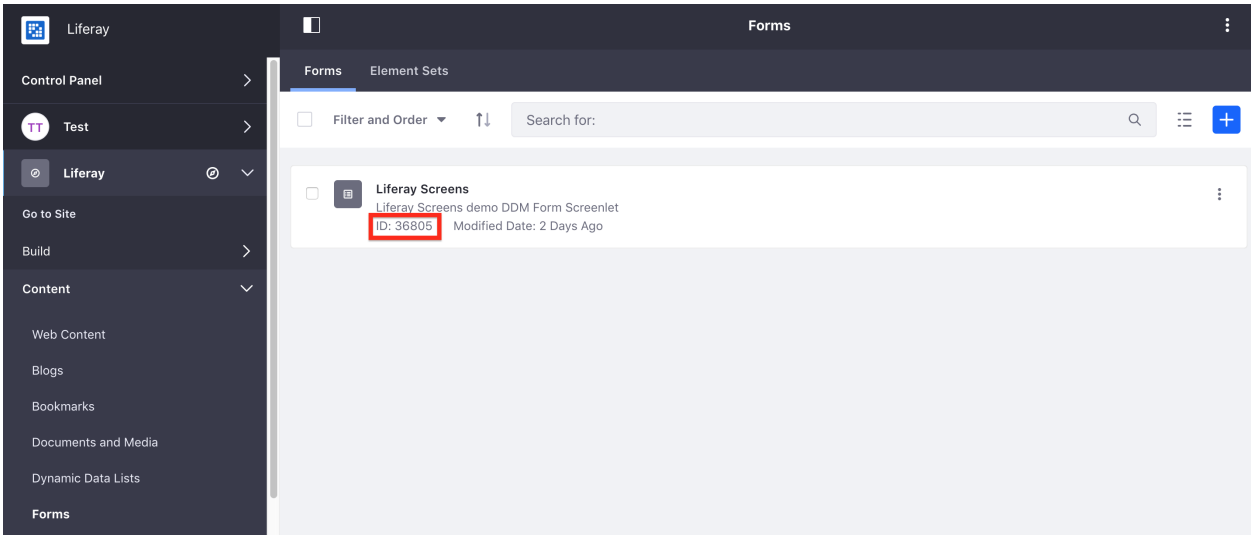


Figure 163.28: The red box in this image highlights a form's ID.

Permissions

If your form includes at least one Documents and Media field, you must grant permissions in the target repository and folder. For more information, see Granting File Permissions and Roles, and Setting Folder Permissions. To set permissions for Documents and Media's Home folder, navigate to Documents and Media and select *Options* ( ) → *Home Folder Permissions*.

Role	Permissions	Add Repository	Add Document Type	Add Shortcut	Update	Add Document	Subscribe	View	Add Folder	Add Metadata Set
Guest	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐
Owner	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑
Portal Content Reviewer	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Power User	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Site Content Reviewer	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Site Member	☐	☐	☐	☑	☐	☑	☑	☑	☑	☐
User	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

Figure 163.29: Select which roles can add a document to a Documents and Media folder.

Methods

Method
    Return Type
    Explanation
    load()

void
Starts the request to load the form. The form fields are shown when the response is received.
setDDMFormListener()
void
Sets the listener for this form.

Listener

DDM Form Screenlet delegates some events to an object that implements to the `DDMFormListener` interface. This interface lets you implement the following methods:

`onFormLoaded(FormInstance formInstance)`: Called when the form instance successfully loads.

`onError(Exception e)`: Called when an error occurs in the process. For example, this method is called when an error occurs while loading a form instance.

`onDraftLoaded(FormInstanceRecord formInstanceRecord)`: Called when a draft is retored.

`onDraftSaved(FormInstanceRecord formInstanceRecord)`: Called when a draft is saved.

`onFormSubmitted(FormInstanceRecord formInstanceRecord)`: Called when a form is successfully submitted.

CHAPTER 164

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 the 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 blog entry.
- **Image Display Screenlet:** Shows a single image file from the Documents and Media Library.
- **Video Display Screenlet:** Shows a single video file from the Documents and Media Library.
- **Audio Display Screenlet:** Shows a single audio file from the Documents and Media Library.
- **PDF Display Screenlet:** Shows a single PDF file from the Documents and Media Library.
- **File Display Screenlet:** Shows a single file from the 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.

164.1 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 OAuth 2.
- **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.

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, see the tutorial Using Themes in iOS Screenlets.

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. For more information, see the User Guide's authentication section.

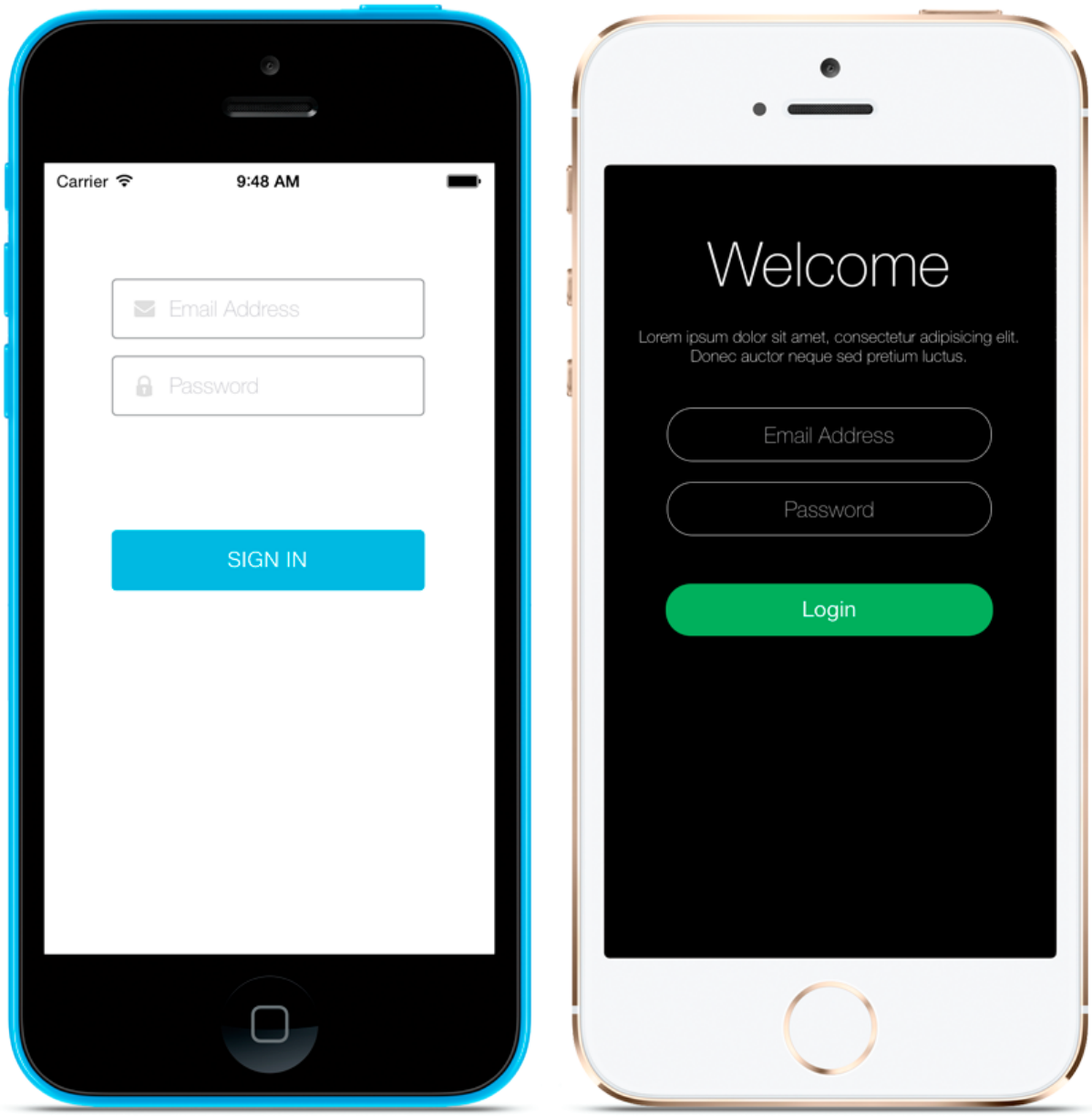


Figure 164.1: The Login Screenlet using the Default and Flat7 Themes.

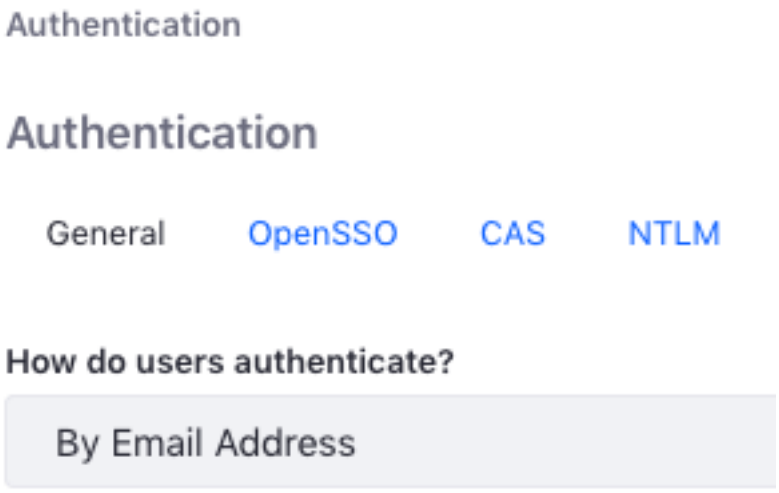


Figure 164.2: Setting the authentication method in your Liferay instance.

OAuth

For instructions on using OAuth with Login Screenlet, see the tutorial on using OAuth 2 with Liferay Screens.

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`, `cookie`, `oauth2Redirect`, or `oauth2UsernameAndPassword`. 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 one of the OAuth values or `cookie`, the Screenlet defaults to basic authentication with the `email` option. | `oauth2clientId` | `string` | The ID of the OAuth 2 application in the portal. You can find this value in the portal's OAuth 2 Admin portlet. | `oauth2redirectUrl` | `string` | The URL that the mobile browser will redirect the user to after successful login. You must configure this in the portal's OAuth 2 Admin portlet, and associate the URL with the iOS app. | `oauth2clientSecret` | `string` | The client secret of the OAuth 2 application in the portal. You can find this value in the portal's OAuth 2 Admin portlet. | `oauth2Scopes` | `string` | The portal permissions to request. You can define a set of permissions associated with an OAuth 2 application in the portal's OAuth 2 Admin portlet. Use this attribute to request a subset of those permissions. Separate multiple scopes with a space (e.g.,

"scope1 scope2 scope3"). | 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))
    }

}
```

164.2 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

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`)

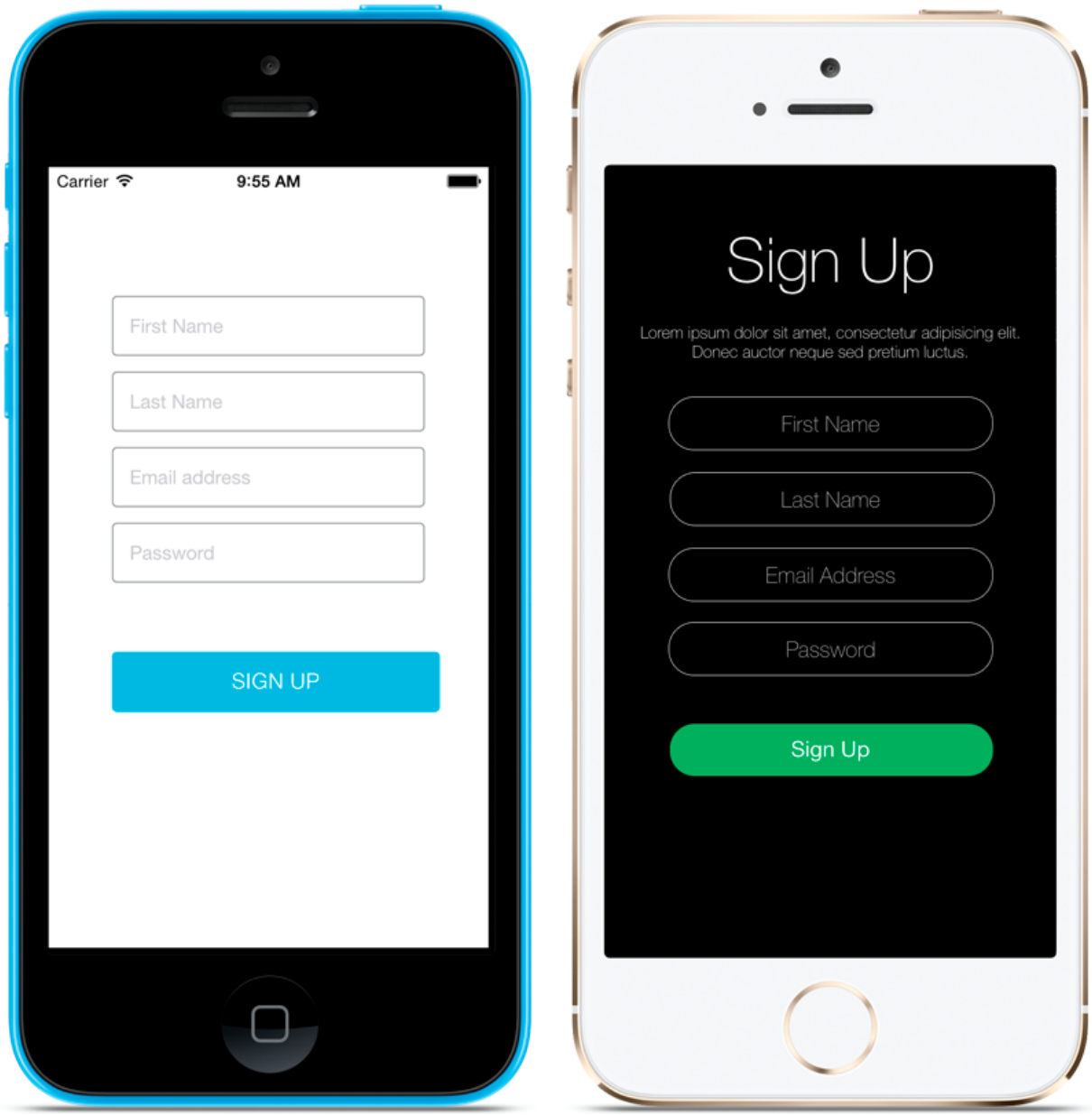


Figure 164.3: The Sign Up Screenlet with the Default and Flat7 Themes.

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.

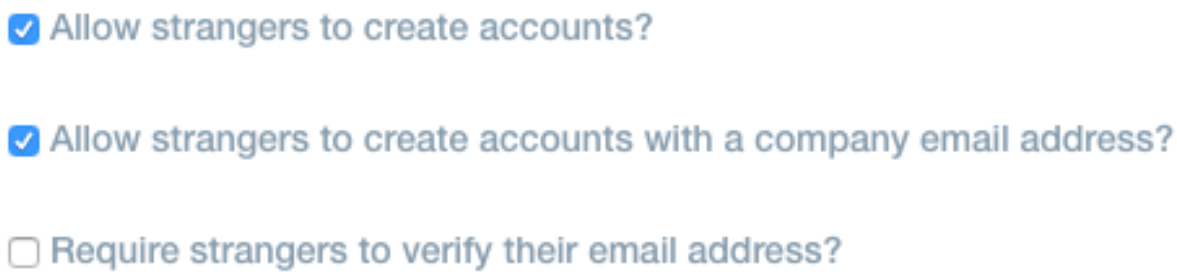


Figure 164.4: The Liferay instance's authentication settings.

For more details, see the Authentication 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 the attributes in the portal's `User` entity.

- `- screenlet:onSignUpError:`: Called when an error occurs in the process. The `NSError` object describes the error.

164.3 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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`)

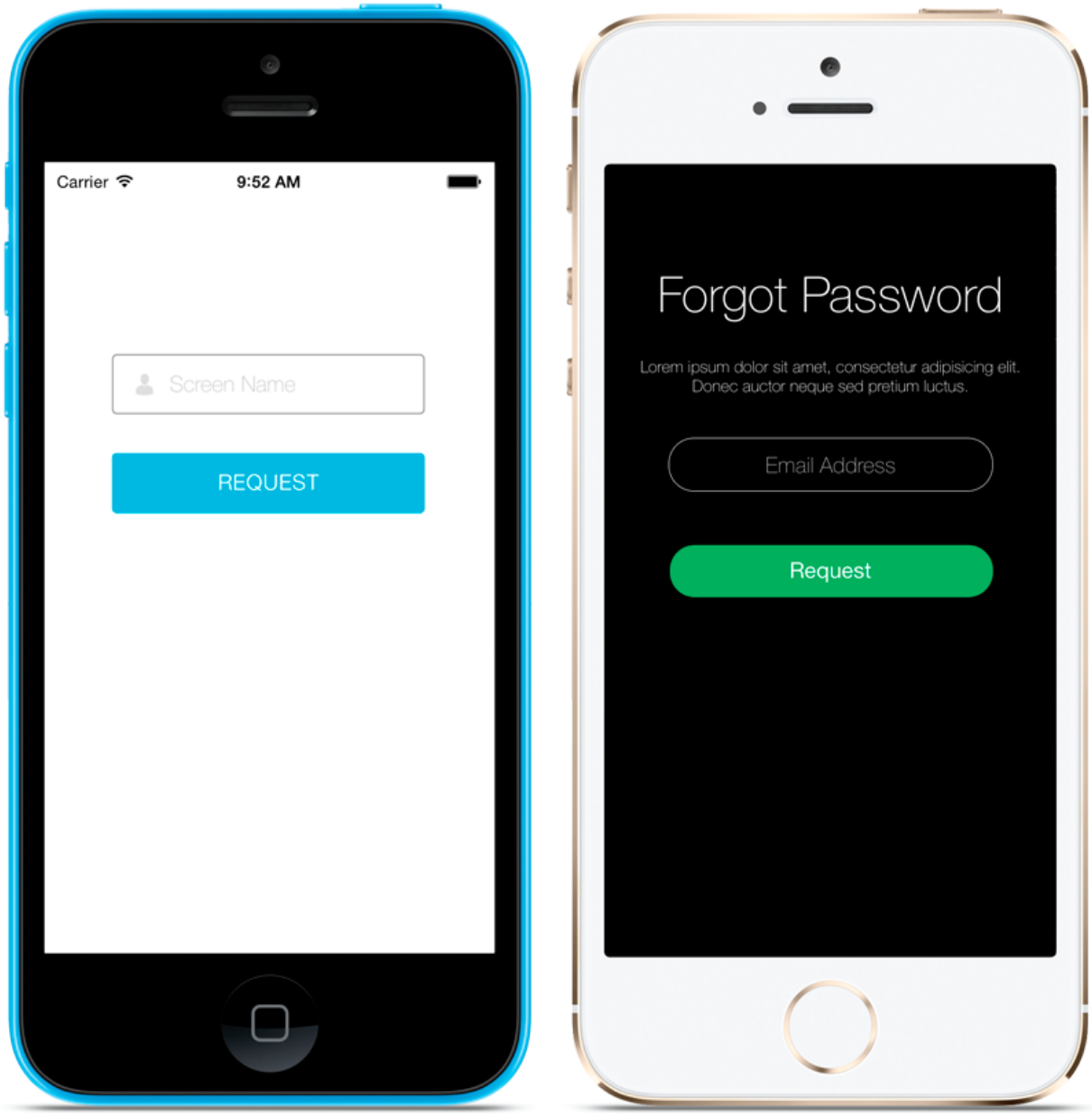


Figure 164.5: The Forgot Password Screenlet with the Default and Flat7 Themes.

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:

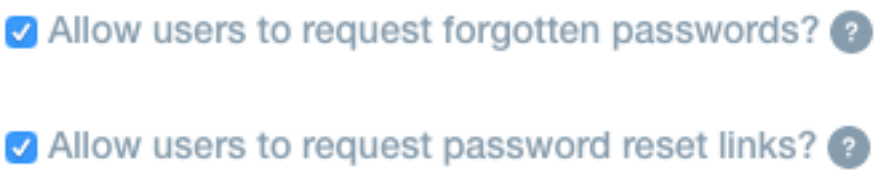


Figure 164.6: Checkboxes for the password recovery features in Liferay Portal.

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, see the Authentication 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.

164.4 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

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`	

Service	Method	Notes

Module

- None

Themes

- Default (`default`)
- Flat7 (`flat7`)

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

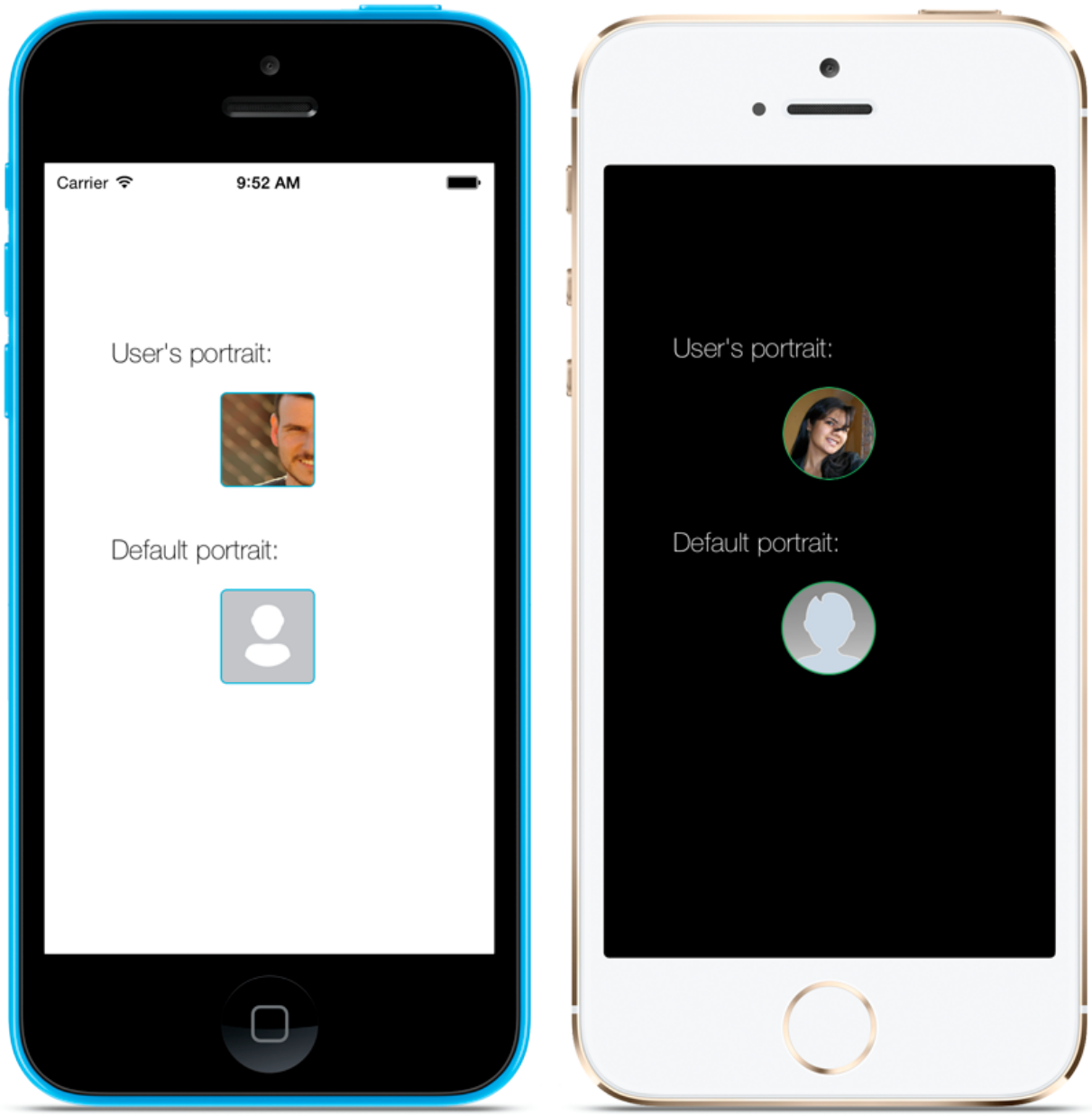


Figure 164.7: The User Portrait Screenlet using the Default and Flat7 Themes.

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.

164.5 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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
`Screensddlrecord Service` (Screens compatibility plugin)	`getDdlRecord`	Load record
`DLAppService`	`addFileEntry`	Upload document
`DDLRecordService`	`addRecord`	Submit form
`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.

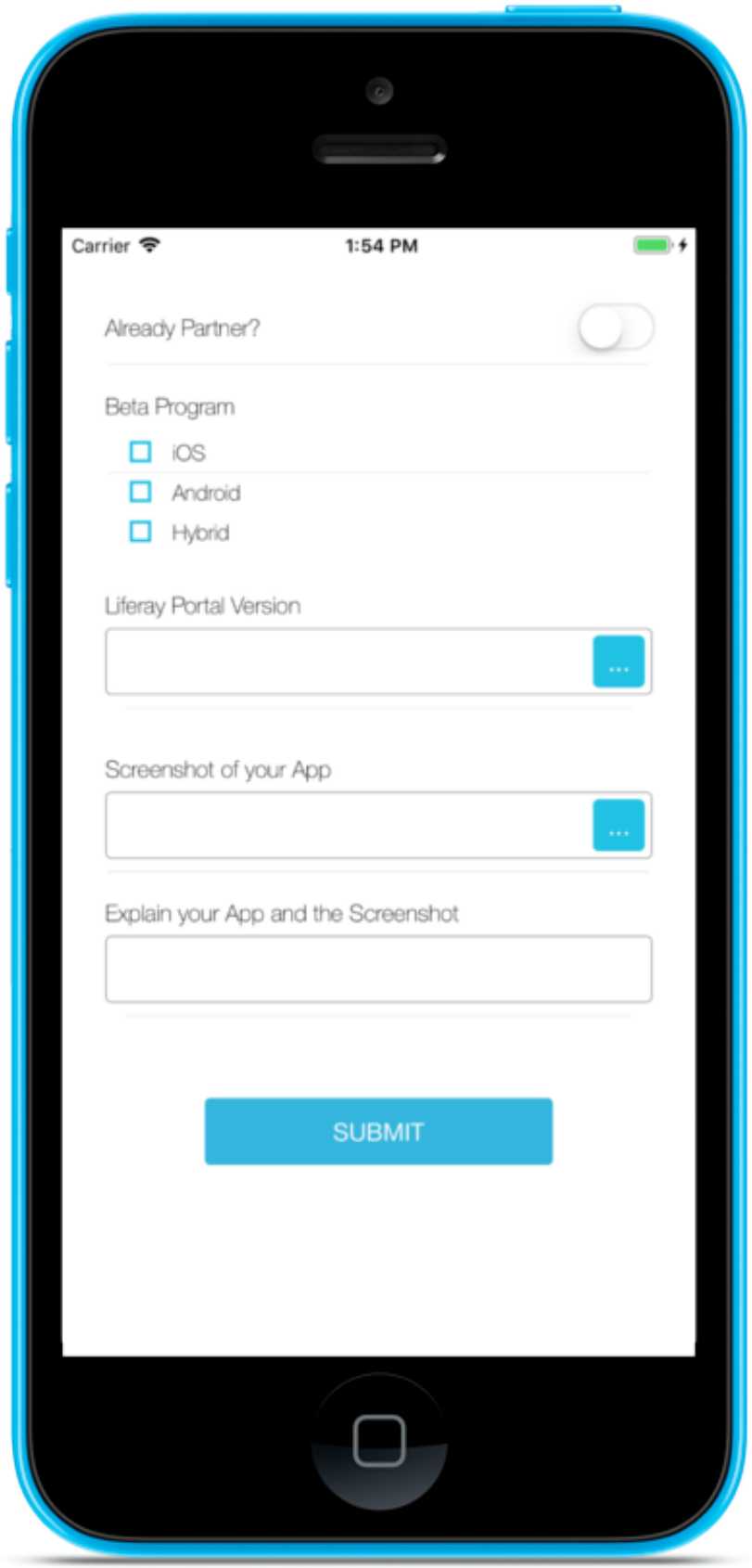


Figure 164.8: DDL Form Screenlet using the Default (`default`) Theme.

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:

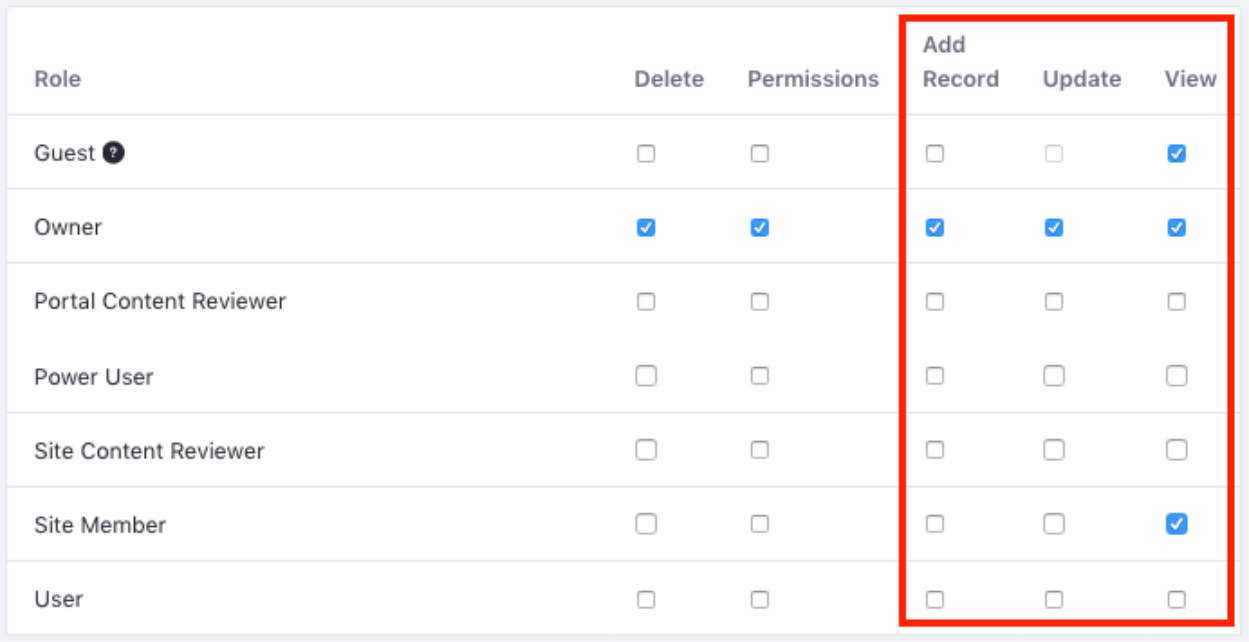


Figure 164.9: The permissions for adding, viewing, and editing DDL records.

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, please 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:

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

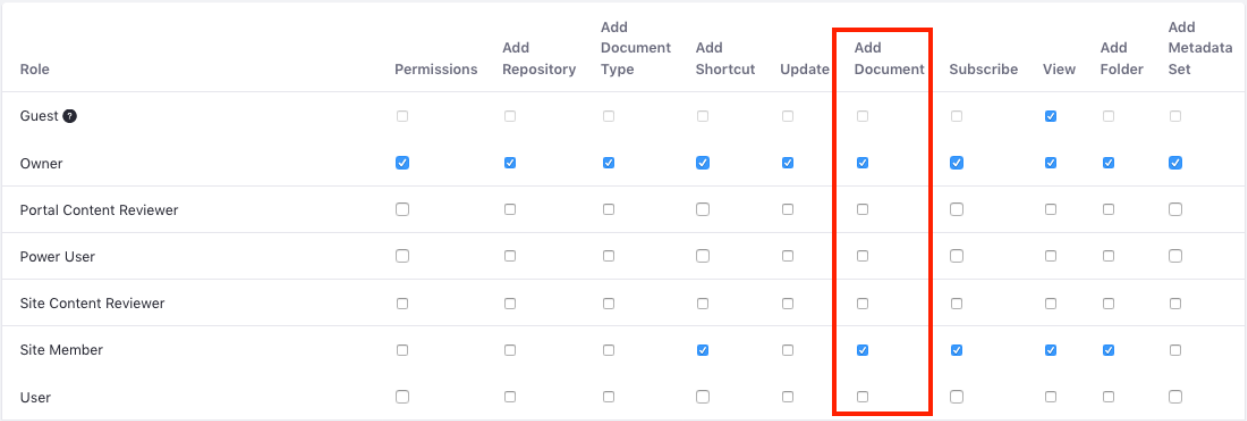


Figure 164.10: The permission for adding a document to a Documents and Media folder.

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 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.

164.6 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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
`Screensddlrecord`Service (Screens compatibility plugin)	`getDdlRecords`	With `ddlRecordSetId`, or `ddlRecordSetId` and `userId`
`Screensddlrecord`Service (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:

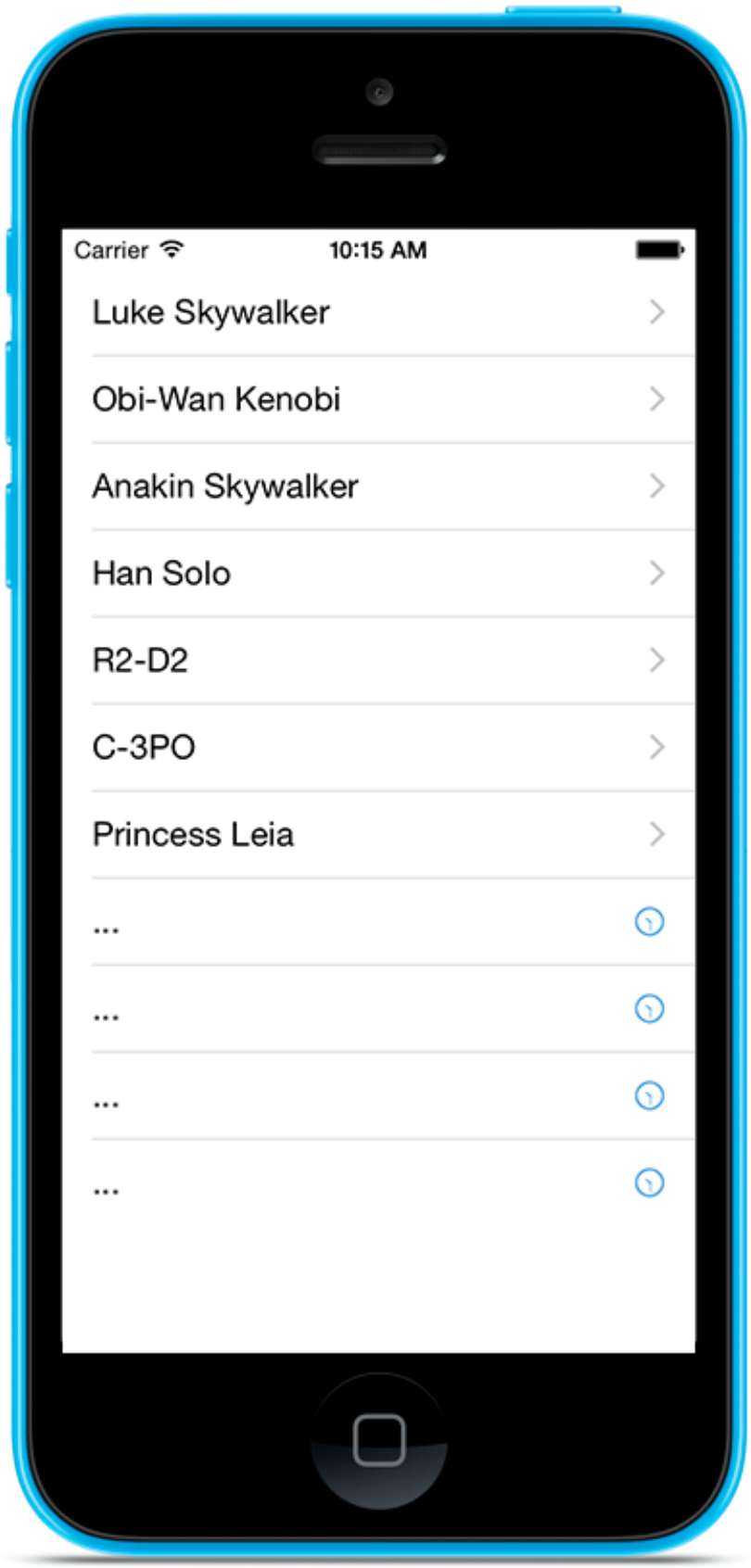


Figure 164.11: The DDL List Screenlet using the Default (`default`) Theme.

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.

164.7 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

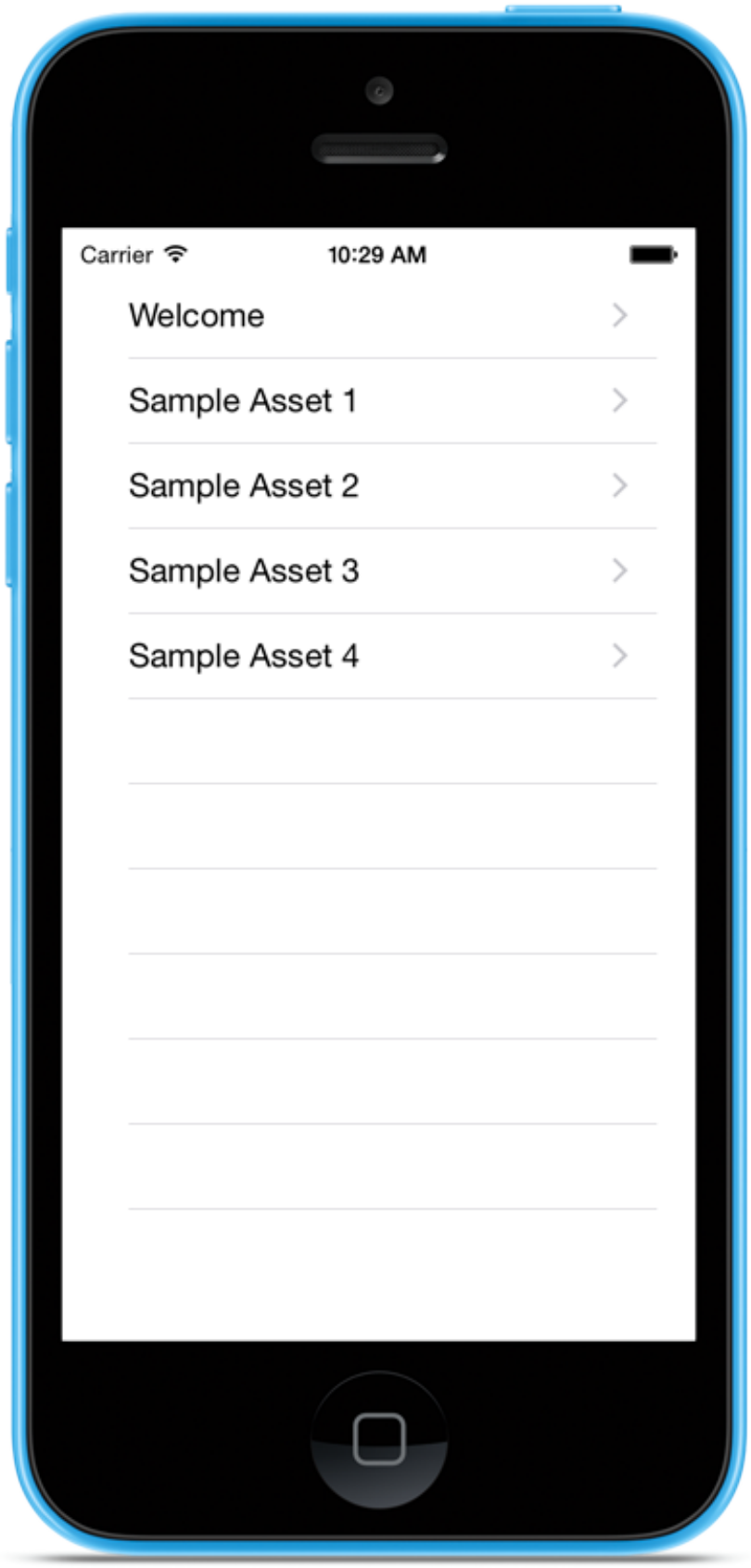


Figure 164.12: Asset List Screenlet using the Default (`default`) Theme.

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 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.

164.8 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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`	
`Screensddlrecord Service` (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 web content section of the 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

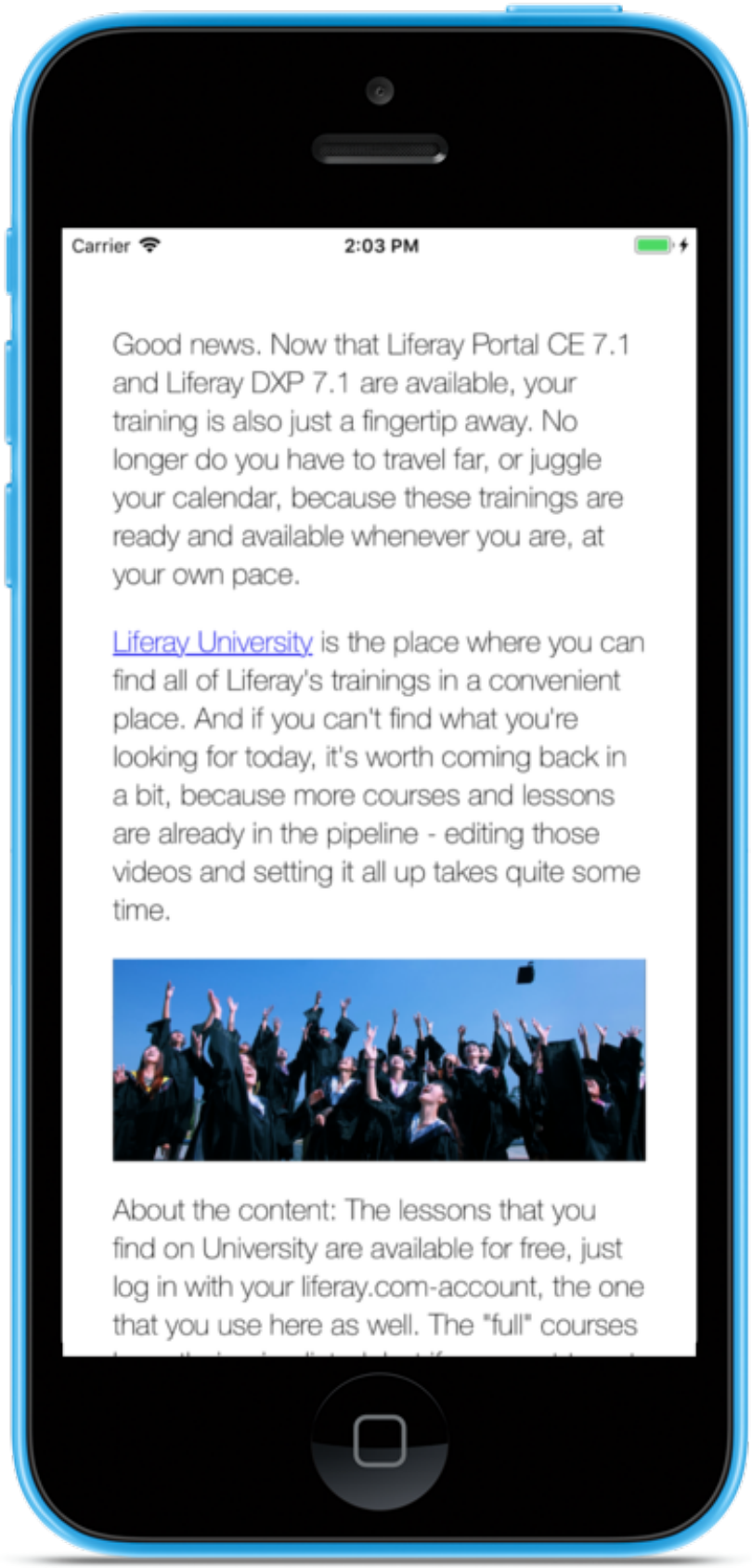


Figure 164.13: The Web Content Display Screenlet using the Default (`default`) Theme

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`

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.

164.9 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

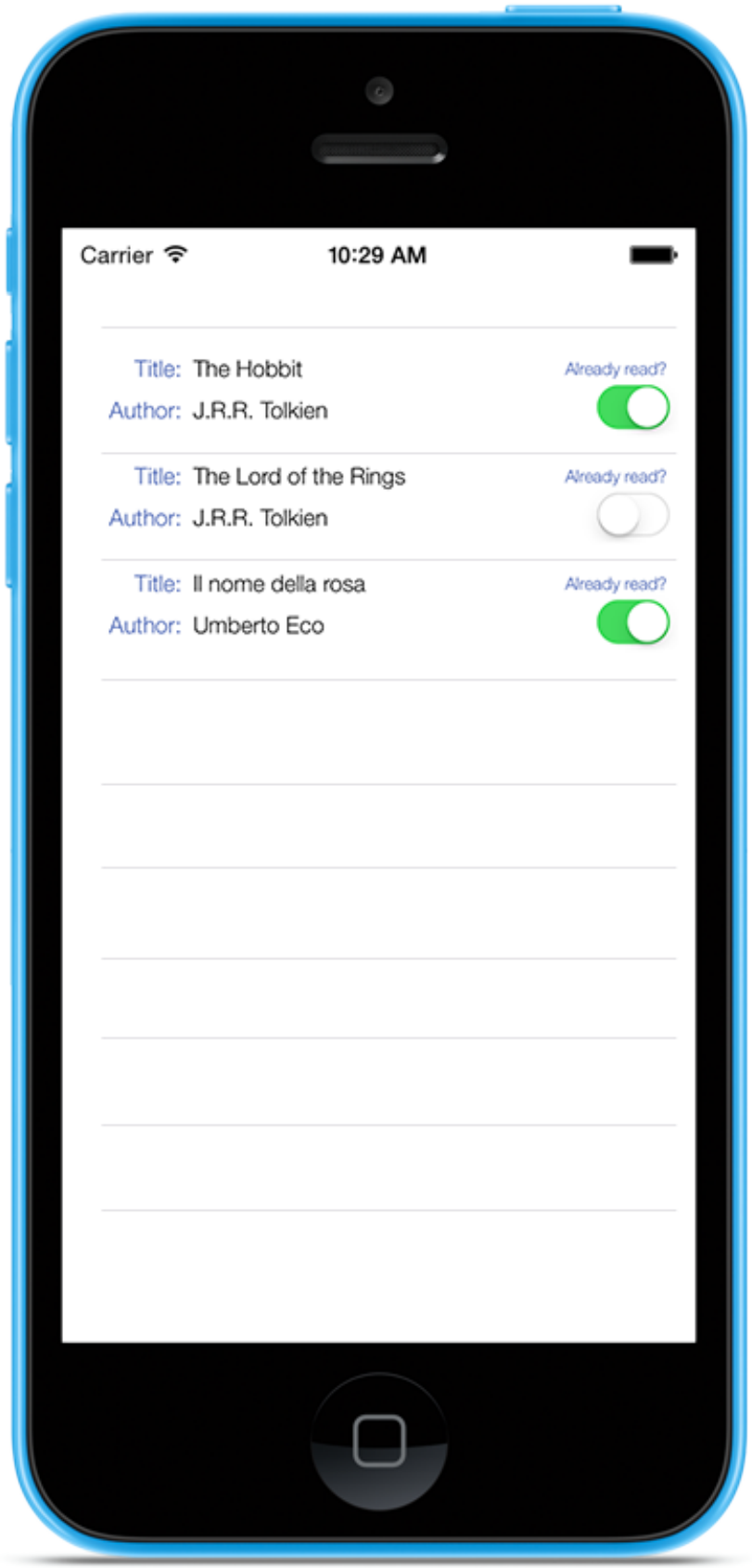


Figure 164.14: Web Content List Screenlet using the Default (`default`) Theme.

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 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. |

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.

164.10 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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

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

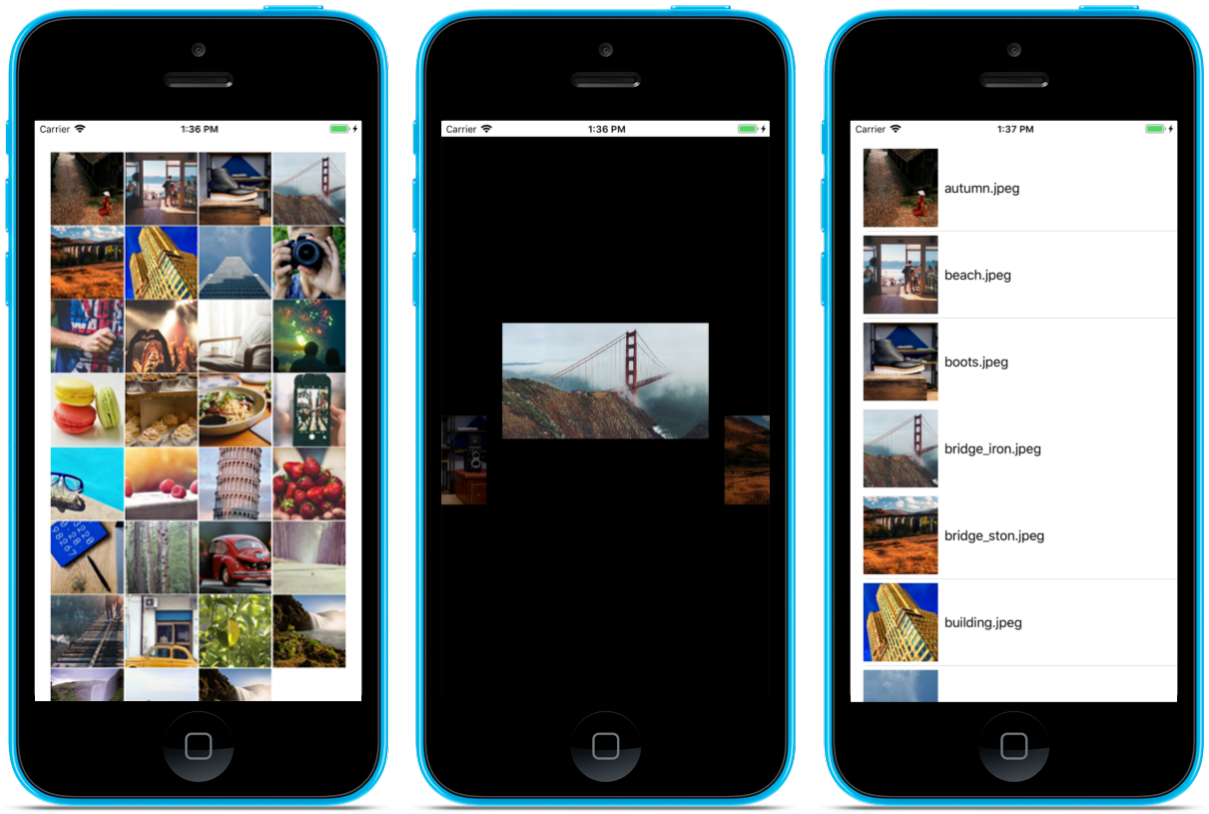


Figure 164.15: Image Gallery Screenlet using the Grid, Slideshow, and List Themes.

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
}
```

164.11 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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

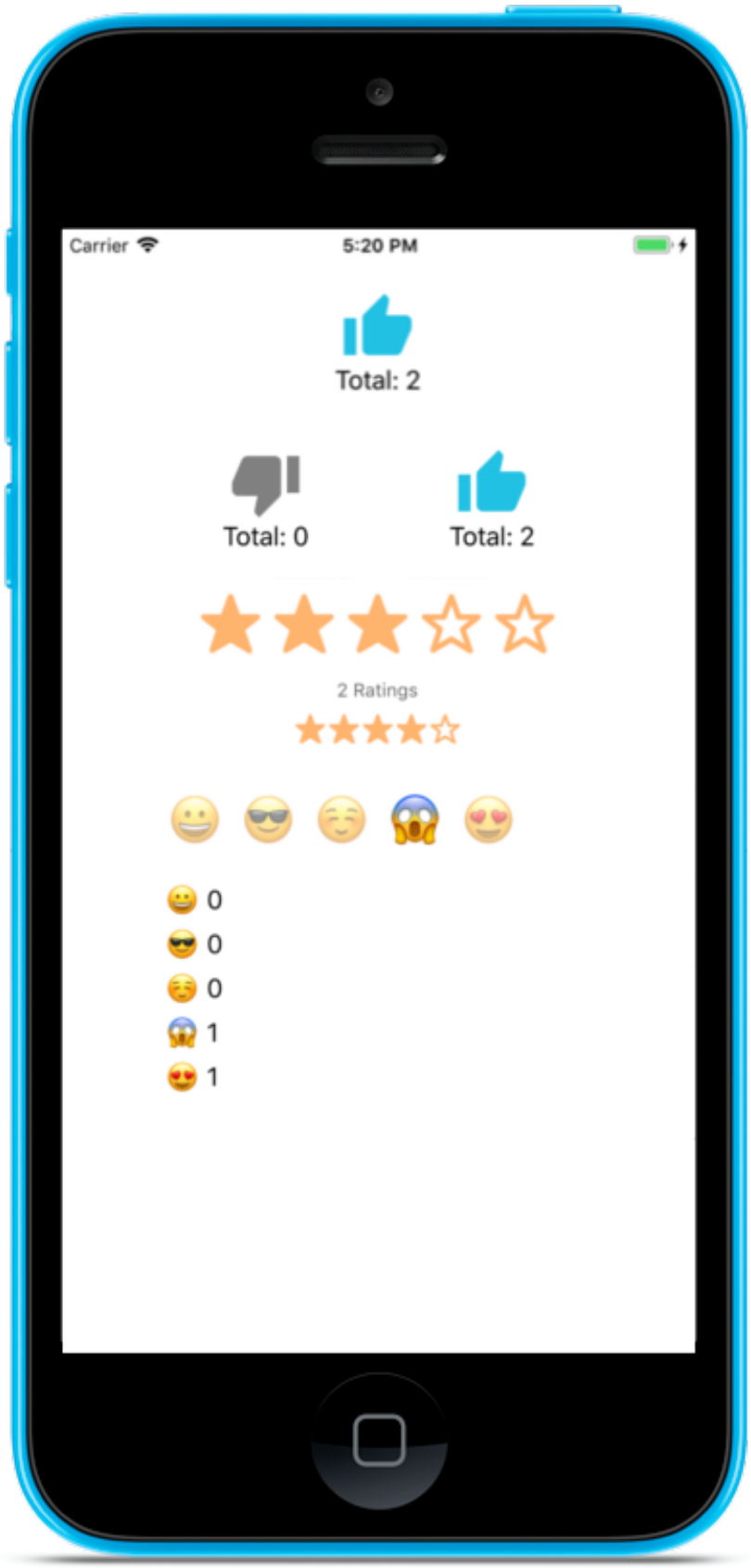


Figure 164.16: Rating Screenlet's different Themes.

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` | 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.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.

164.12 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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. |

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.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` |

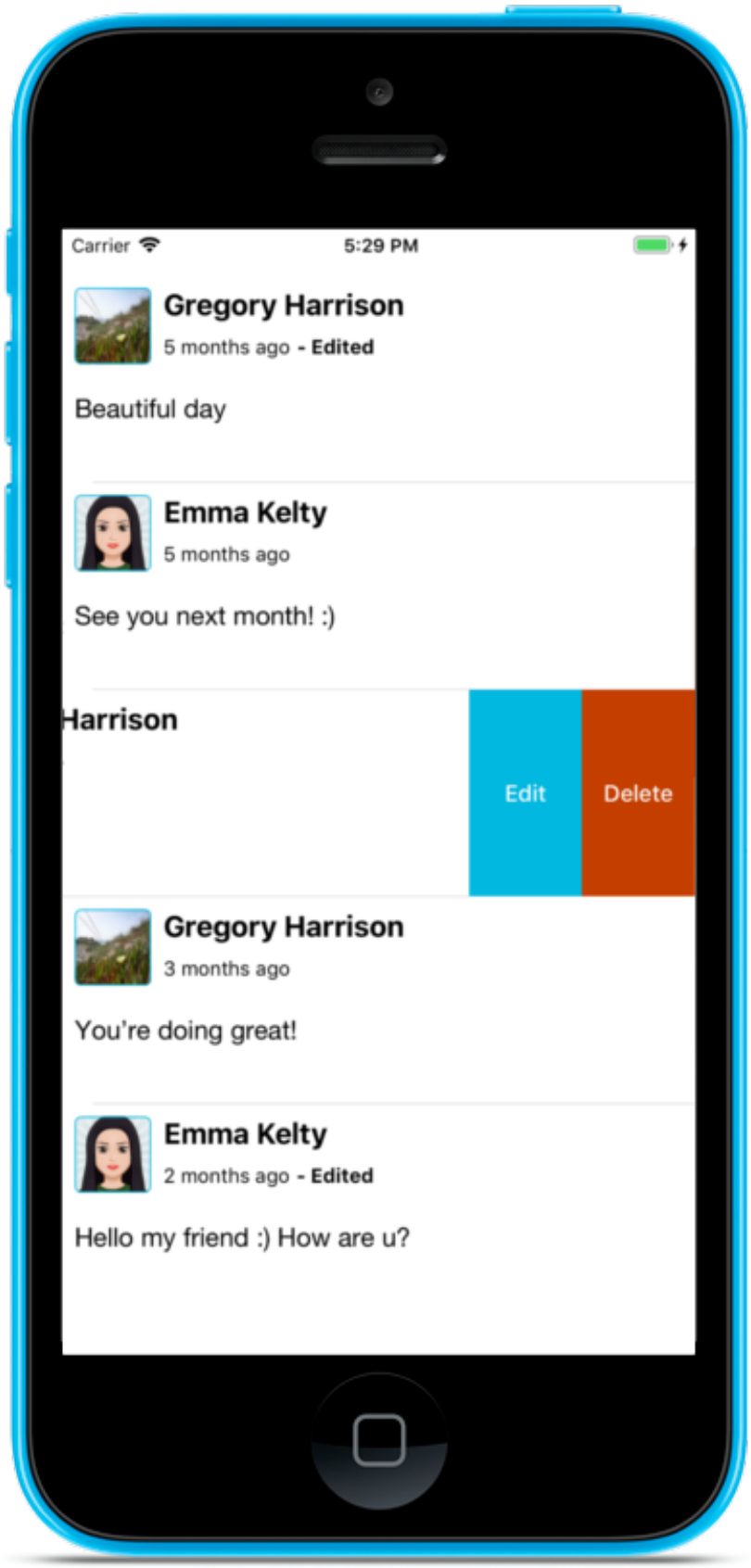


Figure 164.17: Comment List Screenlet using the Default Theme.

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.

164.13 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

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

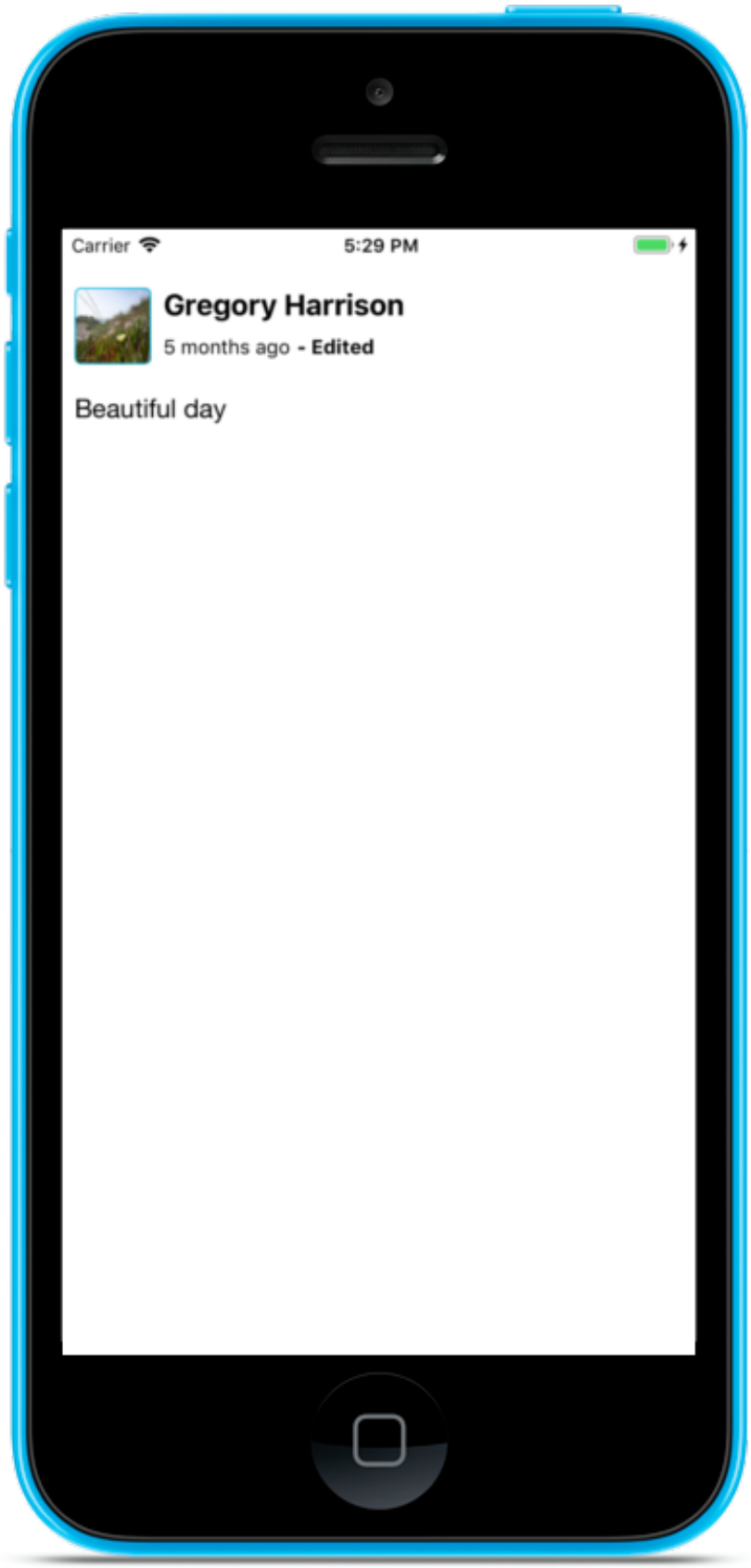


Figure 164.18: Comment Display Screenlet using the Default Theme.

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.

164.14 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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. |

Required Attributes

- `className`
- `classPK`

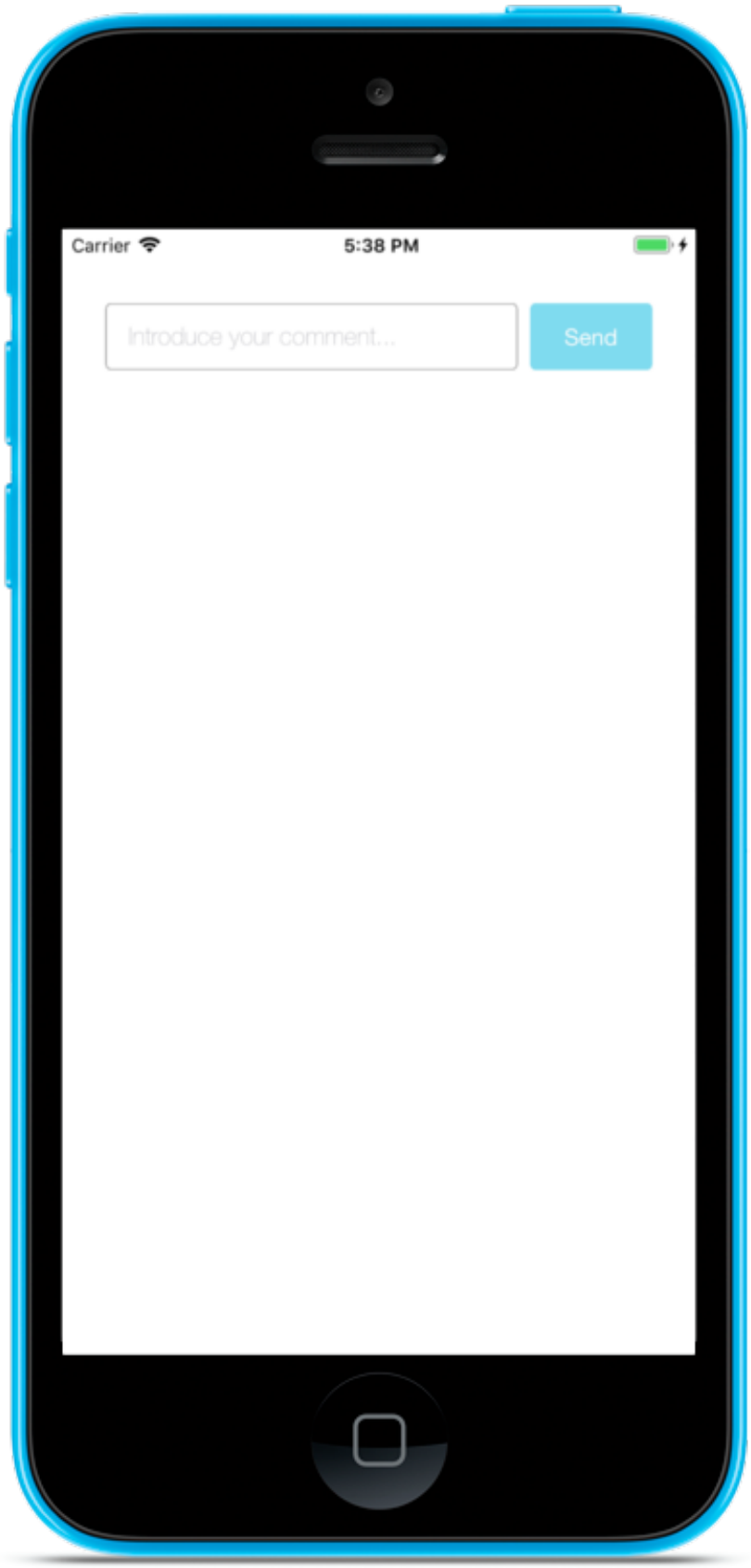


Figure 164.19: Comment Add Screenlet using the Default Theme.

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.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.

164.15 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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 164.20: 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.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
}
```

164.16 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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:

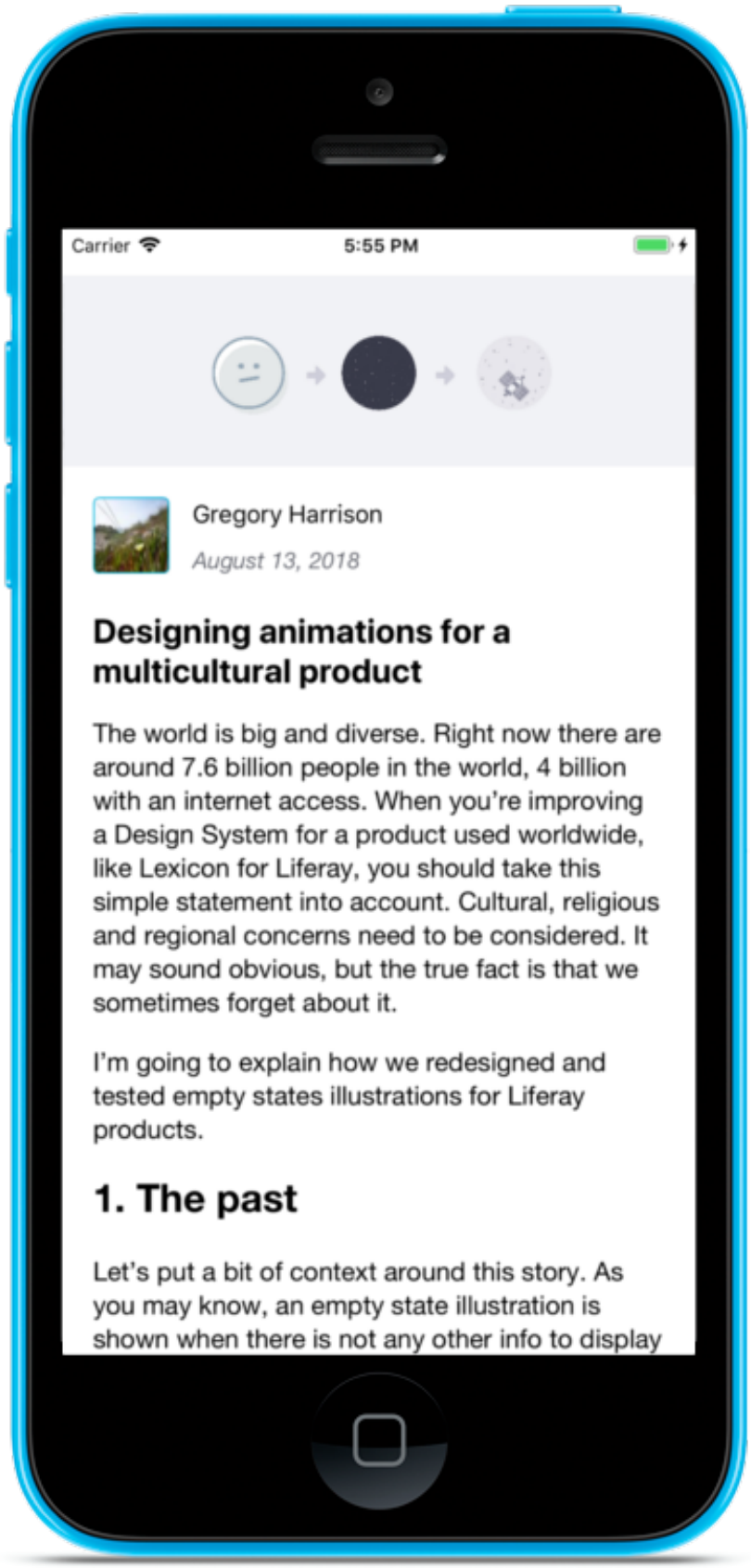


Figure 164.21: Blogs Entry Display Screenlet using the Default Theme.

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.

164.17 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

Figure 164.22: Image 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 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.

164.18 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

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

Figure 164.23: Video Display Screenlet using the Default Theme.

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.

164.19 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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. |

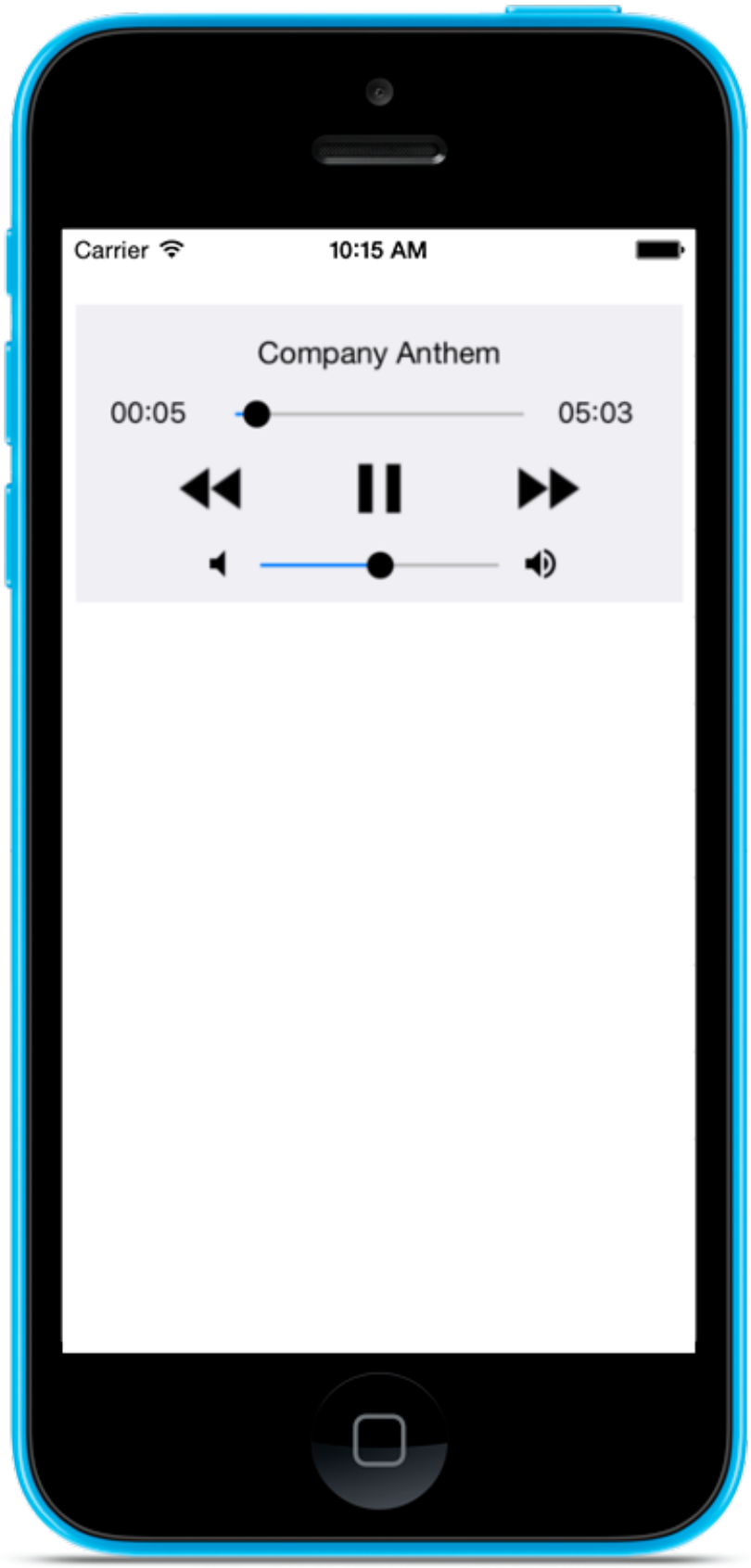


Figure 164.24: Audio Display Screenlet using the Default Theme.

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. |

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.

164.20 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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

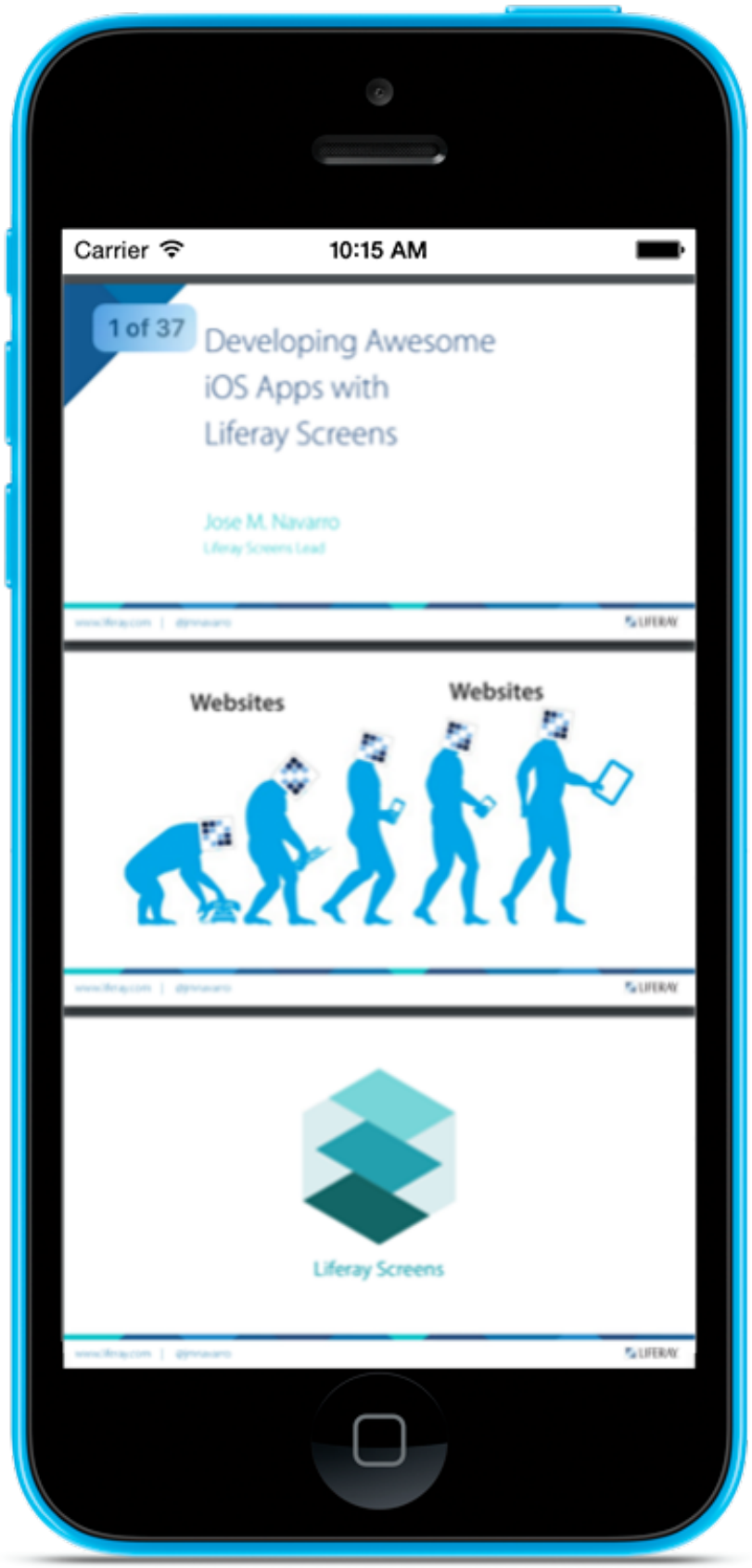


Figure 164.25: PDF Display Screenlet using the Default Theme.

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` | `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.

164.21 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
- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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`
`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:

Figure 164.26: File Display Screenlet using the Default View.

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. 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.

164.22 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

- Liferay Screens Compatibility app (CE or EE/DXP). This app is preinstalled in Liferay CE Portal 7.0/7.1 and Liferay DXP.

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.

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.

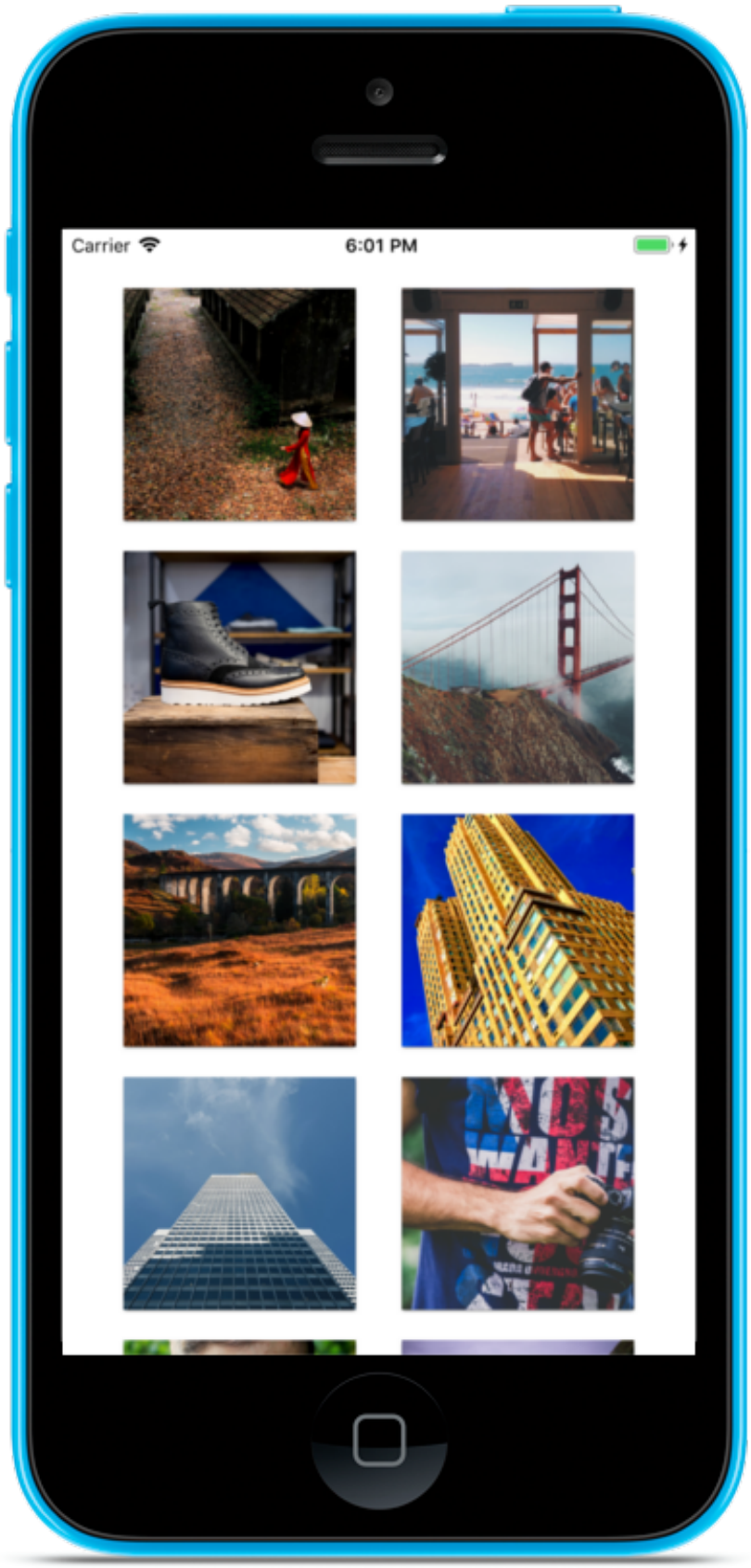


Figure 164.27: Web Screenlet using the Default Theme.

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) {
    ...
}
```

164.23 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 165

THEMES

Theme development is a multistep process, involving many tools and endless possibilities. This section of reference docs provides the following helpful information for theme development:

- A Theme anatomy reference guide

165.1 Theme Reference Guide

A theme is made up of several files. Although most of the files are named after their matching components, their functions may be unclear. This reference guide explains each file's usage to make clear which files to modify.

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 Dev Studio DXP.

Themes developed with the theme generator have the anatomy shown below. Although themes developed with Dev Studio DXP have a slightly different anatomy built with the theme project template, the core theme files are the same.

- `theme-name/`
    - `src/`
        - `css/`
            - `_clay_custom.scss`
            - `_clay_variables.scss`
            - `_custom.scss`
            - `_liferay_variables_custom.scss`
        - `images/`
            - (custom images)

        * js/

            - main.js

        * templates/

            - init_custom.ftl
            - navigation.ftl
            - portal_normal.ftl
            - portal_pop_up.ftl
            - portlet.ftl

        * WEB-INF/

            - liferay-look-and-feel.xml
            - liferay-plugin-package.properties
            - src/
            - resources-importer/
            - (Many directories)

    - liferay-theme.json
    - package.json

Regarding CSS files, you should only modify `_clay_custom.scss`, `_clay_variables.scss`, `_custom.scss`, and `_liferay_variables_custom.scss`.

You can of course overwrite any CSS file you want, but if you modify any other files, you're removing styling that 7.0 needs to work properly.

Theme Files

_clay_custom.scss

Used for Clay 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 `_clay_variables.scss`.

_clay_variables.scss

Used to store custom Sass variables. This file gets injected into the Bootstrap/Lexicon build, so you can overwrite variables and change how those libraries are compiled.

_custom.scss

Used for custom CSS styles. You should place all of your custom CSS modifications in this file.

_liferay_variables_custom.scss

Used for overwriting variables defined in `_liferay_variables.scss` without wiping out the whole file.

init_custom.ftl

Used for custom FreeMarker variables i.e. theme setting variables.

navigation.ftl

The theme template for the theme's navigation.

portal_normal.ftl

Similar to a static site's `index.html`, this file acts as a hub for all 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.

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 language, version, and base theme, for Node.js tool developed themes. You can update this file 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 configuration settings are also placed in this file. For a full explanation of the properties available for this file please see the 7.1 Properties documentation.

165.2 Theme Components and Workflow

If you want to develop a website, you must have three key components: CSS, JavaScript, and HTML. Liferay DXP supports SASS as well as multiple JavaScript frameworks. The HTML, however, is rendered via FreeMarker theme templates. This reference guide provides an overview of Liferay DXP's theme development components and workflow, covering the following topics:

- Theme templates
- Theme customizations and extensions
- Portlet customizations and extensions
- Theme workflow
- CSS Frameworks and extensions

Theme Templates

Liferay DXP provides several default FreeMarker templates that each handle a key piece of functionality for the page:

- `portal_normal.ftl`: Similar to a static site's `index.html`, this file acts as a hub for all theme templates and provides the overall markup for the page.

- `init.ftl`: Contains common FreeMarker variables that can be used in your theme templates. Useful for reference if you need access to theme objects. **We recommended that you DO NOT override this file**.

- `init_custom.ftl`: Used to override FreeMarker variables in `init.ftl` and to define new variables, such as theme settings.

- `portlet.ftl`: This template controls the theme's portlets. If your theme uses Portlet Decorators, you can modify this file to create application decorator-specific theme settings. See the Portlet Decorators tutorial for more info.

- `navigation.ftl`: Contains the navigation markup. To customize pages in the navigation, you must use the `liferay.navigation_menu` macro. Then you can leverage ADTs for the navigation menu. Note that `navigation.ftl` also defines the hamburger icon and `navbar-collapse` class that provides the simplified navigation toggle for mobile viewports, as shown in the snippet below for the Classic theme:

```
<#if has_navigation && is_setup_complete>
  <button aria-controls="navigationCollapse" aria-expanded="false"
  aria-label="Toggle navigation" class="navbar-toggler navbar-toggler-right"
  data-target="#navigationCollapse" data-toggle="collapse" type="button">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div aria-expanded="false" class="collapse navbar-collapse" id="navigationCollapse">
    <@liferay.navigation_menu default_preferences="${preferences}" />
  </div>
</#if>
```

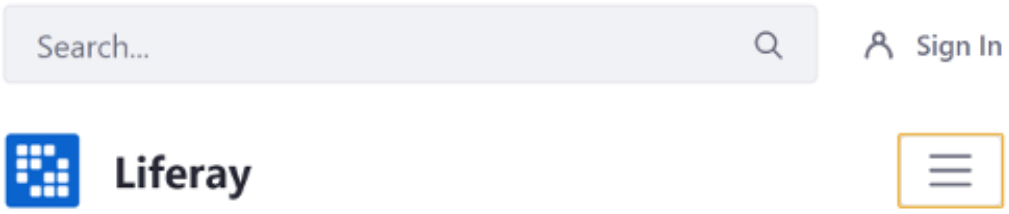


Figure 165.1: The collapsed navbar provides simplified user-friendly navigation for mobile devices.

- `portal_pop_up.ftl`: The theme template controlling pop up dialogs for the theme's portlets. Similar to `portal_normal.ftl`, `portal_pop_up.ftl` provides the markup template for all pop-up dialogs, such as a portlet's Configuration menu. It also has access to the FreeMarker variables defined in `init.ftl` and `init_custom.ftl`.

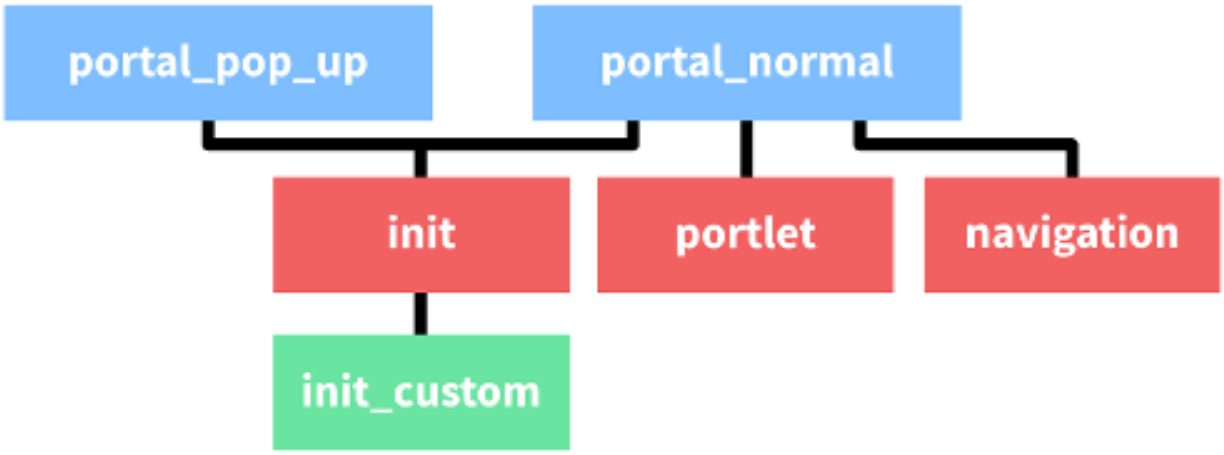


Figure 165.2: Each theme template provides a portion of the page's markup and functionality.

Theme Template Utilities

Liferay DXP provides several FreeMarker variables and macros that you can use in your theme templates to include portlets, use taglibs, access theme objects, and more. You can see examples of these in `portal_normal.ftl`. These utilities are included in the files listed below:

- `Init.ftl`: Provides access to common theme variables
- `FTL_Liferay.ftl`: Provides macros for commonly used portlets and theme resources. See the Macros tutorial for more information.
- `taglib-mappings.properties`: Maps the portal taglibs to FreeMarker macros. Taglibs let you quickly create common UI components. This properties file is also provided separately for each app taglib. For convenience, these FreeMarker macros appear in the FreeMarker Taglib

Mappings reference guide. See the Taglib tutorials for more information on using each taglib in your theme templates.

CSS Frameworks and Extensions

As noted above, Liferay DXP supports the Sass CSS extension, so you can take full advantage of Sass mixins, nesting, partials, and variables in your CSS.

Also important to note is Clay CSS, the web implementation of Liferay's Lexicon design language. An extension of Bootstrap, Clay CSS fills the gaps between Bootstrap and the needs of Liferay DXP, providing additional components and CSS patterns that you can use in your themes. Clay base, Liferay's Bootstrap API extension, along with Atlas, a custom Bootstrap theme, creates Liferay DXP's Classic theme. See the importing Clay CSS tutorial for more information.

Theme Customizations and Extensions

The theme templates, along with the CSS, provide much of the overall look and feel for the page, but additional extension points/customizations are available. The following extensions and mechanisms are available for themes:

- **Color Schemes:** specifies configurable color scheme settings for Administrator's to configure via the Look and Feel menu. See the color scheme tutorial for more information.
- **Configurable Theme Settings:** settings that let Administrators configure aspects of a theme that may need changed frequently, such as controlling the visibility of certain elements, changing a daily quote, etc. See the Configurable Theme Settings tutorial for more information.
- **Context Contributor:** Exposes Java variables and functionality for you to use in your FreeMarker templates. This lets you use non-JSP templating languages for themes, ADTs, and any other templates used in Liferay DXP. See the Context Contributors tutorial or more information.
- **Theme Contributor:** a package containing UI resources, not attached to a theme, that you want to include on every page. See the Theme Contributors tutorial for more information.
- **Themelet:** small, extendable, and reusable pieces of code that contain CSS and JavaScript. It can be shared with other developers to provide common components for themes, and it only requires the files you want to extend. See the Themelets tutorial for more information.

Portlet Customizations and Extensions

You can customize portlets with these mechanisms and extensions:

- **Portlet FTL Customizations:** customize the base template markup for all portlets. See the Theming Portlets tutorial for more information.
- **Application Display Templates (ADTs):** provides an alternate display style for a portlet. Note that not all portlets support ADTs. See the Application Display Templates (ADTs) User Guide for more information.
- **Portlet Decorator:** lets you customize the exterior decoration for a portlet. See the Portlet Decorators tutorial for more information.
- **Web Content Template:** defines how structures are displayed for web content. See the Web Content Templates User Guide articles for more information.

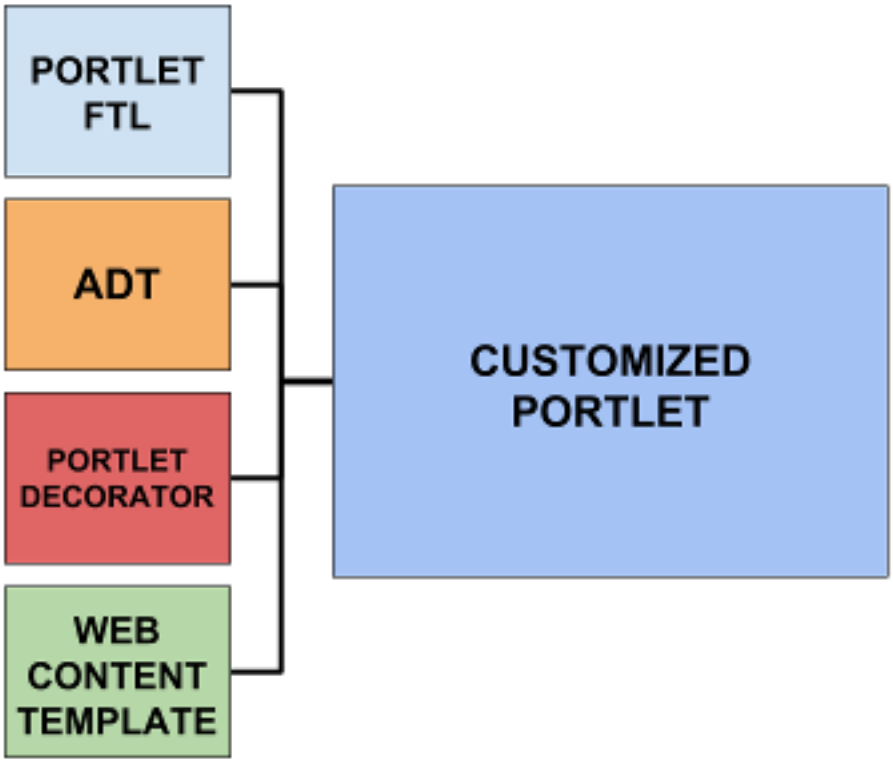


Figure 165.3: There are several extension points for customizing portlets

Theme Workflow

Themes are built on top of one of the following base themes:

- **Unstyled:** provides basic markup, functions, and images for Portal
- **Styled:** inherits from the Unstyled base theme and adds some styling on top

You can use the development tools you're most comfortable with so you can focus on creating a well designed theme. The following Liferay tools help you build themes:

- Theme Builder Gradle Plugin
- Liferay Theme Generator
- Dev Studio
- Blade CLI's Theme Template.

Depending on the tool you choose (Theme Generator, Gradle, Blade CLI, Maven, or Dev Studio), the theme anatomy is a bit different. The overall development process is the same though:

1. Mirror the structure of the files you want to modify. The main modifications are placed in the following files:

    - `portal_normal.ftl`: main theme markup
    - `_custom.scss`: custom CSS styling
    - `main.js`: the theme's JavaScript

2. Build and deploy the theme to your Liferay DXP server.

3. Apply the theme through the Look and Feel menu by selecting your theme's thumbnail.

The finished theme is bundled as a WAR (Web application ARchive) file.

Note: While developing your theme, we recommend that you enable Developer Mode. This unminifies JS files and disable caching for CSS and FreeMarker template files, making the debugging process much easier.

During theme development, if you've built your theme with the Liferay Theme Generator, you can use some helpful Gulp tasks to make the process easier:

- **build:** builds your theme's files based on the specified base theme. See the gulp build tutorial for more information.
- **extend:** sets the base theme or themelet to extend. See the gulp extend tutorial for more information.
- **init:** specifies the app server to deploy your theme to (automatically run during the initial creation of the theme). See the gulp init tutorial for more information.
- **kickstart:** copies files from an existing theme into your theme to help kickstart it. See the gulp kickstart tutorial for more information.
- **status:** lists the base theme/themelets that your theme extends. See the gulp status tutorial for more information.
- **watch:** watches for changes to your theme's files and automatically deploys them to the server when a change is made. See the gulp watch tutorial for more information.

165.3 Understanding the Page Layout

Knowing the layout's structure is crucial to targeting the correct markup for styling, organizing your content, and creating your site. Your page layout is unique to the requirements and design for your site. The Unstyled theme's default page layout is organized into three key sections:

- **Header:** contains the navigation, site logo and title (if shown), and sign-in link when the user isn't logged in
- **Main Content:** contains the portlets or fragments for the page
- **Footer:** contains additional information, such as the copyright or author

Portlets or Fragments

The `#content Section` makes up the majority of the page. Portlets or fragments are contained inside the `#main-content div`. Liferay DXP ships with a default set of applications that provide common functionality, such as forums and Wikis, documents and media, blogs, and more. For more information on using Liferay DXP and its native portlets, see the User & Admin documentation. You can also create custom portlets for your site. Portlets can be added via the Add Menu (referred to as widget), included in a sitemap through the Resources Importer, or they can be embedded in the page's theme. See the portlet tutorials section for more information on creating and developing portlets.

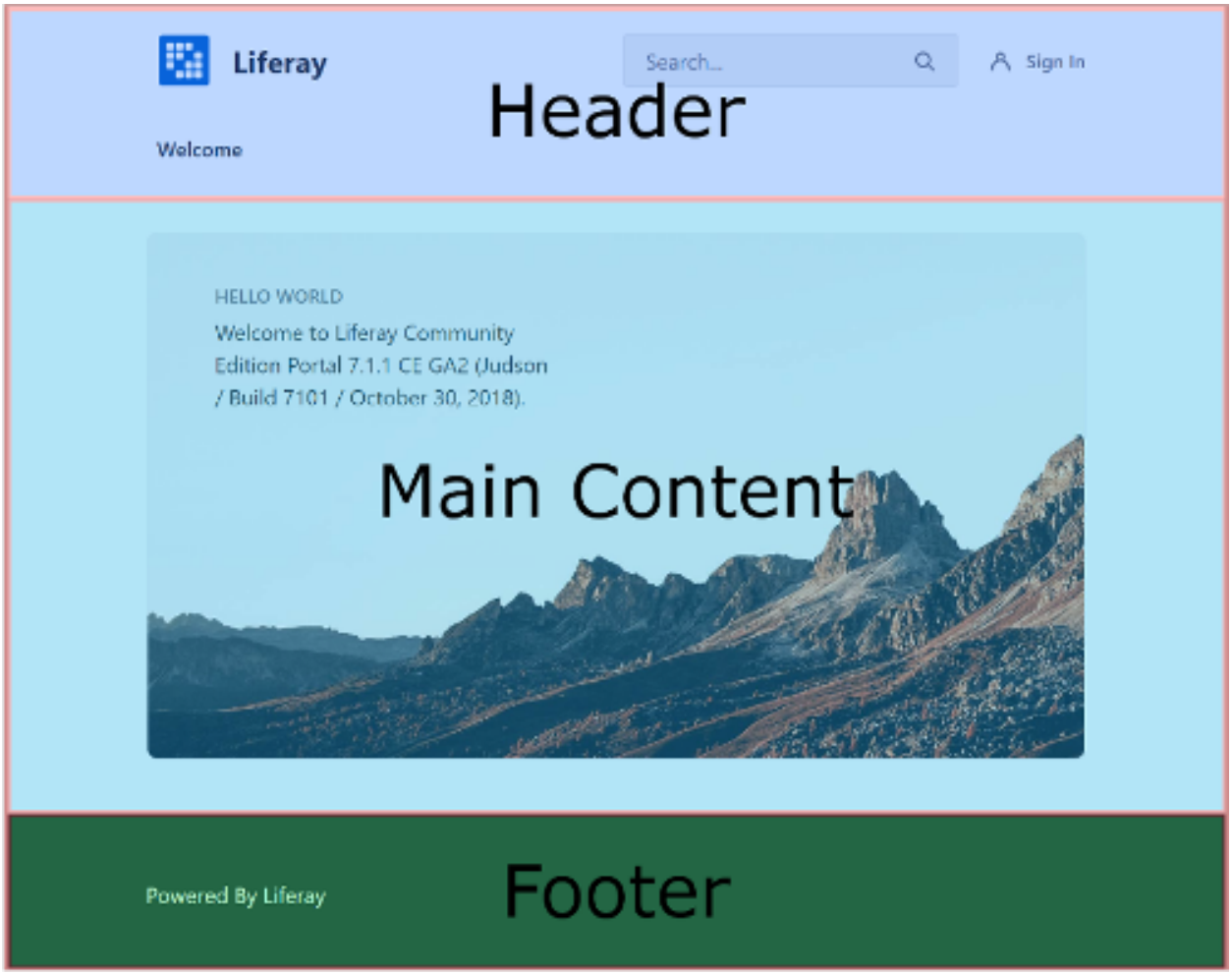


Figure 165.4: The page layout is broken into three key sections.

You can target the elements and IDs shown in the table below to style the page:

Element	ID	Description
div	#wrapper	The container div for the page contents
header	#banner	The page's header
section	#content > #main-content	The main contents of the page (portlets or fragments)
footer	#footer	The page's footer

As shown in the diagram above, you can also add fragments to a page. You can have a page that contains portlets or a content page that contains fragments, not both. Fragments are components—composed of CSS, JavaScript, and HTML—that provide key pieces of functionality for the page (i.e. a carousel or banner). Liferay DXP provides an editor for creating collections of fragments that you can then add to the page. These fragments can be edited on the page to suit your vision.

Layout Templates, Page Templates, and Site Templates

The page layout within the `#content` Section is determined by the Layout Template. Several layout templates are included out-of-the-box. You can also create custom layout templates manually or

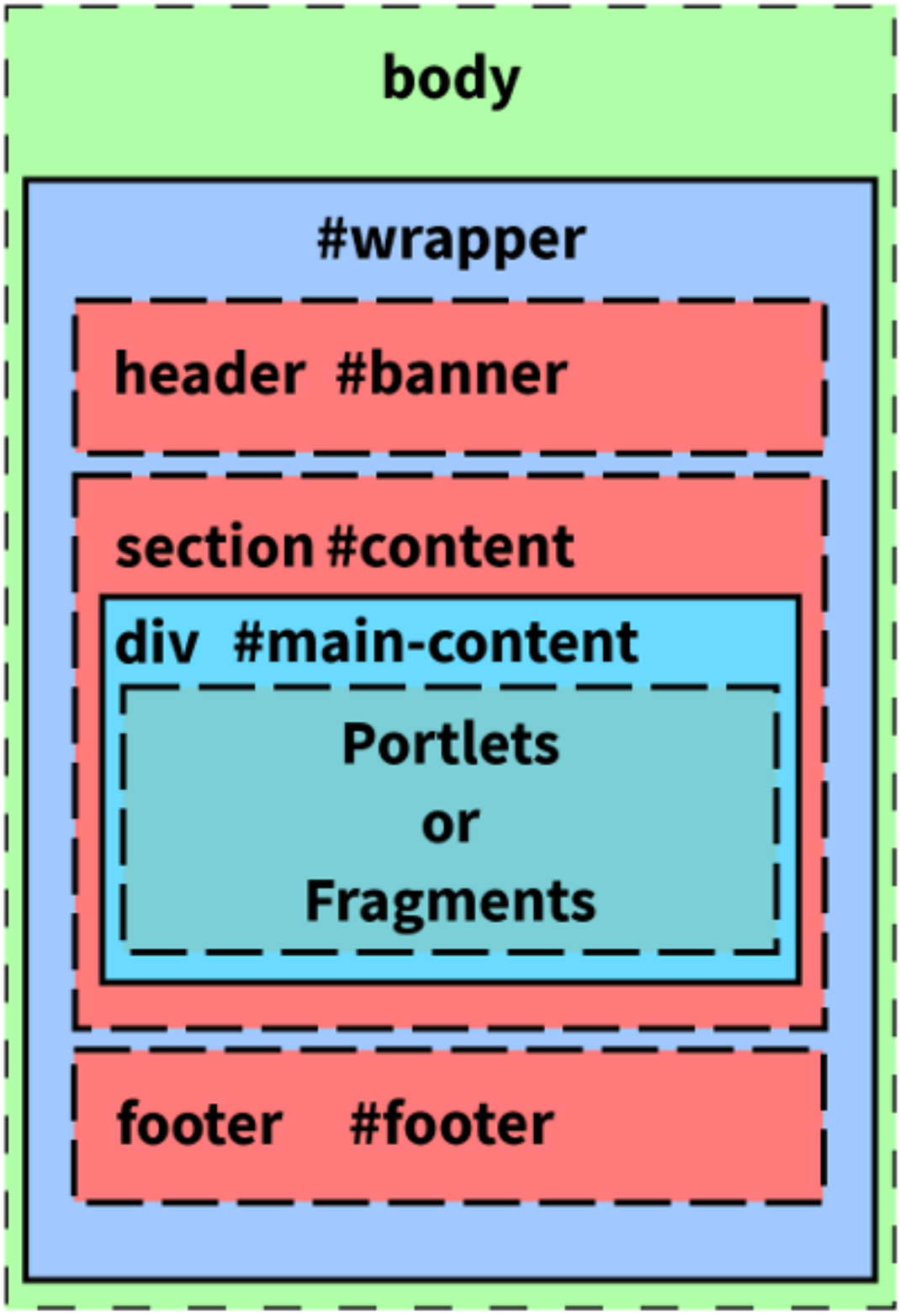


Figure 165.5: Each section of the page has elements and IDs that you can target for styling.

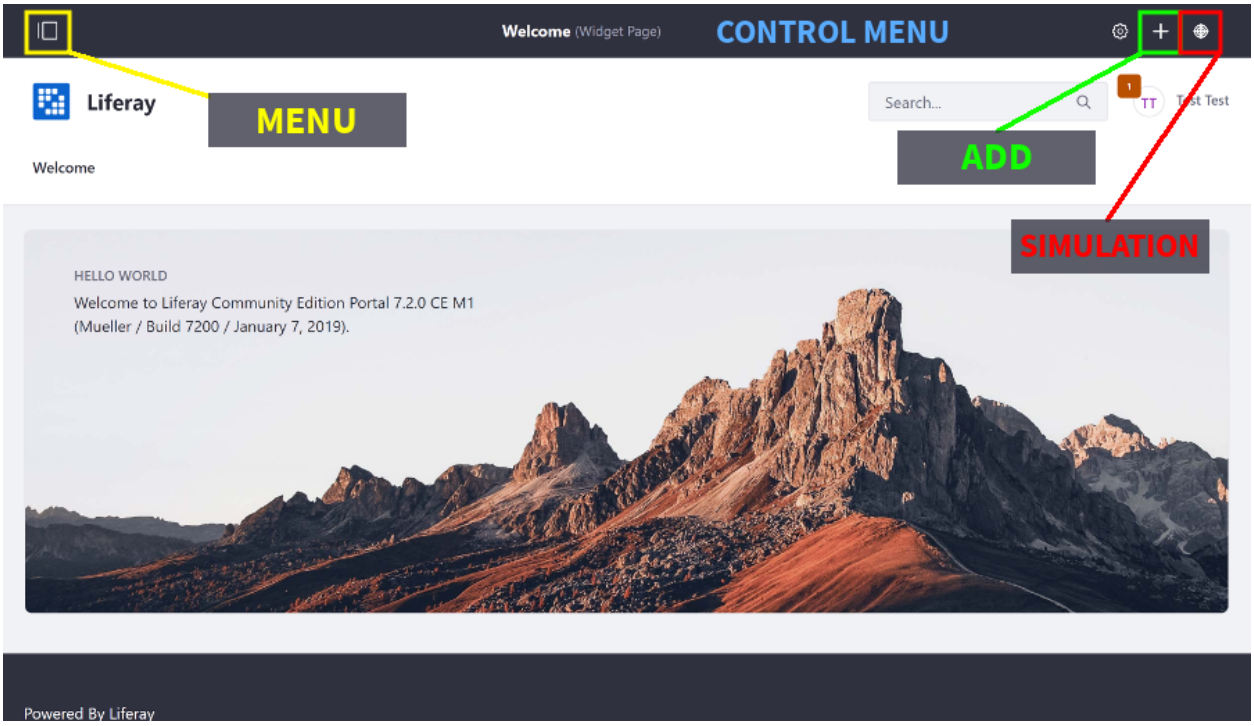


Figure 165.6: Remember to account for the product navigation sidebars and panels when styling your site.

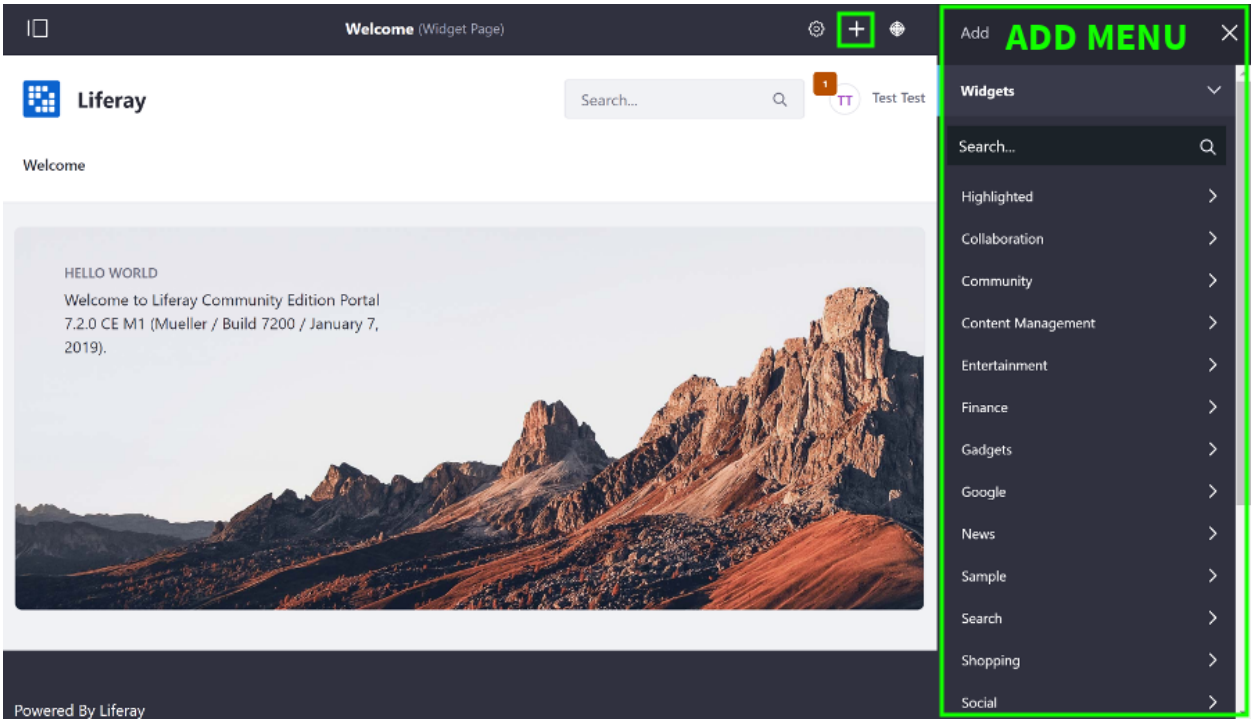


Figure 165.7: The Add Menu pushes the main contents to the left.

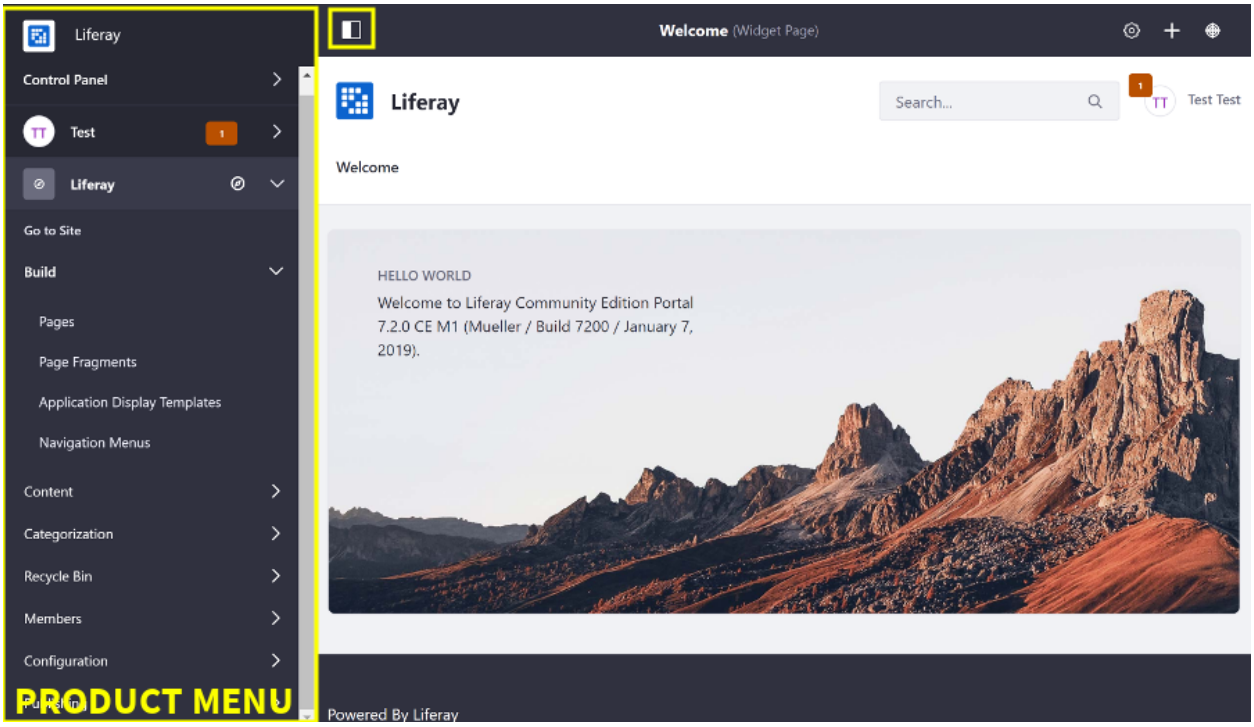


Figure 165.8: The Product Menu pushes the main contents to the right.

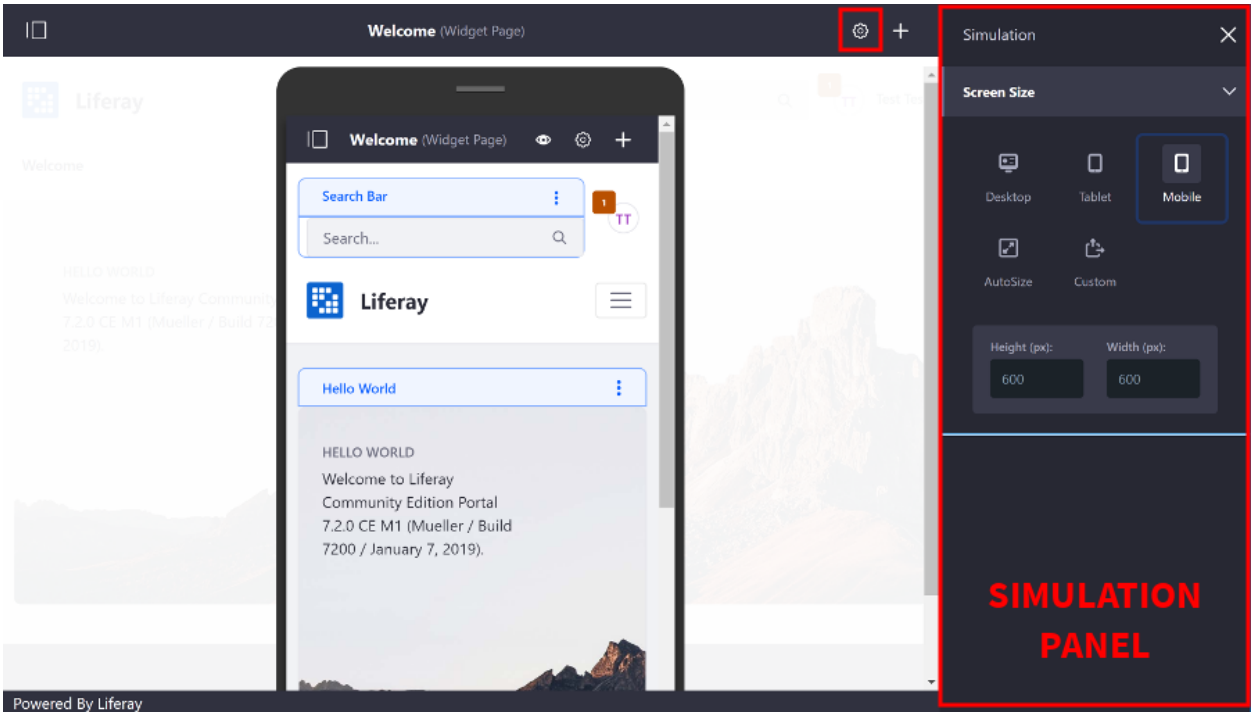


Figure 165.9: The Simulation Panel pushes the main contents to the left.

CHAPTER 166

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, most of the Liferay Gradle plugins covered in this section are already applied by default. The com.liferay.gradle.plugins.workspace and com.liferay.gradle.plugins dependencies provide them, both of which are preset in workspace by default.

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.

166.1 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 Resolving Third Party Library Package Dependencies 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. Open the Gogo shell.

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. Open the Gogo shell.

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.

166.2 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`

166.3 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
```

166.4 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.

166.5 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"
}
```

166.6 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.supp`

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"
}
```

166.7 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
```

166.8 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"
}
```

166.9 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}.${platf ${bitMode}.${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 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 type `File` 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.

go$command${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}"]`

166.10 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.

166.11 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"
}
```

166.12 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

166.13 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}/META-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 | "${downloadLiferayModuleConfigGenerator.moduleDir}/bin/index.js"

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.

166.14 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.

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.

166.15 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`.

166.16 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: "3.0.31"
    }

    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: "1.0.29"
}
```

166.17 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
```

166.18 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.

166.19 REST Builder Gradle Plugin

The REST Builder Gradle plugin lets you generate a REST layer defined in the REST Builder rest-config.yaml and rest-openapi.yaml 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.rest.builder", version: "1.0.21"
    }
```

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

apply plugin: "com.liferay.portal.tools.rest.builder"
```

The REST Builder plugin automatically applies the `java` plugin.

Since the plugin automatically resolves the Liferay REST 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 `buildREST` | - | `BuildRESTTask` | Runs the Liferay REST Builder.

BuildRESTTask

Tasks of type `BuildRESTTask` 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` | REST Builder command line arguments `classpath` | `project.configurations.restBuilder main` | `"com.liferay.portal.tools.rest.builder.RESTBuilder"` `systemProperties` | `[]`

Task Properties Property Name | Type | Default Value | Description `copyrightFile` | `File` | `null` | The file that contains the copyright header. `restConfigDir` | `File` |`${project.projectDir}` | The directory that contains the `rest-config.yaml` and `rest-openapi.yaml` files.

In the typical scenario, the `rest-config.yaml` and `rest-openapi.yaml` files are contained in the project directory of `my-rest-app-impl`. In the `build.gradle` of the same module, apply the `com.liferay.rest.builder` plugin.

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 added to use REST Builder.

Liferay REST Builder Dependency

By default, the plugin creates a configuration called `restBuilder` and adds a dependency to the latest released version of Liferay REST Builder.

```
dependencies {
    restBuilder group: "com.liferay", name: "com.liferay.portal.tools.rest.builder", version: "1.0.22"
}
```

166.20 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.

166.21 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
```

166.22 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.generator 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.

166.23 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` model 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"
}
```

166.24 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.paren` 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.

166.25 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"
}
```

166.26 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"
}
```

166.27 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.

166.28 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"
}
```

166.29 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.

166.30 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"
}
```

166.31 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"
}
```

CHAPTER 167

MAVEN

Liferay provides plugins that you can apply to your Maven project. This reference documentation describes

- Configuring the plugin in your `pom.xml` file.
- The plugin's available goals you can leverage.
- The plugin's configuration properties.

If you're looking for additional instructions on using Maven with your modules, see the Maven tutorials.

167.1 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/7.1.0-b3/liferay-ce-portal-tomcat-7.1-b3-20180611140920623.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.

167.2 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.

167.3 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.

167.4 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.

167.5 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.

167.6 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.

167.7 REST Builder Plugin

The REST Builder plugin lets you generate a REST layer defined in the REST Builder rest-config.yaml and rest-openapi.yaml 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.portal.tools.rest.builder</artifactId>
            <version>1.0.22</version>
            <configuration>
            </configuration>
        </plugin>
    ...
    </plugins>
</build>
```

You can view an example POM containing the REST Builder configuration here.

Goals

The plugin adds one Maven goal to your project:

Name | Description rest-builder:build | Runs the Liferay REST Builder.

Available Parameters

You can set the following parameters in the <configuration> section of the POM:

Parameter Name | Type | Default Value | Description copyrightFile | File | null | The file that contains the copyright header. restConfigDir | File | ${project.projectDir} | The directory that contains the rest-config.yaml and rest-openapi.yaml files.

167.8 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.

167.9 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.

167.10 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.

167.11 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.

167.12 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.

167.13 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.

167.14 Content Targeting Report Template

In this article, you'll learn how to create a Liferay Content Targeting Report application as a Liferay module. To create a Content Targeting Report via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t content-targeting-report -p [package name] -c [class name] [project name]
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.content.targeting.report \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.1
```

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 `content-targeting-report`. To create a report project called `hits-by-country` with a package prefix of `com.liferay` and a class name of `HitsByCountryReport`, use this command:

```
blade create -t content-targeting-report -p com.liferay -c HitsByCountry hits-by-country
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.content.targeting.report \
    -DgroupId=com.liferay \
    -DartifactId=hits-by-country \
    -Dpackage=com.liferay \
    -Dversion=1.0 \
    -DclassName=HitsByCountry \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.1
```

The command above creates a Content Targeting Rule project named `hits-by-country` in the current folder. In the class, you're creating a service of type `com.liferay.content.targeting.api.model.Report` and extending the `com.liferay.content.targeting.api.model.BaseJSPReport` class. 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 folder structure looks like this:

- hits-by-country
    - gradle
        - wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src

```
        * main

            · java
            · com/liferay/content/targeting/report
            · HitsByCountryReport.java

            · resources
            · content
            · Language.properties

            · META-INF
            · resources
            · edit.jsp
            · view.jsp

    – bnd.bnd
    – build.gradle
    – gradlew
```

The Maven-generated project includes a `pom.xml` file and does not include the Gradle-specific files, but otherwise, appears exactly the same.

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.

167.15 Content Targeting Rule Template

In this article, you'll learn how to create a Liferay Content Targeting Rule application as a Liferay module. To create a Content Targeting Rule via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t content-targeting-rule -p [package name] -c [class name] [project name]
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.content.targeting.rule \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.1
```

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 `content-targeting-rule`. To create a rule project called `weather` with a package prefix of `com.liferay` and a class name of `Weather`, use this command:

```
blade create -t content-targeting-rule -p com.liferay -c Weather weather
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.content.targeting.rule \
    -DgroupId=com.liferay \
    -DartifactId=weather \
    -Dpackage=com.liferay \
    -Dversion=1.0 \
    -DclassName=Weather \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.1
```

The command above creates a Content Targeting Rule project named `weather` in the current folder. In the class, you're creating a service of type `com.liferay.content.targeting.api.model.Rule` and extending the `com.liferay.content.targeting.api.model.BaseJSPRule` class. 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 folder structure looks like this:

- weather
    - gradle
        - wrapper
            - gradle-wrapper.jar
            - gradle-wrapper.properties
    - src
        - main
            - java
            - com/liferay/content/targeting/rule
            - WeatherRule.java
            - resources
            - content
            - Language.properties
            - META-INF
            - resources
            - view.jsp
    - bnd.bnd
    - build.gradle
    - gradlew

The Maven-generated project includes a `pom.xml` file and does not include the Gradle-specific files, but otherwise, appears exactly the same.

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.

167.16 Content Targeting Tracking Action Template

In this article, you'll learn how to create a Liferay Content Targeting Tracking Action application as a Liferay module. To create a Content Targeting Tracking Action via the command line using Blade CLI or Maven, use one of the commands with the following parameters:

```
blade create -t content-targeting-tracking-action -p [package name] -c [class name] [project name]
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.content.targeting.tracking.action \
    -DartifactId=[projectName] \
    -Dpackage=[packageName] \
    -DclassName=[className] \
    -DliferayVersion=7.1
```

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 `content-targeting-tracking-action`. To create a tracking action project called `newsletter` with a package prefix of `com.liferay` and a class name of `Newsletter`, use this command:

```
blade create -t content-targeting-tracking-action -p com.liferay -c Newsletter newsletter
```

or

```
mvn archetype:generate \
    -DarchetypeGroupId=com.liferay \
    -DarchetypeArtifactId=com.liferay.project.templates.content.targeting.tracking.action \
    -DgroupId=com.liferay \
    -DartifactId=newsletter \
    -Dpackage=com.liferay \
    -Dversion=1.0 \
    -DclassName=Newsletter \
    -Dauthor=Joe Bloggs \
    -DliferayVersion=7.1
```

The command above creates a Content Targeting Tracking Action project named `newsletter` in the current folder. In the class, you're creating a service of type `com.liferay.content.targeting.api.model.TrackingAc` and extending the `com.liferay.content.targeting.api.model.BaseJSPTrackingAction` class. 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 folder structure looks like this:

- newsletter
    - gradle
        - wrapper
            - gradle-wrapper.jar

```
            · gradle-wrapper.properties

    – src

        * main

            · java
            · com/liferay/content/targeting/tracking/action
            · NewsletterTrackingAction.java

            · resources
            · content
            · Language.properties

            · META-INF
            · resources
            · view.jsp

    – bnd.bnd
    – build.gradle
    – gradlew
```

The Maven-generated project includes a `pom.xml` file and does not include the Gradle-specific files, but otherwise, appears exactly the same.

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.

CHAPTER 168

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 169

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

Visit a particular sample page to learn more!

CHAPTER 170

NPM SAMPLES

This section focuses on Liferay npm sample portlets built with Gradle. You can view these samples by visiting the `apps/npm` folder corresponding to your preferred build tool:

- Gradle sample apps
- Liferay Workspace sample apps

Note: When building the npm samples, an error can occur caused by the limit of open files allowed by your operating system. Consult your operating system vendor's documentation to learn how to configure the maximum number of open files for your OS.

The following npm samples are documented:

- Angular npm Portlet
- Angular npm Deduplication Portlet
- Billboard.js npm Portlet
- jQuery npm Portlet
- Metal.js npm Portlet
- React npm Portlet
- Simple npm Portlet
- Vue.js npm Portlet

Visit a particular sample page to learn more!

170.1 Angular 6 npm Portlet

The Angular 6 npm Portlet sample provides a portlet that uses the Angular framework (version 6) to render its output.

This portlet showcases Angular's newest version and how to leverage it in Liferay DXP. See this article for more information on what's new with Angular 6.

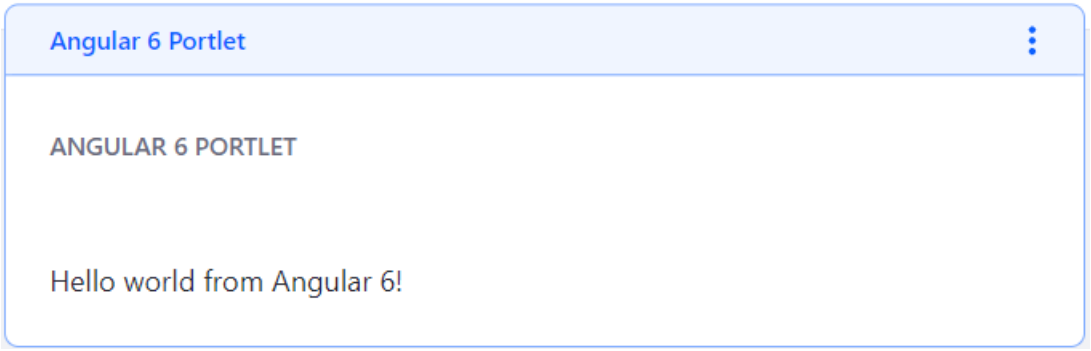


Figure 170.1: Type custom text in the field and watch it instantaneously displayed in the portlet.

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",
    "tsc": "tsc"
},
```

Where Is This Sample?

This sample is built with the following build tool:

- Gradle

170.2 Angular npm Deduplication Sample

The Angular npm Deduplication sample provides a portlet that uses the Angular framework to render its output.

This is done by providing a deduplicated instance of the Angular framework as an OSGi bundle and then leveraging it from a sample portlet.

What API(s) and/or code components does this sample highlight?

This sample is broken into two modules:

- `angular-consumer-portlet`
- `angular-provider`

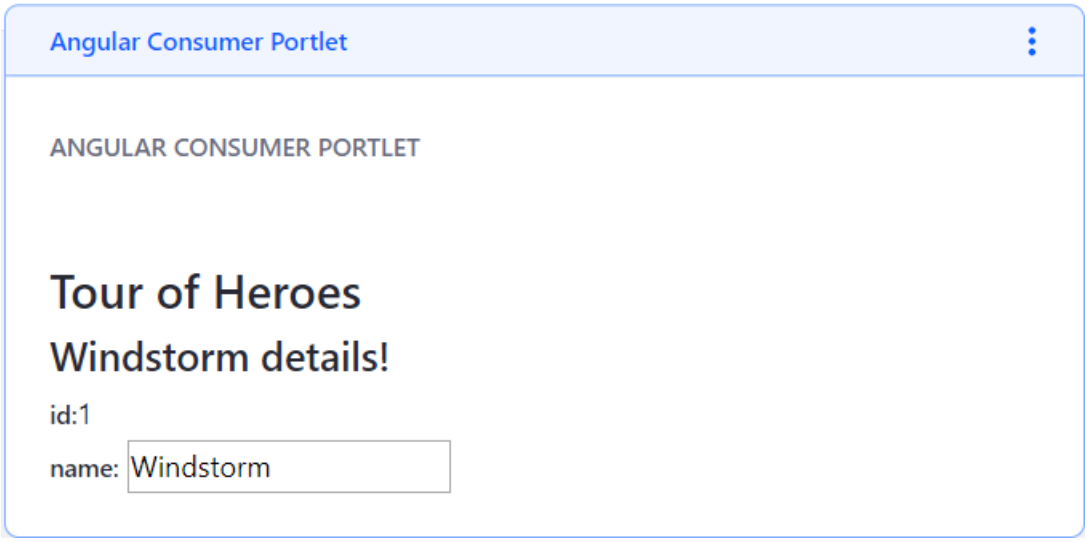


Figure 170.2: Type custom text in the field and watch it instantaneously displayed in the portlet..

The Angular Provider sample generates an OSGi bundle that provides a deduplicated instance of the Angular framework that portlets can share when rendering their output. The Angular Consumer portlet uses the deduplicated instance of the Angular framework.

Note: Both modules must be deployed to the server for this sample to run.

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

170.3 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.

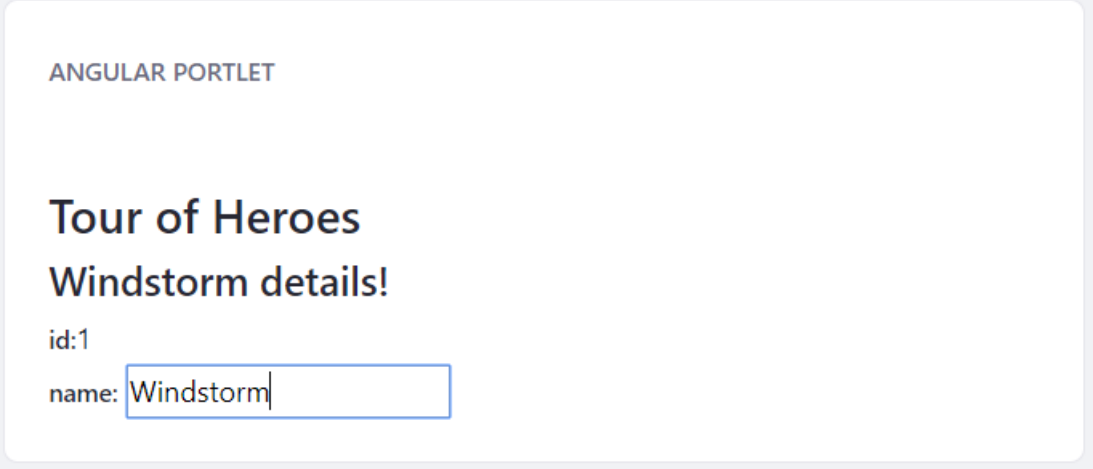


Figure 170.3: Type custom text in the field and watch it instantaneously displayed in the portlet.

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?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

170.4 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.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

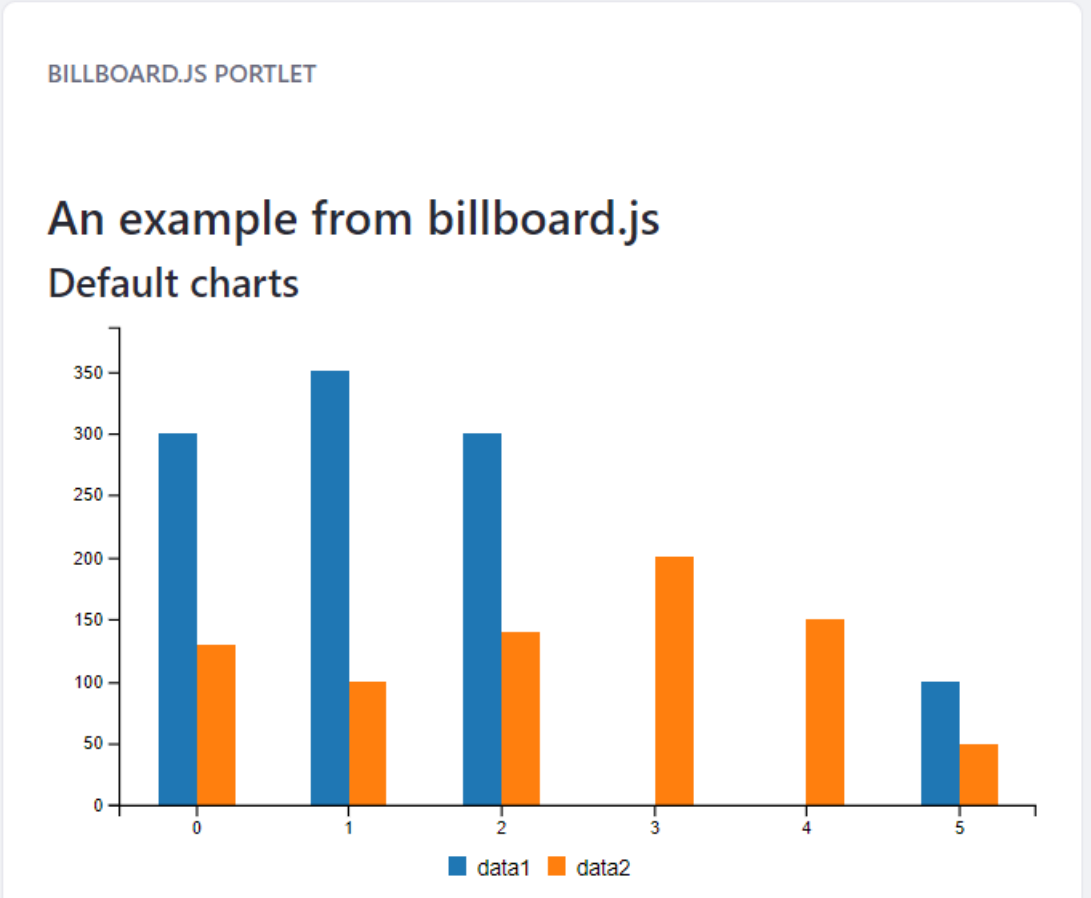


Figure 170.4: The Billboard.js npm Portlet shows off some nice looking graphs using Billboard.js.

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?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

170.5 jQuery npm Portlet

The jQuery npm Portlet sample provides a portlet that uses the jQuery framework to render its output.

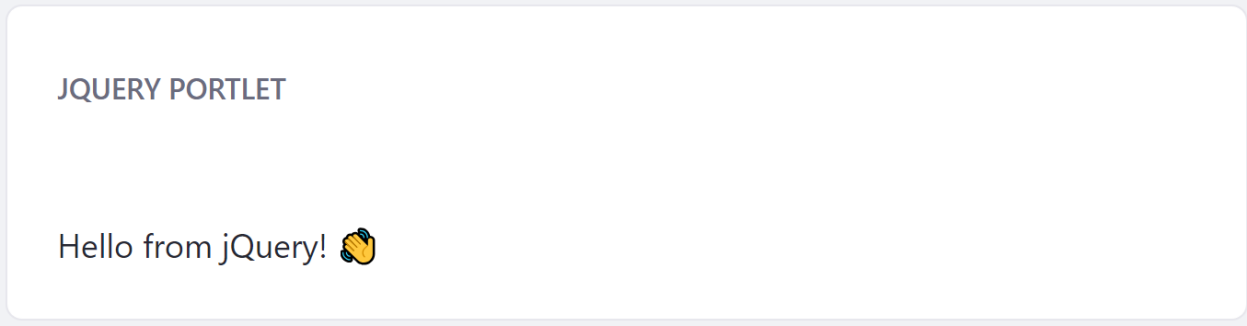


Figure 170.5: Clicking on the portlet's hand symbol displays a message.

This portlet showcases the fast HTML document traversal jQuery offers.

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?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

170.6 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.

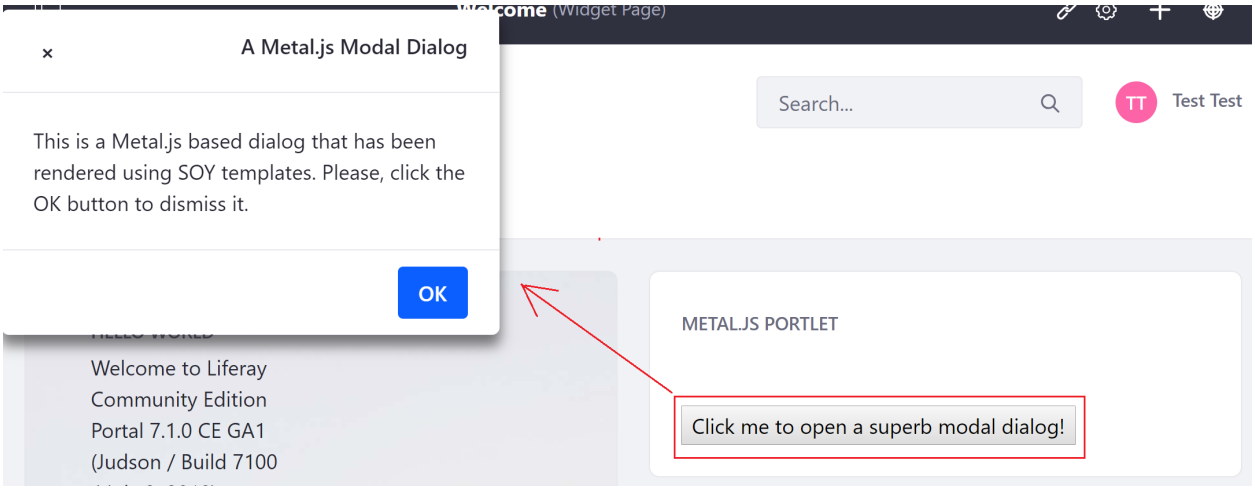


Figure 170.6: Clicking the button returns displays a dialog window.

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": "metalsoy && babel --source-maps -d build/resources/main/META-INF/resources src/main/resources/META-INF/resources && liferay-npm-bundler"
},
```

Where Is This Sample?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

170.7 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.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

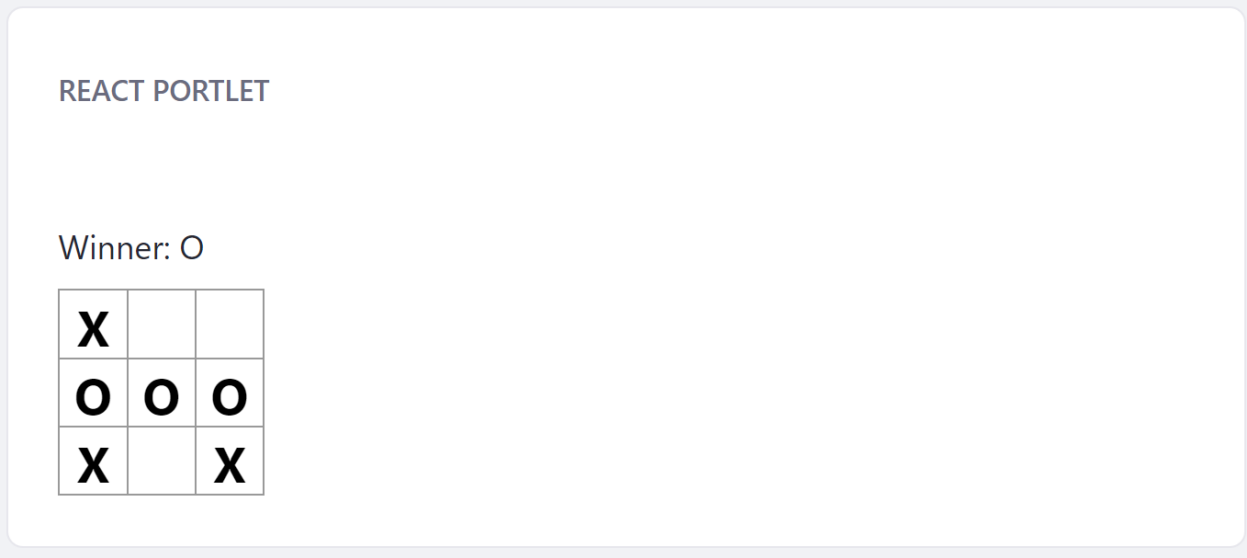


Figure 170.7: You can play the game Tic-tac-toe with this sample portlet.

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?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

170.8 Simple npm Portlet

The Simple npm Portlet sample provides a portlet that uses the isarray npm package when rendering its output.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

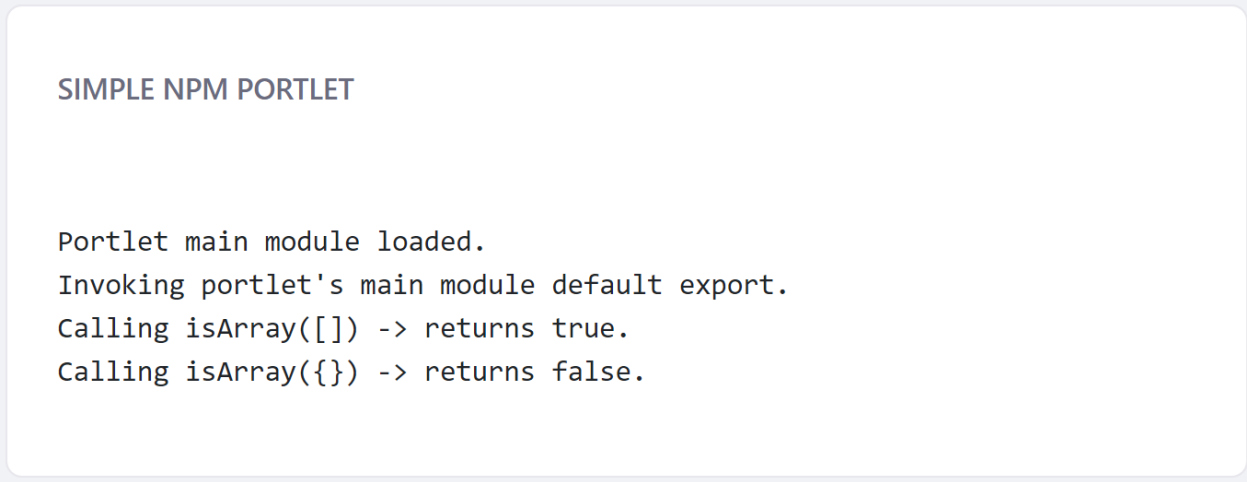


Figure 170.8: The portlet's status and actions are displayed as output.

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?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

170.9 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.

What API(s) and/or code components does this sample highlight?

This sample leverages the npm development workflow support.

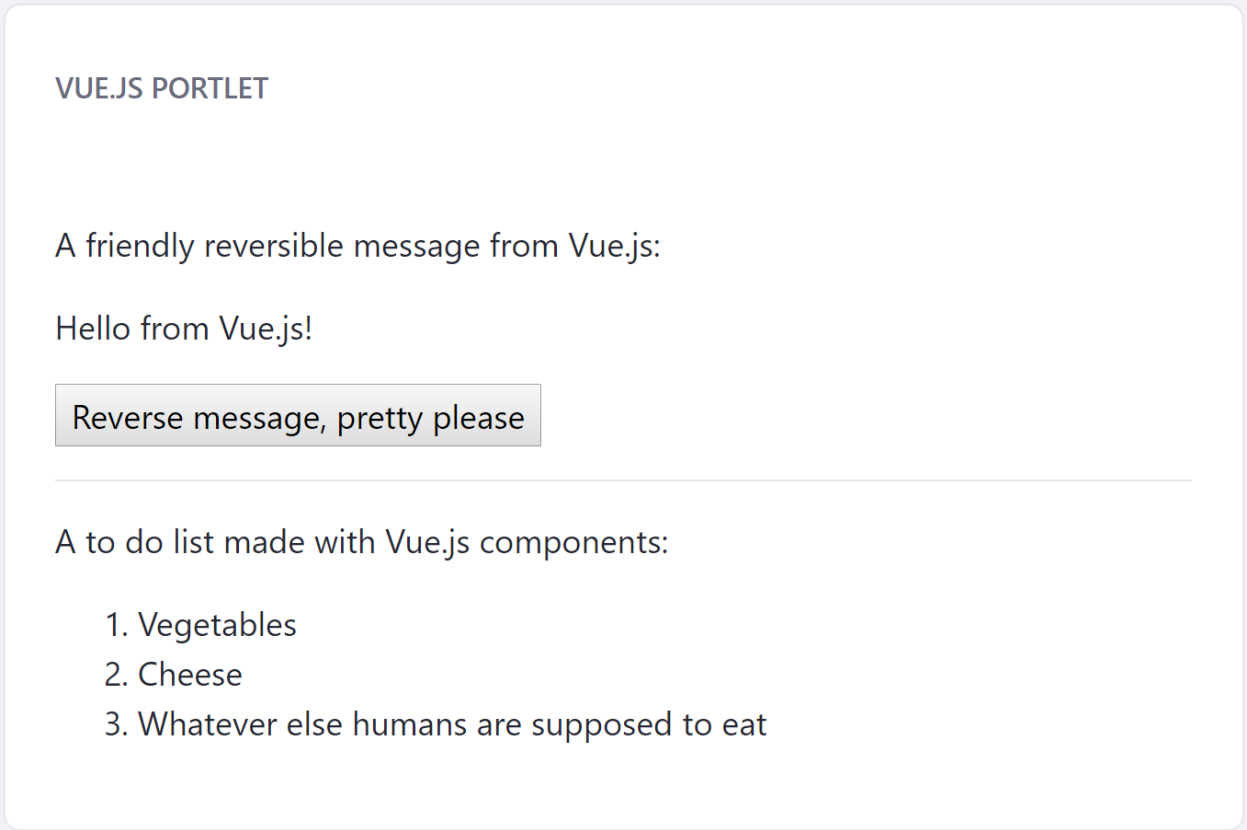


Figure 170.9: Clicking the portlet's button reverses the message.

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?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

CHAPTER 171

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!

171.1 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* ( ) → *Widgets* → *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!

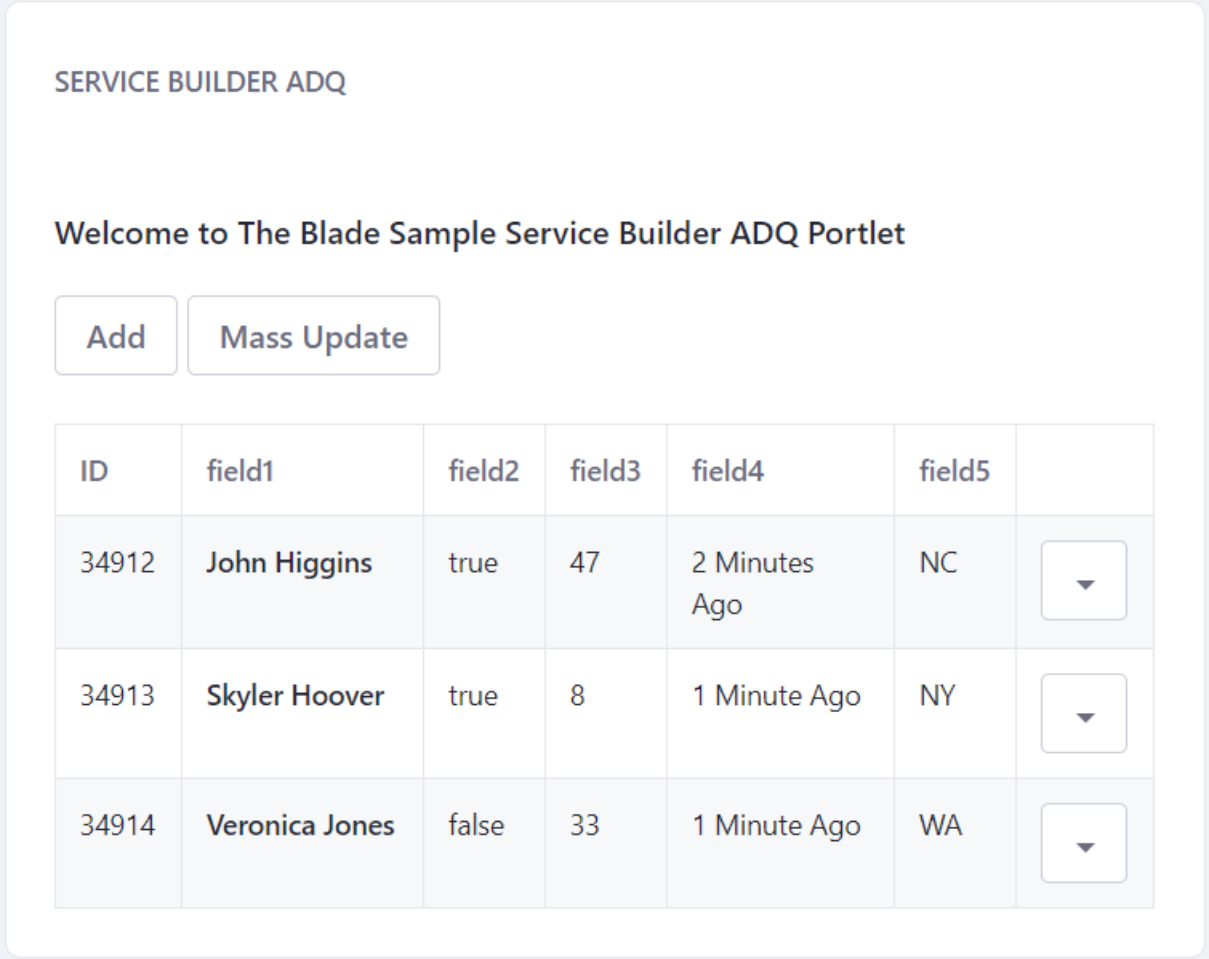


Figure 171.1: This sample provides options to add entities and perform a mass update.

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));
```

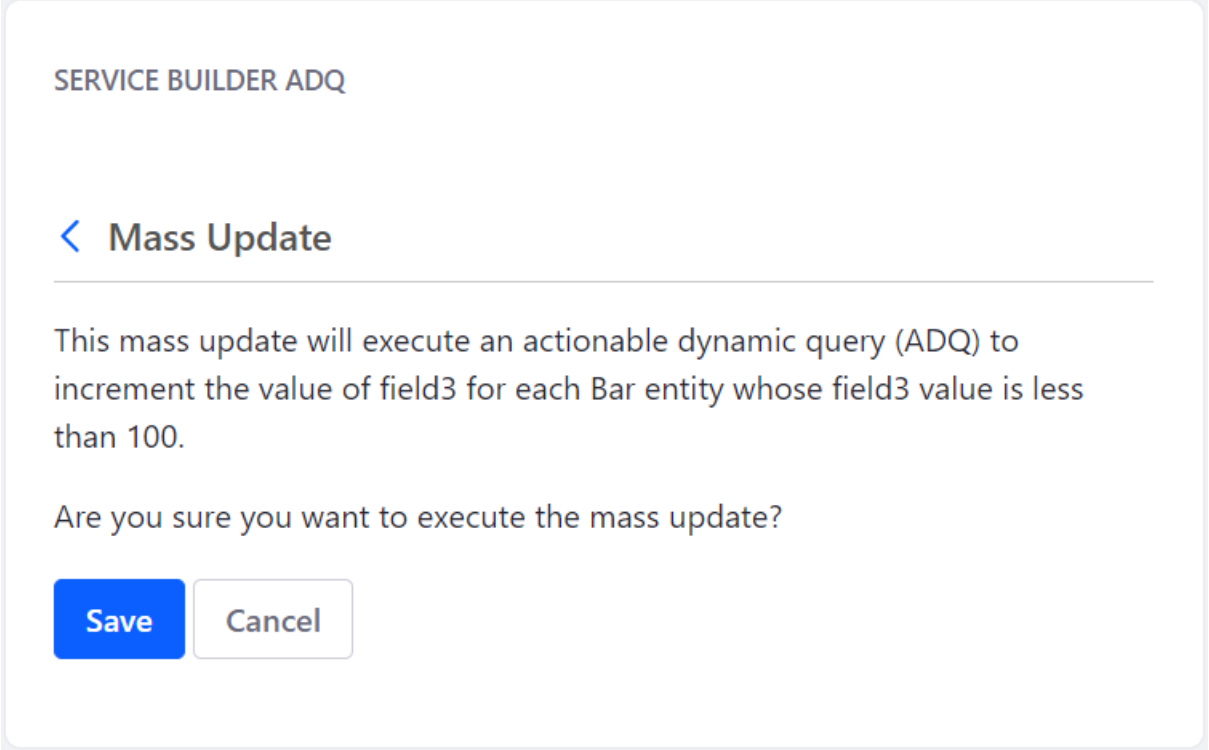


Figure 171.2: Clicking the *Save* button executes the mass update.

```
            }

        });

    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.

171.2 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` 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.

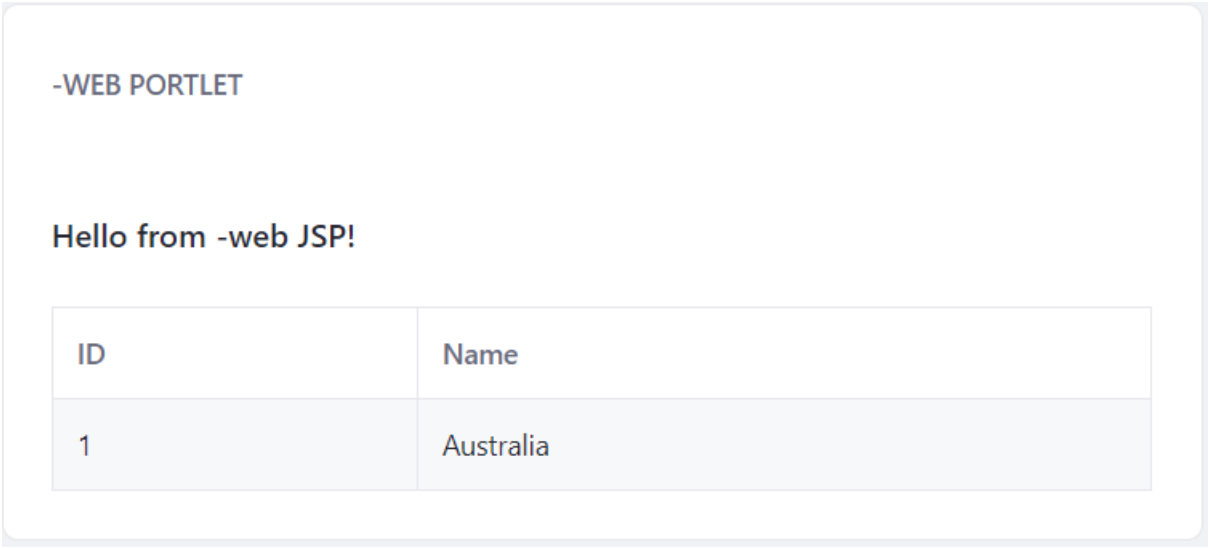


Figure 171.3: This sample prints out the values previously inputted into the database.

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?

The sample configures the data source using Spring Beans and 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.

Configuring the Data Source

The `-service` module's `src/main/resources/META-INF/spring/ext-spring.xml` file configures the external data source connection and applies the alias `extDataSource` to the data source. The `service.xml` file `entity` element specifies the data source via the attribute assignment `data-source="extDataSource"`. The `ext-spring.xml` and `service.xml` files demonstrate the configuration steps explained in Connecting the Data Source Using Spring Beans.

Accessing Data

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.

171.3 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, open your `[LIFERAY_HOME]/tomcat-9.0.6/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. Open your `[LIFERAY_HOME]/tomcat-9.0.6/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` 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. 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.

Figure 171.4: This sample prints out the values previously inputted into the database.

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.

171.4 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.

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.

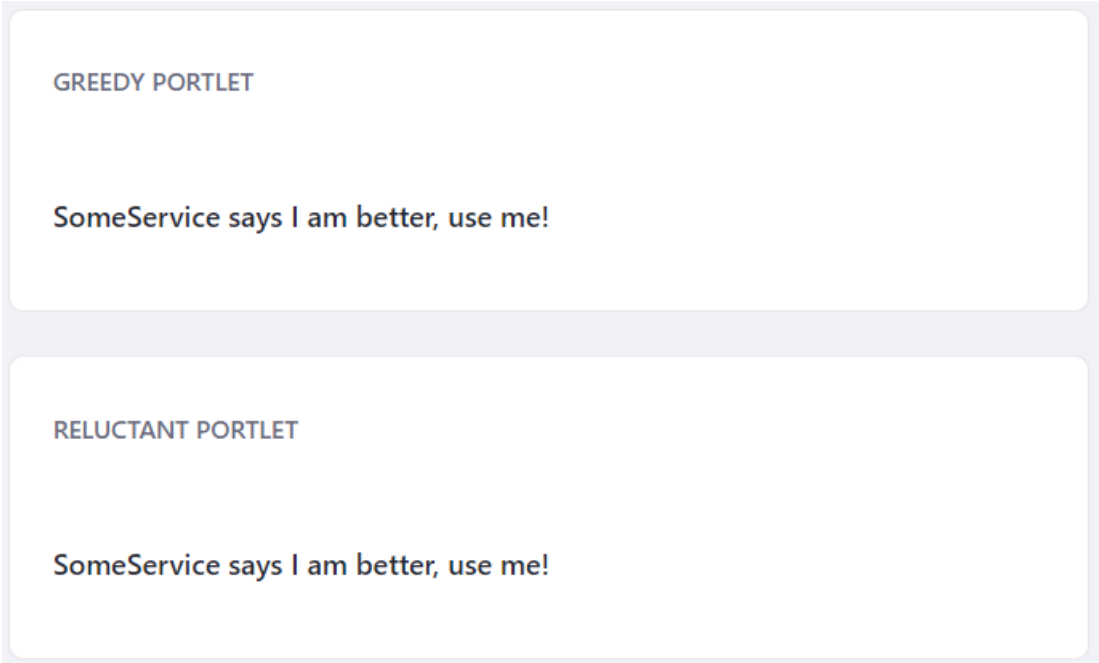


Figure 171.5: The Greedy Policy Option app provides two portlets that only print text. You'll dive deeper later to discover their interesting capabilities involving services.

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
  - `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.

Figure 171.6: *Reluctant Portlet* displays the message "SomeService says I am default!"

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.

Figure 171.7: *Greedy Portlet* displays the message "SomeService says I am better, use me!"

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.

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.

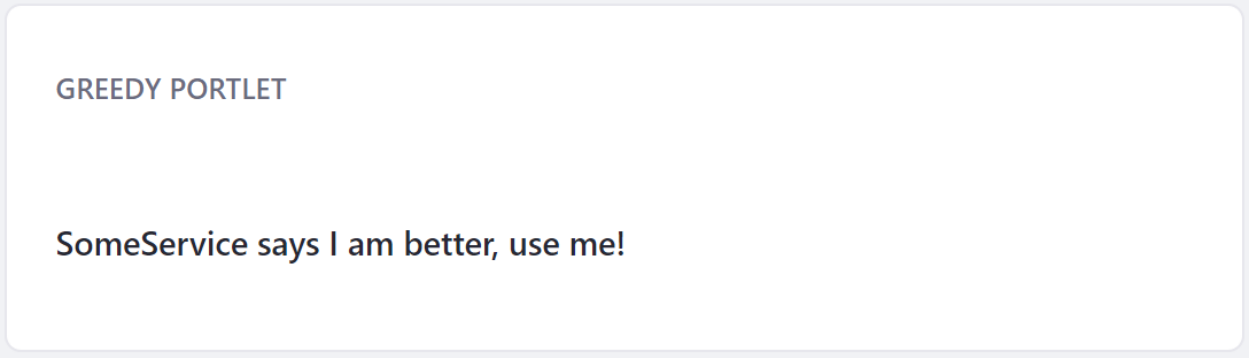


Figure 171.8: The *Greedy Portlet* is using a `HigherRankedService` instance

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` 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.ser`

Note: For deploying to 7.0, use file with suffix `.config`. For earlier versions (i.e., Liferay DXP 7.0 Fix Packs earlier than Fix Pack 8 and Liferay CE Portal 7.0 GA3 or earlier), 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.

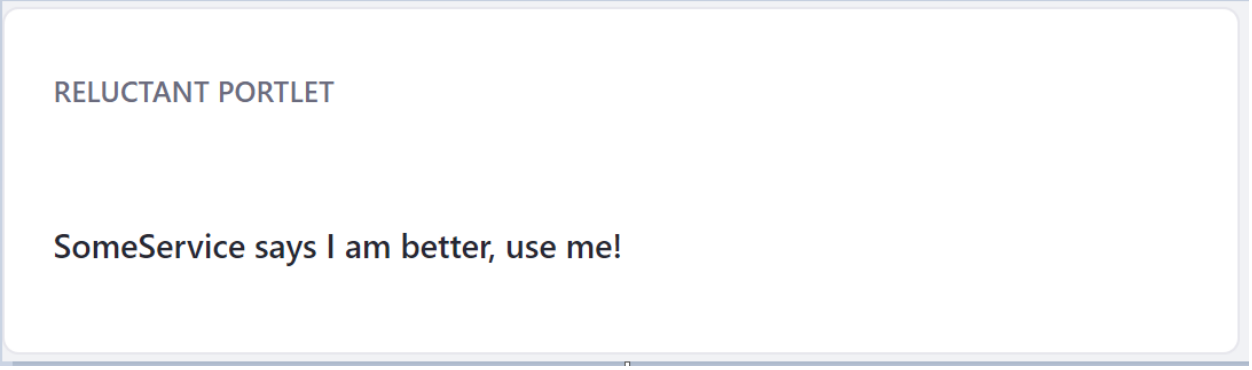


Figure 171.9: *Reluctant Portlet* is using the `HigherRankedService` instance instead of a `DefaultService` instance.

Reluctant Portlet displays a new message "SomeService says I am better, use me!".

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

171.5 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`.

Where Is This Sample?

This sample is built with the following build tools:

- Gradle
- Liferay Workspace

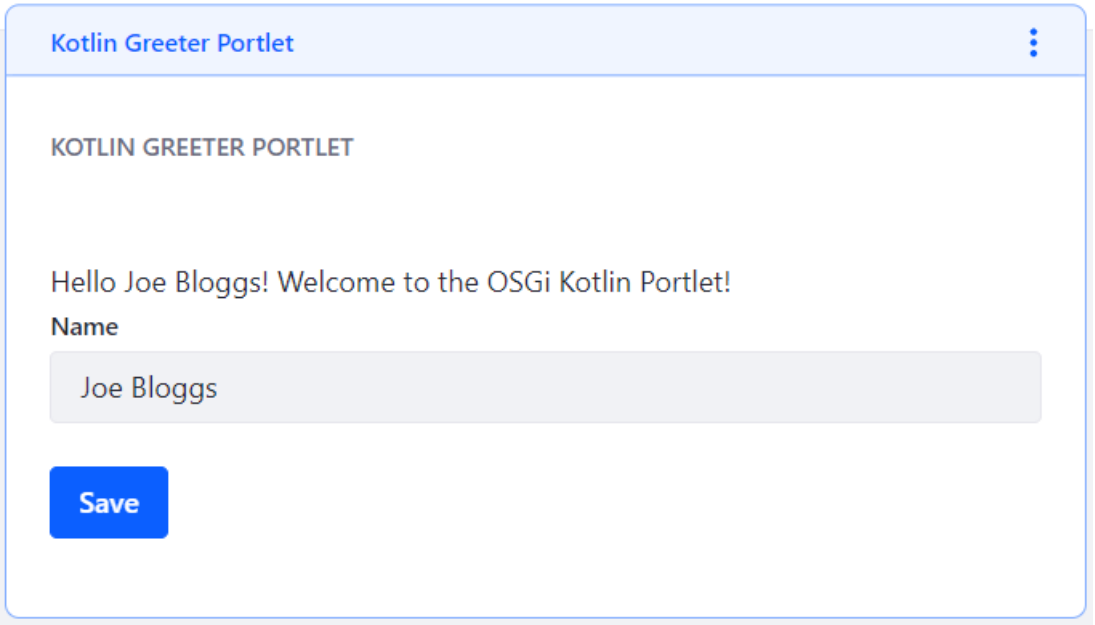


Figure 171.10: After saving the inputted name, it's displayed as a greeting on the portlet page.

171.6 Shared Language Keys

The Shared Language Keys sample provides a JSP portlet that displays language keys.

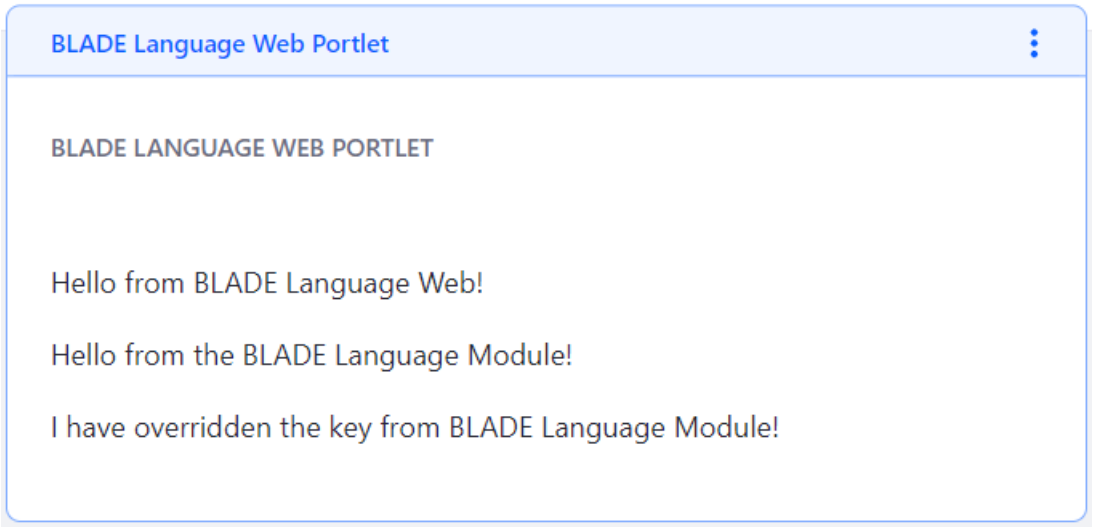


Figure 171.11: The sample JSP portlet displays three language keys.

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!

When you place the sample BLADE Language Web portlet on a Liferay DXP page, you're presented with three language keys:

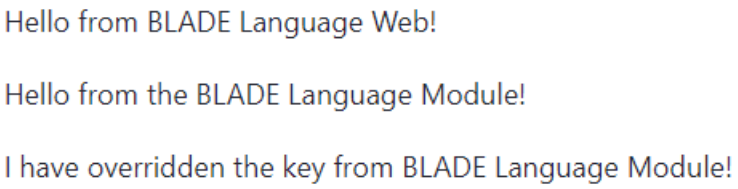


Figure 171.12: The Language Web portlet displays three phrases, two of which are shared from a different module.

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

171.7 Simulation Panel App

The Simulation Panel App provides new functionality in Liferay DXP's Simulation Menu. When deploying this sample with no customizations, the *Simulation Sample* feature is provided in the Simulation Menu with four options.

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

171.8 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.

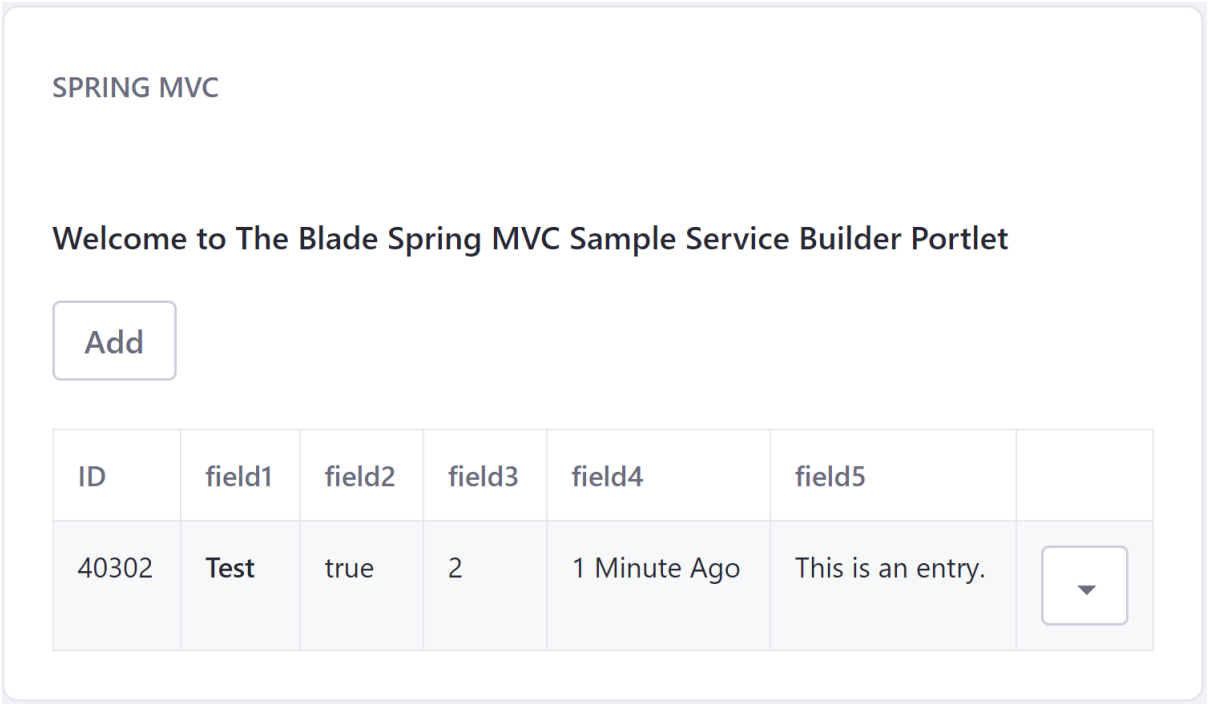


Figure 171.13: Click *Add* and fill out the sample fields to generate a custom entry in the portlet's table.

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 172

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

Visit a particular sample page to learn more!

172.1 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.

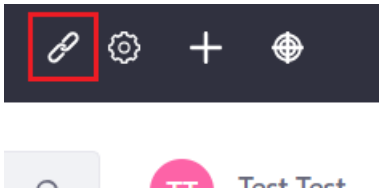


Figure 172.1: The User area of the Control Menu is provided an additional link button when the Control Menu Entry sample is deployed to Liferay DXP.

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

172.2 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:

Likewise, the Documents and Media portlet provides the same option after selecting *Show Actions* from the portlet's Configuration menu.

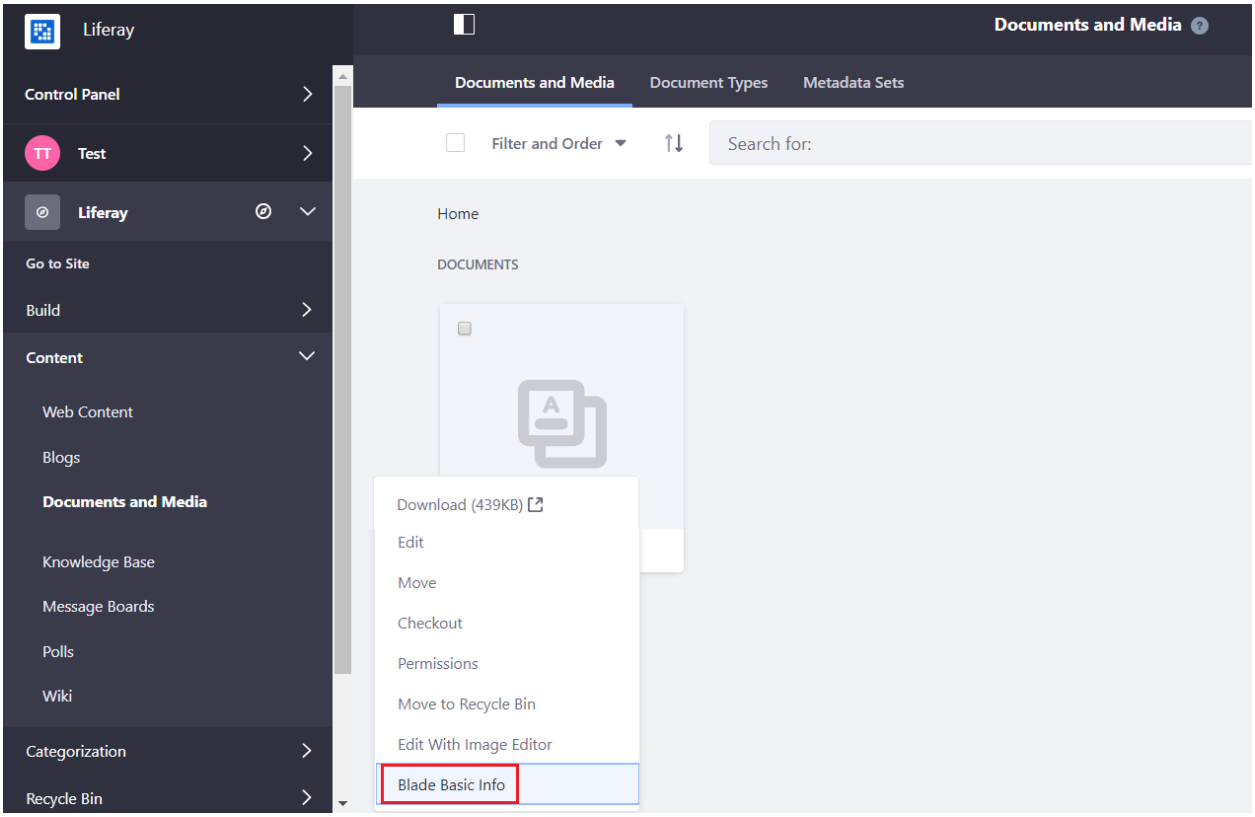


Figure 172.2: The new *Blade Basic Info* option is available from the entry's Options menu.

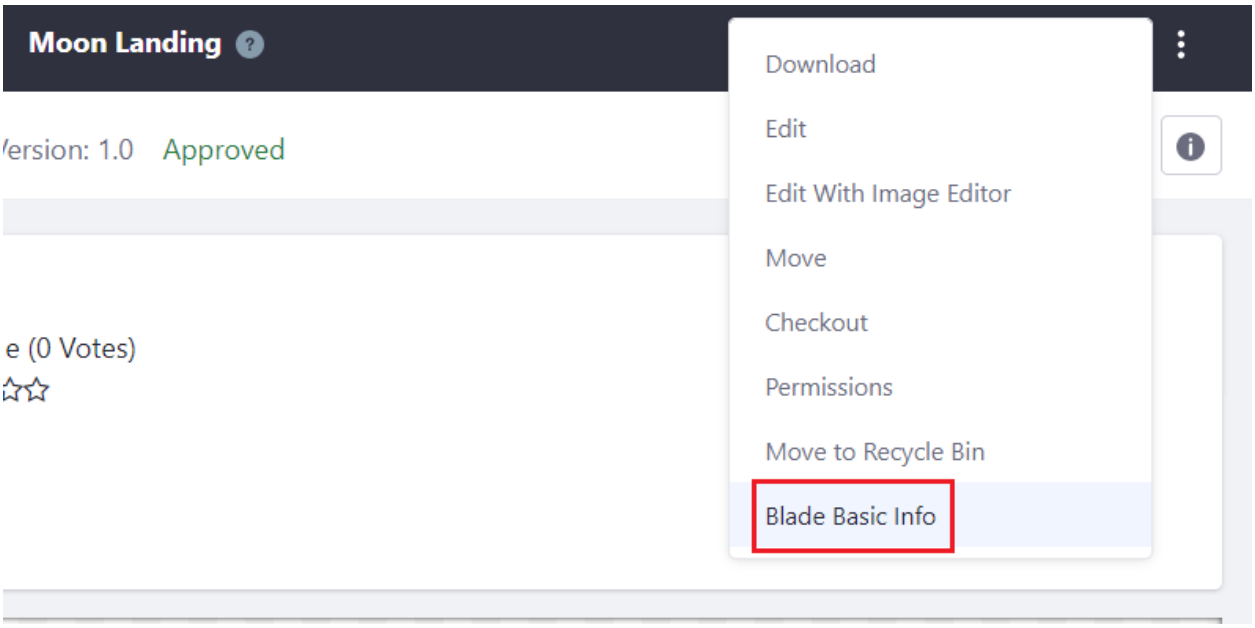


Figure 172.3: The new option is also available from the portlet's Document Details.

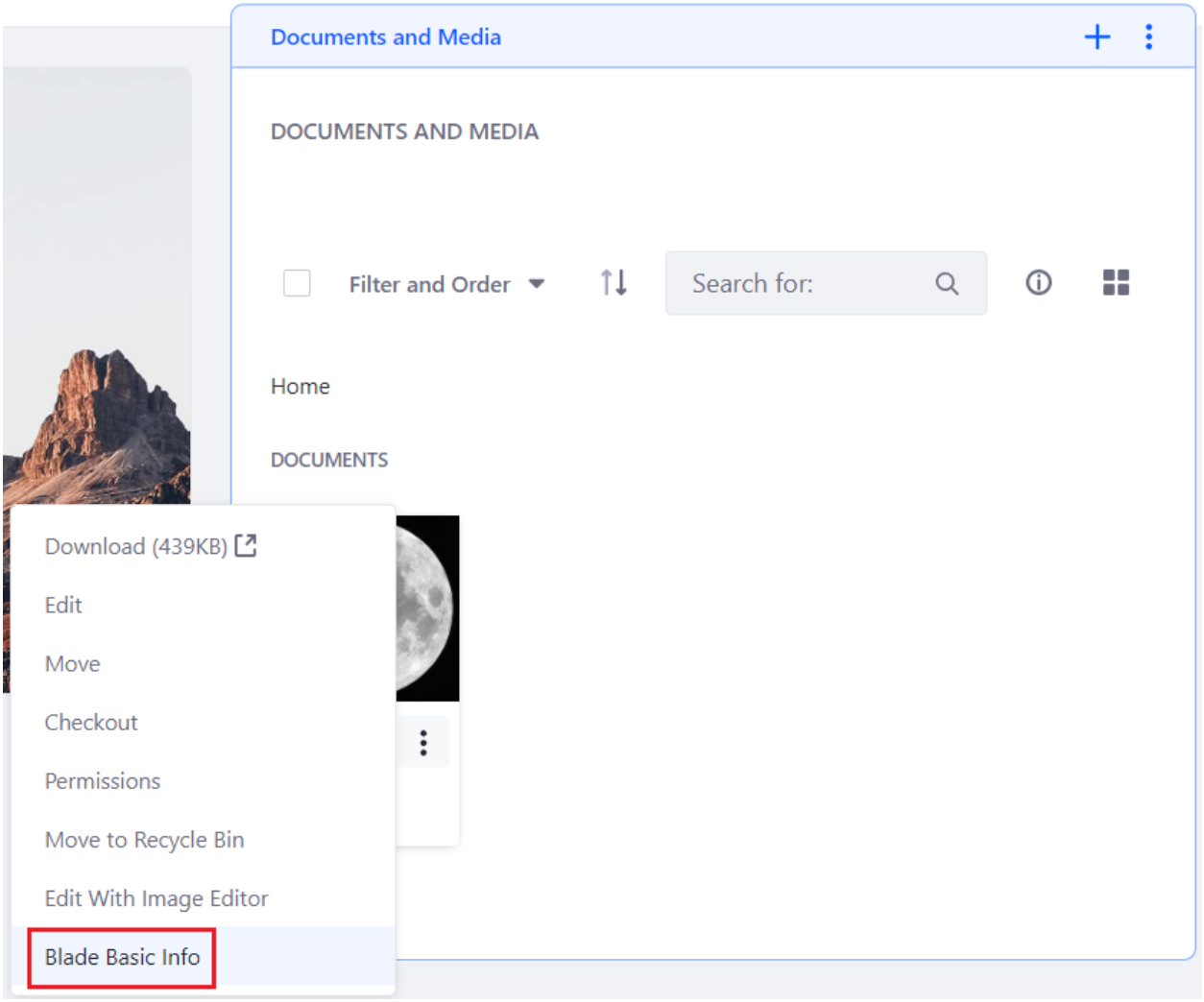


Figure 172.4: You can access the new *Blade Basic Info* option from the Documents and Media portlet added to a page.

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.
- `BladeDocumentActionPortlet`: Provides the portlet class, which extends the GenericPortlet. This class generates what is shown when the context menu option is selected.

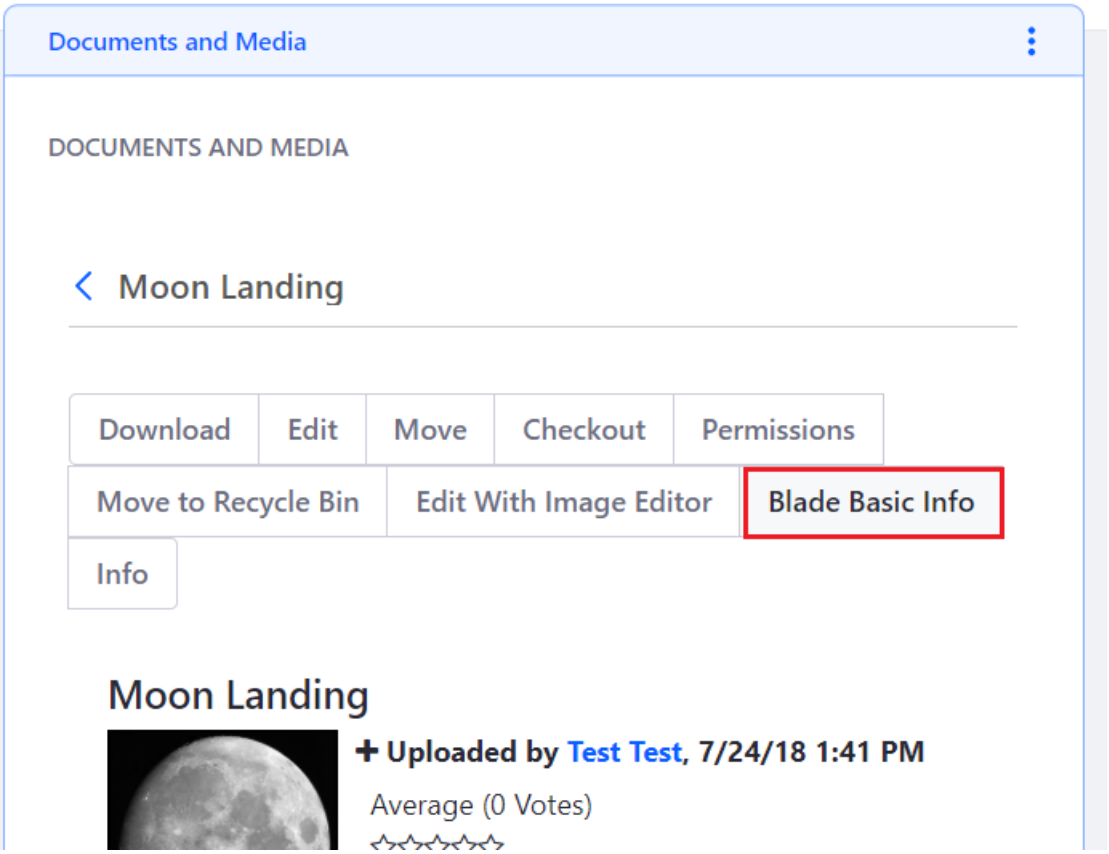


Figure 172.5: The Documents And Media portlet provides the option from its Document Detail too.

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

172.3 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. Navigate to the Control Panel → *Configuration* → *Gogo Shell*.
3. Execute `help` to view all the available commands. The sample Gogo shell command is listed.

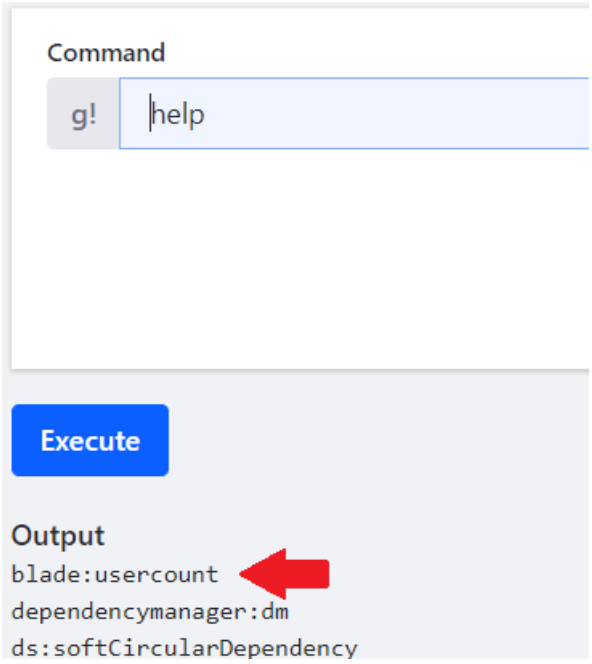


Figure 172.6: The sample Gogo shell command is listed with all the available commands.

4. Execute `usercount` to execute the new custom command. The number of users on your running Liferay Portal installation is printed.

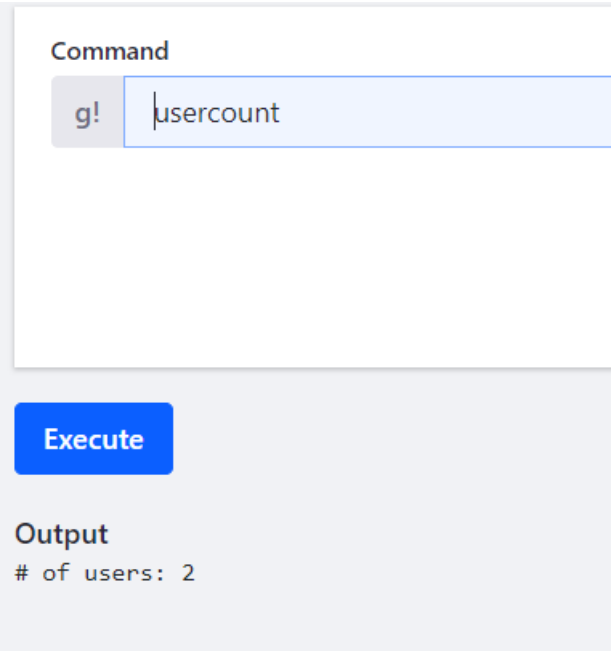


Figure 172.7: The outcome of executing the `usercount` command.

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

172.4 Index Settings Contributor

The Index Settings Contributor sample demonstrates how to add a custom type mapping to Liferay DXP. You can demo this sample by completing the following steps:

1. Navigate to the *Control Panel* → *Configuration* → *Search* menu.

2. Click *Execute* for the *Reindex all search indexes* action.

   All properties defined in your `.json` file are added to Liferay DXP's search engine. This sample adds the following index properties:

   - `sampleDate`
   - `sampleDouble`
   - `sampleLong`
   - `sampleText`

   You'll verify this next.

3. Find your Liferay DXP's instance ID. This can be found in the *Control Panel* → *Configuration* → *Virtual Instances* menu.

4. Navigate to the following URL:

   ```
   http://localhost:9200/liferay-[INSTANCE_ID]/_mapping/LiferayDocumentType?pretty
   ```

   Be sure to insert your instance ID into the URL.

5. Verify the added properties are listed.

What API(s) and/or code components does this sample highlight?

This sample leverages the IndexSettingsContributor API.

Figure 172.8: This sample added four new index properties.

How does this sample leverage the API(s) and/or code component?

Liferay's search engine provides an API to define custom mappings. To use it, follow these fundamental steps:

1. Define the new mapping. In this sample, the mapping is defined in the `META-INF/mappings/resources/index-type-mappings.json` file. Notice that the default document for Liferay DXP is called `LiferayDocumentType`. The mapping's features can be found in Elasticsearch's docs.

2. Inject the mapping into Elasticsearch. The `IndexSettingsContributor` class' components are invoked during the reindexing stage and receive a `TypeMappingsHelper` as a hook to add new mappings.

This sample has two classes:

- `ResourceUtil`: reads the `.json` file.
- `IndexSettingsContributor`: allows the addition of type mappings on Liferay DXP's search engine.

The `IndexSettingsContributor`'s `contribute` method adds the type mappings:

```
@Override
public void contribute(
    String indexName, TypeMappingsHelper typeMappingsHelper) {
    try {
        String mappings = ResourceUtil.readResouceAsString(
            "META-INF/resources/mappings/index-type-mappings.json");

        typeMappingsHelper.addTypeMappings(indexName, mappings);
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

For the `ResourceUtil.readResouceAsString` parameter, you should pass the path for the `.json` file that contains the properties to be added.

Also, it is important to highlight the `IndexSettingsContributor`'s `@Component` annotation that registers a new service to the OSGi container:

```
@Component(
    immediate = true,
    service = com.liferay.portal.search.elasticsearch6.settings.IndexSettingsContributor.class
)
```

This sample demonstrates the essentials needed to contribute your own index settings.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace
- Maven

172.5 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

172.6 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:

You can also verify that the model listener sample was executed by navigating to the new page's *Options → Configure Page → SEO* option. The HTML Title field looks like this:

```
==> ./catalina.out <==
About to create layout: My Test Page
```

Figure 172.9: The sample model listener's message in the console.

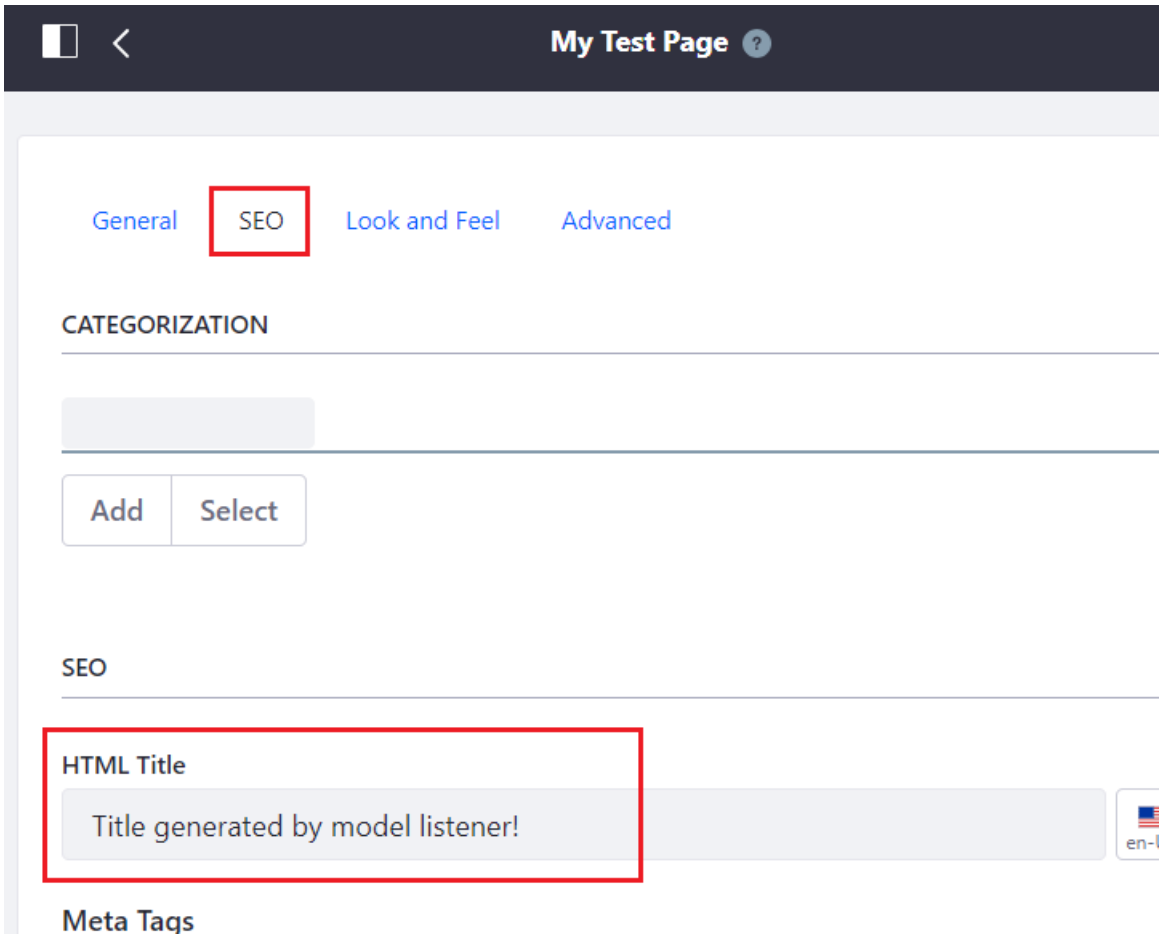


Figure 172.10: The page's HTML title updated by the model listener sample.

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.

```
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

172.7 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*.

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*.

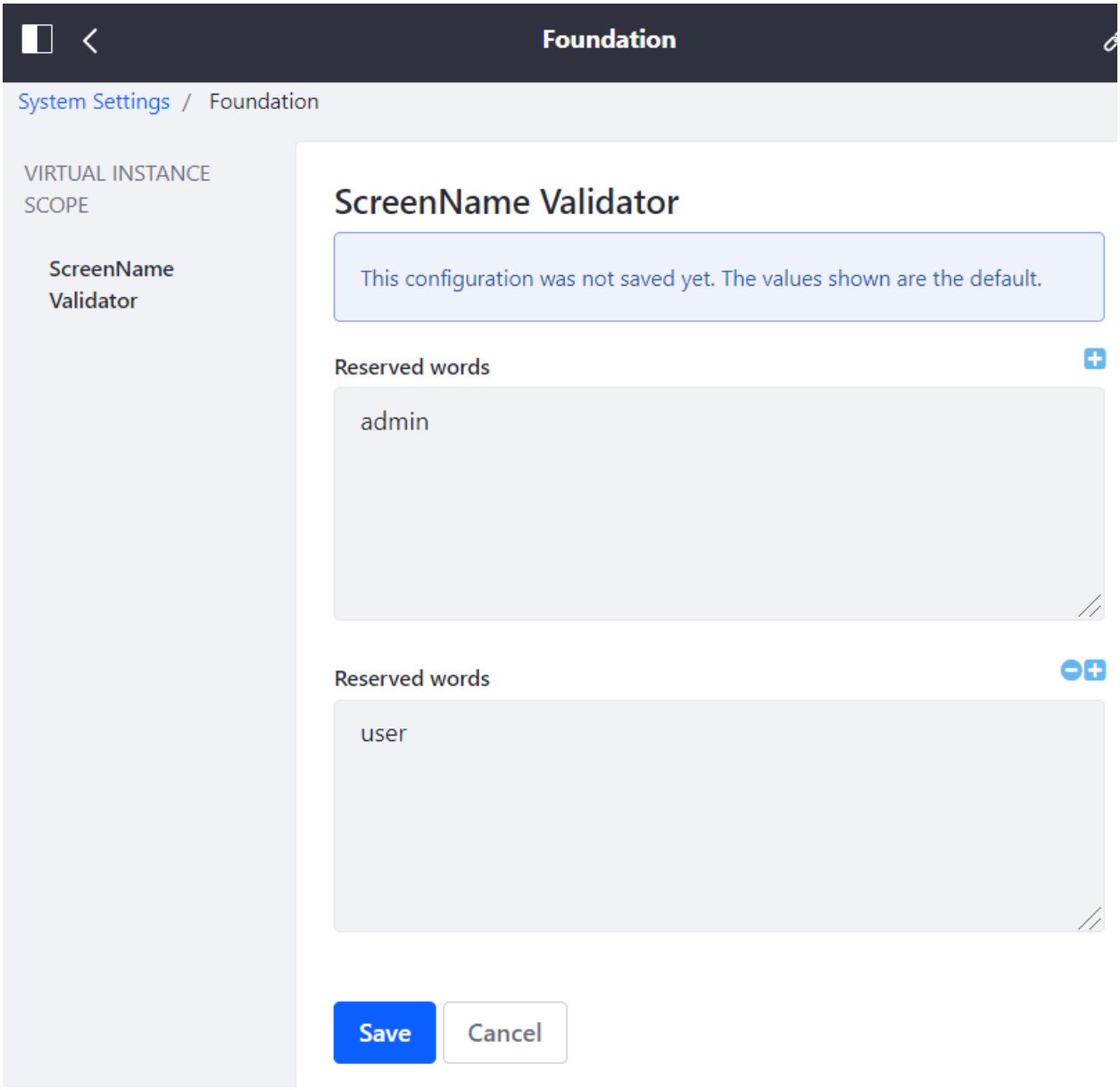


Figure 172.11: Enter reserved words for the screen name validator.

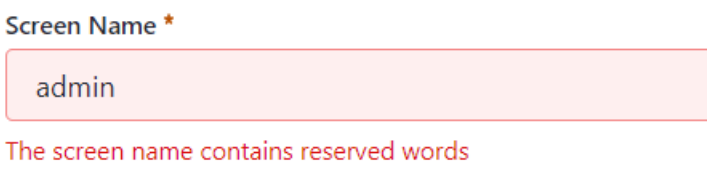


Figure 172.12: The error message displays when inputting a reserved word for the screen name.

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.CustomScreenNameVa` class.

You can also customize this sample's configuration by adding more properties in its `com.liferay.blade.samples.screenname.validator.CustomScreenNameConfiguration` 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

172.8 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.

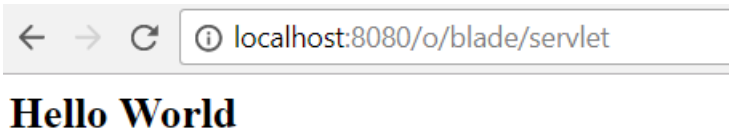


Figure 172.13: The servlet displays *Hello World* from the configured servlet page URL.

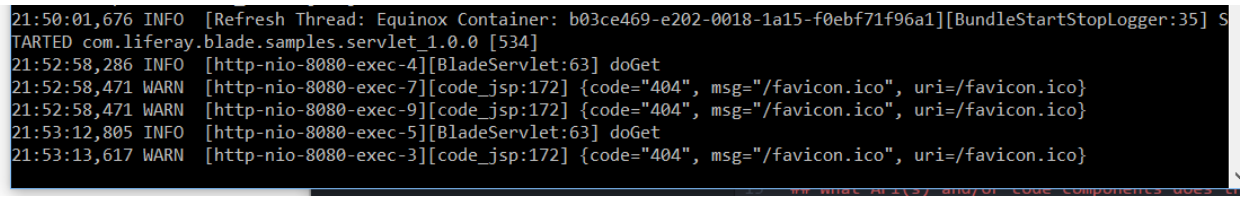


Figure 172.14: The servlet also logs info in the console.

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.

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 173

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

Visit a particular sample page to learn more!

173.1 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.

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"
```

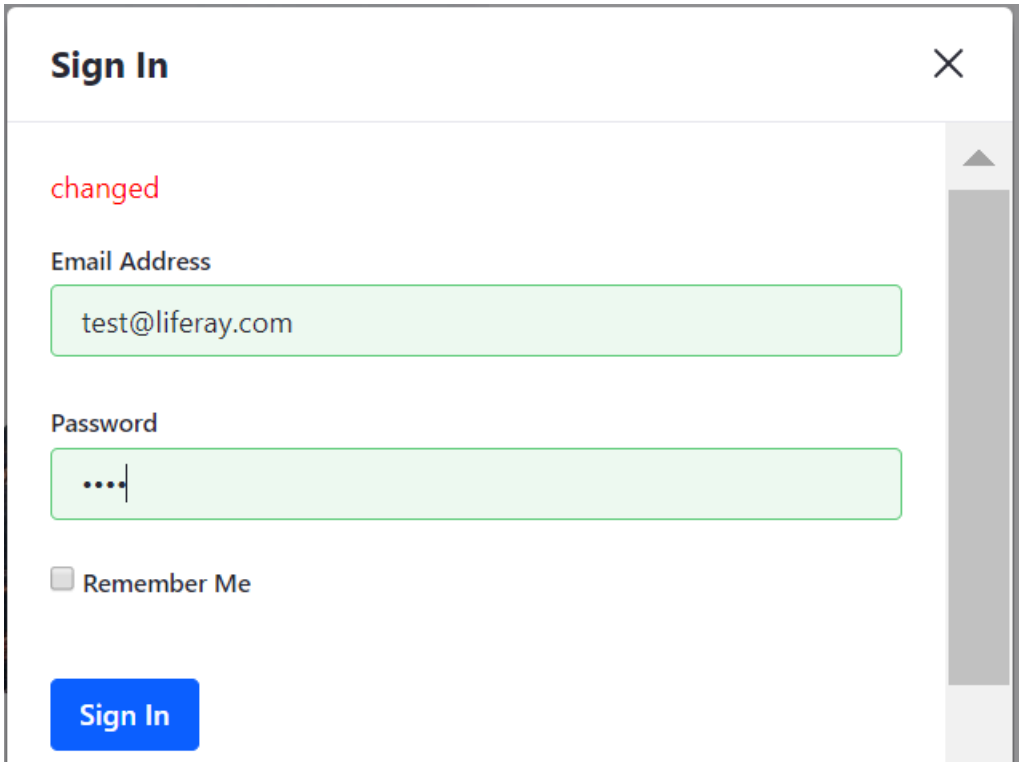


Figure 173.1: The customized Sign In form with the new *changed* text.

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 JSP Overrides Using Custom JSP Bag tutorial.

For more information on other ways to customize JSPs, see the Customizing 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

173.2 Resource Bundle Override

This example overrides the default `javax.portlet.title.com_liferay_login_web_portlet_LoginPortlet` language key for Liferay DXP's default Login portlet. After deploying this sample to Liferay DXP, the Login portlet's *Sign In* title is modified to display *Login Portlet Override*.

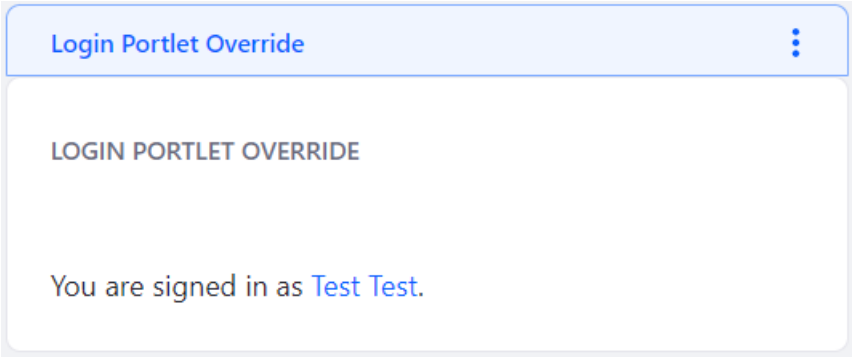


Figure 173.2: The customized Login portlet displays the new language key.

For reference, the Login portlet's language keys are stored in the liferay-portal Github repo's `modules/apps/login/login-web/src/main/resources/content` folder.

What API(s) and/or code components does this sample highlight?

This sample leverages the `Provide-Capability` OSGi manifest header.

How does this sample leverage the API(s) and/or code component?

This sample conveys the recommended approach to override a portlet'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 a portlet's language keys are

- Provide the new language keys that will override the original ones.
- Prioritize the new module's resource bundle.

This sample's `src/main/resources/content` folder holds the language properties file to override. Since this example's goal is to override only the English keys, the `Language_en.properties` is added. You can add more language properties files for additional language key locales you want to override (e.g., `Language_en.properties` for Spanish).

Once your language keys are in place, you must use OSGi manifest headers to specify your custom language keys are for the target module. To compliment the target module's resource bundle, you must aggregate your resource bundle with the target module's resource bundle. This is done by ranking your module first to prioritize its resource bundle over the target module resource bundle. See this sample's `bnd.bnd` as an example for setting the `Provide-Capability` OSGi header:

```
Provide-Capability:\
    liferay.resource.bundle;\
        resource.bundle.base.name="content.Language",\
    liferay.resource.bundle;\
        bundle.symbolic.name=com.liferay.login.web;\
        resource.bundle.aggregate:String="(bundle.symbolic.name=com.liferay.blade.login.web.resource.bundle.override),(bundle.symbolic.name=com.lifer
        resource.bundle.base.name="content.Language";\
        service.ranking:Long="2";\
        servlet.context.name=login-web
```

For more information on the `Provide-Capability` header and its parts, see the Prioritze Your Module's Resource Bundle section.

This approach can be used to override any portlet'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 Overriding Global Language Keys tutorial.

For more information on using a resource bundle to override a module'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 174

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

Visit a particular sample page to learn more!

174.1 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.

Where Is This Sample?

There are three different versions of this sample, each built with a different build tool:

- Gradle

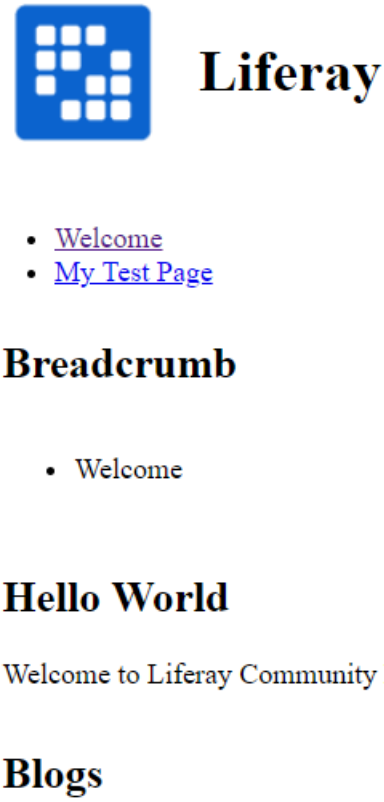


Figure 174.1: A theme based off of the Styled base theme is created when the Theme Blade sample is deployed to Liferay Portal.

- Liferay Workspace
- Maven

174.2 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

174.3 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.

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!*.

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:

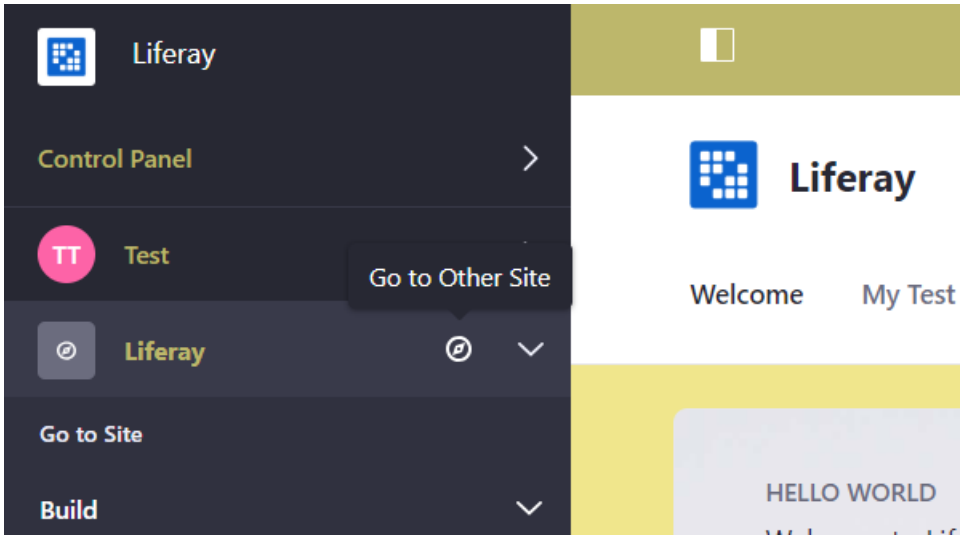


Figure 174.2: Your Liferay DXP pages and menu fonts now have a yellow tint.

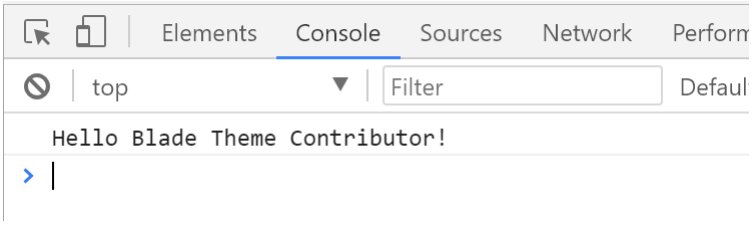


Figure 174.3: The message is printed to your browser's console window using JavaScript.

```
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

CHAPTER 175

EXT

This section focuses on Liferay Ext modules. You can view these sample apps by visiting the `ext` folder corresponding to your preferred build tool:

- Gradle sample apps
- Liferay Workspace sample apps

Visit the sample page to learn more!

175.1 Login Web Ext

The Login Ext Module sample demonstrates how to customize a default Liferay module's source code. This example replaces the default `login.jsp` file in the `com.liferay.login.web` bundle by adding the text *Hello from com.liferay.login.web.ext module! 2 + 2 = 4* to the Sign In form.

It also prints the following text to the console when you select *Forgot Password* from the Sign In form:

```
In com.liferay.login.web.internal.portlet.action.ForgotPasswordMVCRenderCommand render
```

Before deploying the sample, you must stop the original bundle you intend to override. This is because the Ext sample's generated JAR includes the original bundle source plus your modified source files. Follow the instructions below to do this:

1. Connect to your portal instance using Gogo Shell.

2. Search for the bundle ID of the original bundle to override. To find the `com.liferay.login.web` bundle, execute this command:

   ```
   lb -s | grep com.liferay.login.web
   ```

   This returns output similar to this:

   ```
   423|Active   |   10|com.liferay.login.web (3.0.4)
   ```

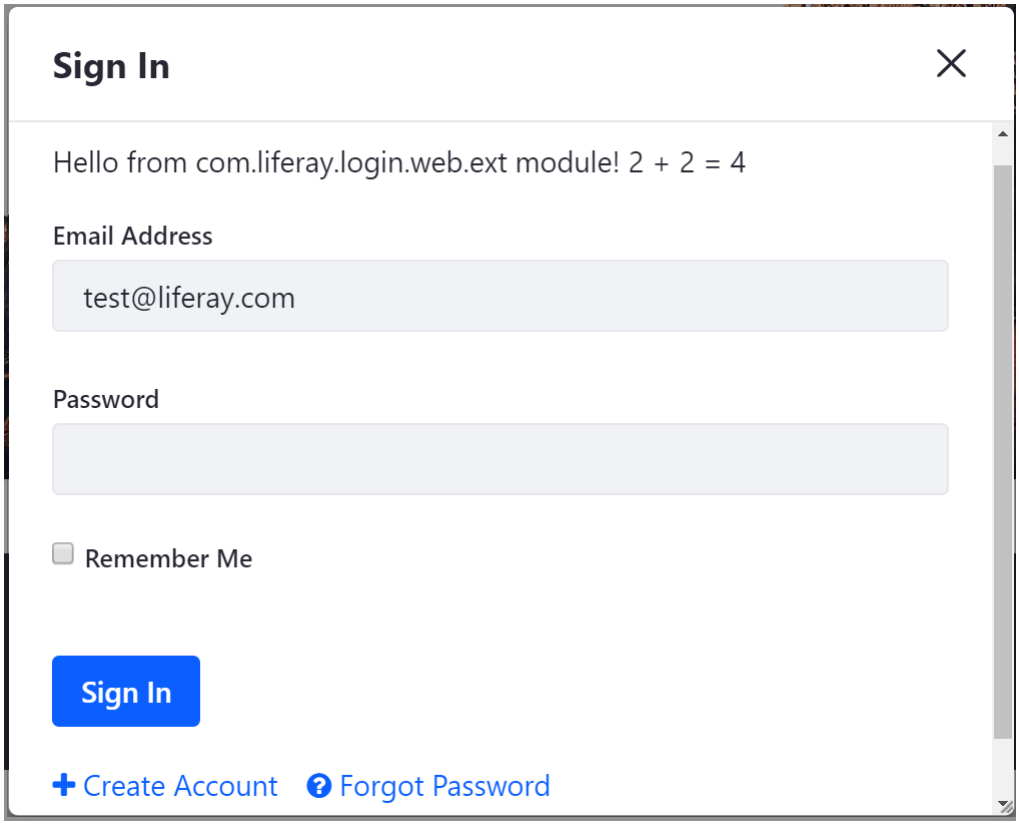


Figure 175.1: The Login Ext module customizes the original Login module.

Make note of the ID (e.g., 423).

3. Stop the bundle:

```
stop 423
```

Once the original bundle is stopped, deploy the Ext module. Note that you cannot leverage Blade or Gradle's `deploy` command to do this. The `deploy` command deploys the module to the `osgi\marketplace\override` folder by default, which does not configure Ext modules properly for usage. You should build and copy the Ext module's JAR to the `deploy` folder manually, or leverage Liferay Dev Studio's drag-and-drop deployment feature.

What API(s) and/or code components does this sample highlight?

This sample demonstrates how to create an Ext module and configure it to replace a default module bundle.

How does this sample leverage the API(s) and/or code component?

You can create your own Ext module project by

- Declaring the original module name and version.
- Providing the source code that will replace the original.

To declare the original module in the `build.gradle` file properly (only supports Gradle), you must specify the original module's Bundle Symbolic Name and the original module's exact version. In this example, this is configured like this:

```
originalModule group: "com.liferay", name: "com.liferay.login.web", version: "3.0.4"
```

If you're leveraging Liferay Workspace, you should put your Ext module project in the `/ext` folder (default); you can specify a different Ext folder name in workspace's `gradle.properties` by adding

```
liferay.workspace.ext.dir=EXT_DIR
```

If you are developing an Ext module project in standalone mode (not associated with Liferay Workspace), you must declare the Ext Gradle plugin in your `build.gradle`:

```
apply plugin: 'com.liferay.osgi.ext.plugin'
```

Then you must provide your own code intended to replace the original one. **Be sure to mimic the original module's folder structure when overriding its JAR.**

The following file types can be overlaid with an Ext module:

- CSS
- Java
- JavaScript
- Language files (`Language.properties`)
- Scss
- Soy
- etc.

The Ext Gradle Plugin helps compile your code into the JAR. For example, `.scss` files are compiled into `.css` files, which are included in your module's JAR file artifact. This is done by the `buildCSS` task.

Where Is This Sample?

There are two different versions of this sample, each built with a different build tool:

- Gradle
- Liferay Workspace

175.2 Felix Gogo Shell

To interact with Liferay DXP's module framework, you can leverage the Gogo shell portlet. You can access this portlet in the Control Panel → *Configuration* → *Gogo Shell*.

Note: You can also interact with Liferay DXP's module framework via a local telnet session. To do this, you must have Developer Mode enabled.

To open the Gogo shell via telnet, execute the following command:

```
telnet localhost 11311
```

Running this command requires a local running instance of Liferay DXP and your machine's telnet command line utilities enabled.

To disconnect the session, execute the `disconnect` command. Avoid using the following commands, which stop the OSGi framework:

- `close`
- `exit`
- `shutdown`

If you have Blade CLI installed and the telnet capability enabled, you can run the Gogo shell via Blade command too:

```
blade sh <gogoShellCommand>
```

Here are some useful Gogo shell commands:

`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

`diag [BUNDLE_ID]`: lists information about why the specified bundle is not working (e.g., unresolved dependencies, etc.)

`headers [BUNDLE_ID]`: lists metadata about the bundle from the bundle's `MANIFEST.MF` file

`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.

`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

`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.

`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

`start [BUNDLE_ID]`: starts the specified bundle

`stop [BUNDLE_ID]`: stops the specified bundle

`system:getproperties`: lists all of the system properties

`uninstall [BUNDLE_ID]`: uninstalls the specified bundle from Liferay's module framework. This does not remove the specified bundle from Liferay's module framework; it's hidden from Gogo's `lb` command, but is still present. Adding a new version of the uninstalled bundle, therefore, will not reinstall it; it will update the currently hidden uninstalled version. To remove a bundle from Liferay's module framework permanently, manually delete it from the `LIFERAY_HOME/osgi` folder. For more information on the `uninstall` command, see OSGi's uninstall documentation.

For more information about the Gogo shell, visit Apache's official documentation.

CHAPTER 176

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.

176.1 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.liferay.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.liferay.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.liferay.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|2.0|2.2| 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.liferay.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.

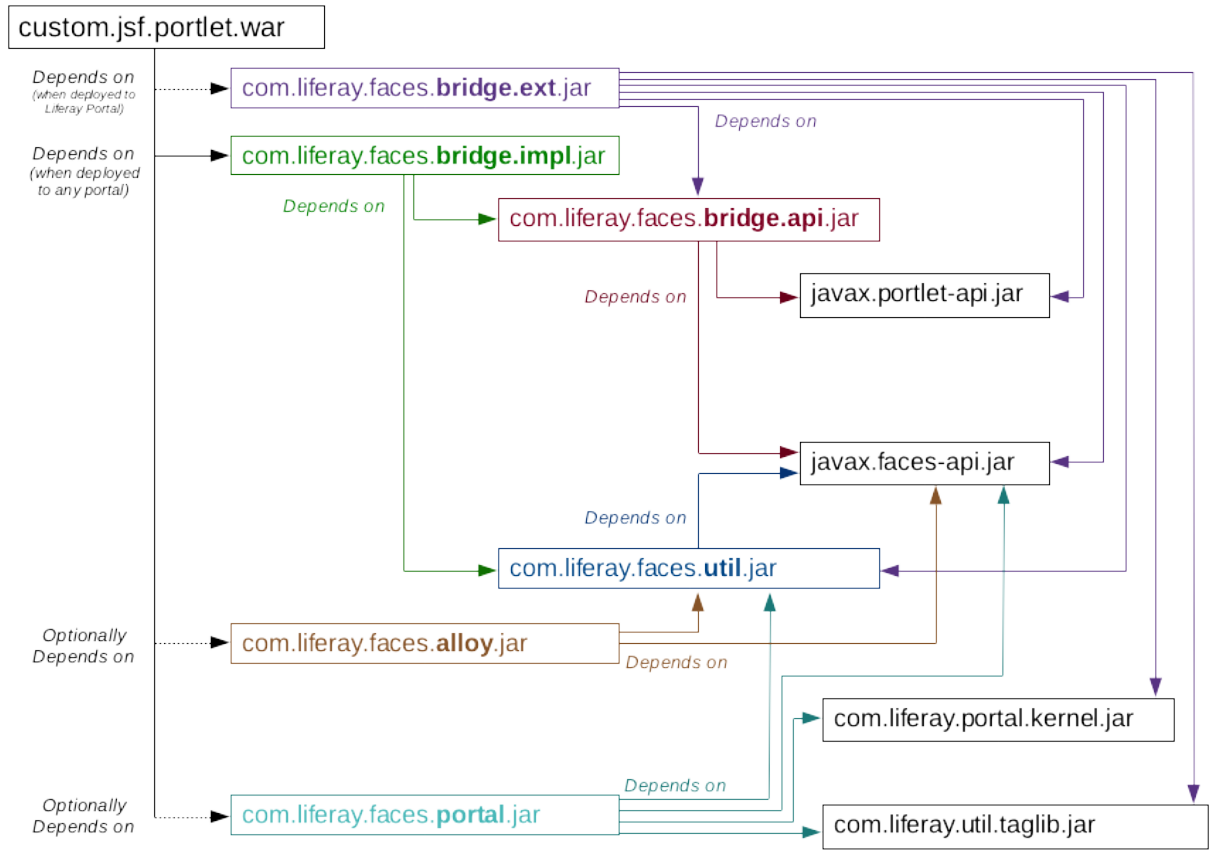


Figure 176.1: The Liferay Faces dependency diagram helps visualize how components interact and depend on each other.

Next, you can view some example configurations to see the new versioning scheme in action.

176.2 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.

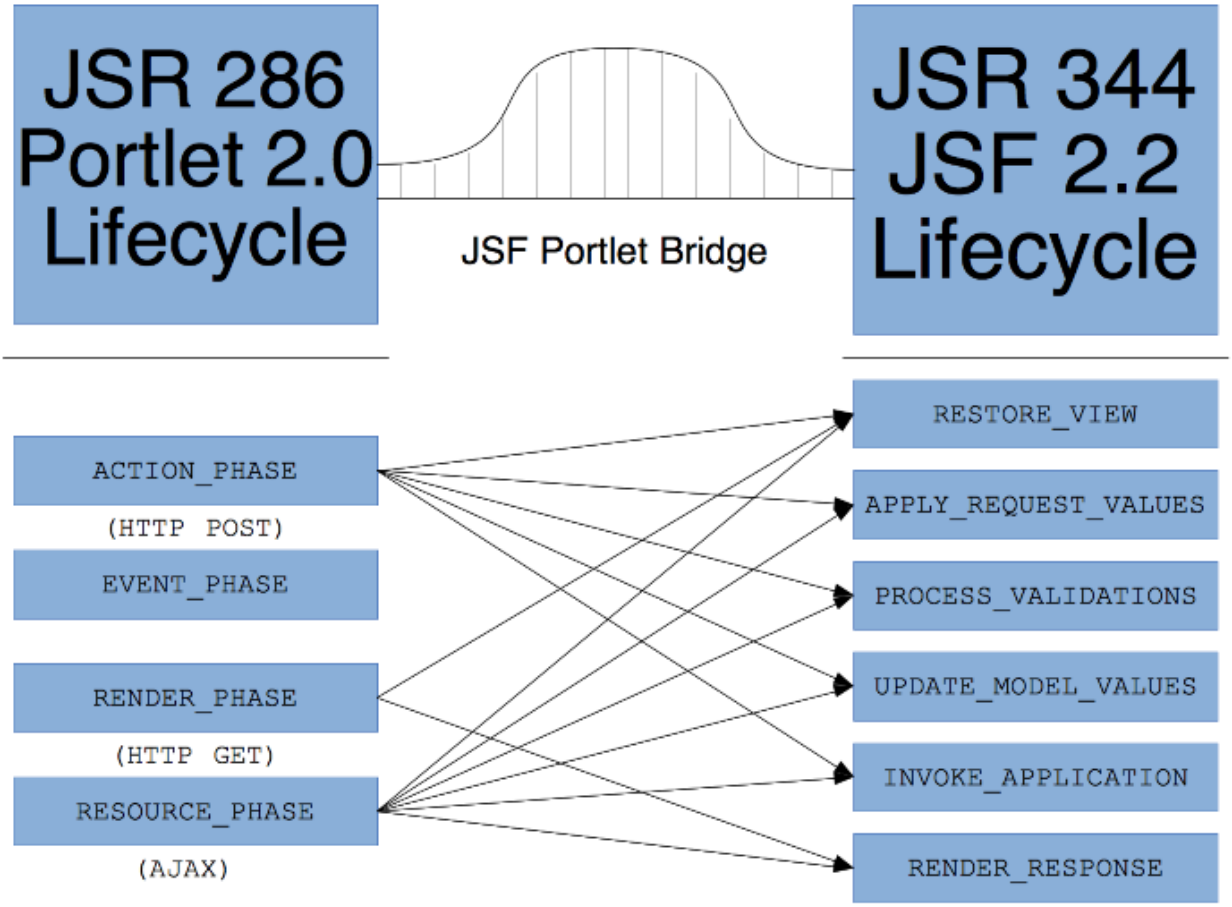


Figure 176.2: The different phases of the JSF Lifecycle are executed depending on which phase of the Portlet lifecycle is being executed.

Liferay Faces Bridge is an essential part of the JSF development process for Liferay DXP. Visit the JSF Portlets with Liferay Faces section of tutorials for more information on JSF development for Liferay DXP.

Related Topics

Understanding Liferay Faces Alloy
    Understanding Liferay Faces Portal
    What is Service Builder?

176.3 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*.

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

176.4 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.

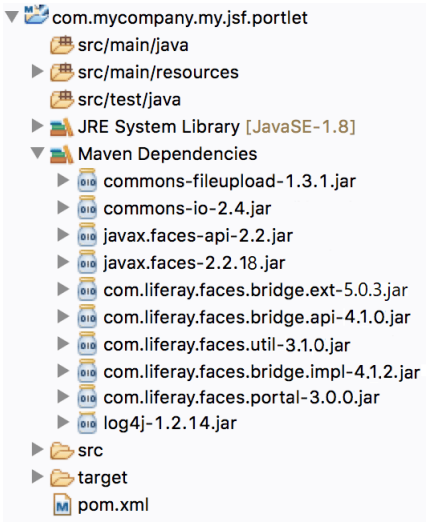


Figure 176.3: The required `.jar` files are downloaded for your JSF portlet based on the JSF UI Component Suite you configured.

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
Understanding Liferay Faces Bridge
Understanding Liferay Faces Alloy

176.5 Page Fragments

Page Fragments are templates made up of CSS, HTML, and JavaScript used to build Content Pages. The HTML, CSS, and JavaScript are all completely standard, just like anywhere else on the web, but are also enhanced with Liferay-specific features. The articles in this section provide additional information about the Liferay-specific features of Page Fragments.

176.6 Embedding Widgets in Page Fragments

You can embed both a selection of Liferay widgets and your own custom widgets in Page Fragments. For a more information on embedding custom widgets, see the Fragment Specific Tags tutorial.

Embedding Liferay Widgets

Many Liferay widgets can be embedded in Page Fragments. Each embeddable widget has a specific tag for use in fragments which looks like `<lfr-widget-[widget-name]>`. When you embed the widget, the complete opening and closing tags must be used like this:

```
<lfr-widget-[widget-name]>
</lfr-widget-[widget-name]>
```

Here is the full list of widgets that can be embedded:

Widget Name	Tag
DDL Display	`<lfr-widget-dynamic-data-list>`
Form	`<lfr-widget-form>`
Asset Publisher	`<lfr-widget-asset-list>`
Breadcrumb	`<lfr-widget-breadcrumb>`
Categories Navigation	`<lfr-widget-categories-nav>`
Flash	`<lfr-widget-flash>`
Media Gallery	`<lfr-widget-media-gallery>`
Navigation Menu	`<lfr-widget-nav>`
Polls Display	`<lfr-widget-polls>`
Related Assets	`<lfr-widget-related-assets>`
Site Map	`<lfr-widget-site-map>`
Tag Cloud	`<lfr-widget-tag-cloud>`
Tags Navigation	`<lfr-widget-tags-nav>`
Web Content Display	`<lfr-widget-web-content>`
Rss Publisher (Deprecated)	`<lfr-widget-rss>`
Iframe	`<lfr-widget-iframe>`

176.7 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.

Get User JSON Web Service Invocation via JavaScript

Among the JavaScript objects that Liferay creates for each page is a `Liferay` object. This object includes a `Service` function that you can use to invoke Liferay's API.

Examine the following JSON web service invocation, written in JavaScript:

```
<script type="text/javascript">

  Liferay.Service(
      '/user/get-user-by-email-address',
```

```
    {
        companyId: Liferay.ThemeDisplay.getCompanyId(),
        emailAddress: 'test@example.com'
    },
    function(obj) {
        console.log(obj);
    }
);

</script>
```

When you run this example, the test@example.com user (JSON object) is returned. You can examine the returned object from your browser 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 an 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>
```

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.

As explained in that tutorial, 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.
openId&locale=en_US&firstName=Joe&middleName=T&lastName=Bloggs&prefixId=0&suffixId=0&male=true&birthdayMonth=1&birthdayDay=1&birthdayYear=1970&jobTi
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 window.

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

CHAPTER 177

CUSTOMIZING CORE FUNCTIONALITY WITH EXT

Ext plugins are deprecated for 7.0 and should only be used if absolutely necessary.
The following app servers should be used for Ext plugin development in Liferay DXP:

- Tomcat 9.0

In most cases, Ext plugins are not 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 (how to).
- Overwriting a class in a 7.0 core JAR. For a list of core JARs, see the Finding Core Artifacts section (how to).
- Modifying Liferay DXP's `web.xml` file (how to).
- Adding to Liferay DXP's `web.xml` file (how to).

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.

Ext plugins are used to customize Liferay DXP's core functionality. You can learn more about what the core encompasses in the Finding Core Artifacts article section. In this section, you'll learn how to

- Create an Ext plugin
- Develop an Ext plugin
- Deploy an Ext plugin
- Redeploy an Ext plugin

You can also dive into the Anatomy of an Ext Plugin to familiarize yourself with its structure. You'll start by creating an Ext plugin.

177.1 Extending Core Classes Using Spring with Ext Plugins

A supported use case for using Ext plugins in Liferay DXP is extending its core classes (e.g., `portal-impl`, `portal-kernel`, etc.) using Spring. You can reference the Finding Core Artifacts section for help distinguishing core classes. Make sure you've reviewed the generalized Customization with Ext Plugins section before creating an Ext plugin.

As an example, you'll create a sample Ext plugin that extends the PortalImpl core class residing in the `portal-impl.jar`. You'll override the `PortalImpl.getComputerName()` method via Spring bean, which returns your server's node name. The Ext plugin will override this method and modify the server's returned node name.

1. Navigate to your Liferay Workspace's root folder and run the following command:

   ```
   blade create -t war-core-ext portal-impl-extend-spring-ext
   ```

   Your Ext plugin is generated and now resides in the workspace's `/ext` folder with the name you assigned.

2. Displaying the server node name in your Liferay DXP installation is set to `false` by default. You'll need to enable this property. To do this, navigate into your Liferay bundle's root folder and create a `portal-ext.properties` file. In that file, insert the following property:

   ```
   web.server.display.node=true
   ```

   Now your server's node name will be displayed once your Liferay bundle is restarted.

3. In the `/extImpl/java` folder, create the folder structure representing the package name you want your new class to reside in (e.g., `com/liferay/portal/util`). Then create your new Java class:

   ```
   package com.liferay.portal.util;

   public class SamplePortalImpl extends PortalImpl {

       @Override
       public String getComputerName() {
           return "SAMPLE_EXT_INSTALLED_" + super.getComputerName();
       }

   }
   ```

The method defined in the extension class overrides the `PortalImpl.getComputerName()` method. The `"SAMPLE_EXT_INSTALLED_"` String is now prefixed to your server's node name.

4. In your Ext plugin's `/extImpl/resources` folder, create a `META-INF/ext-spring.xml` file. In this file, insert the following code:

   ```
   <?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"
>

    <bean class="com.liferay.portal.util.SamplePortalImpl" id="com.liferay.portal.util.PortalImpl" />
</beans>
```

Since you plan on modifying a core service class, you can inject its extension class via a Spring bean. This will ensure your new class is recognized. Assign your extension class's fully defined class name (e.g., `com.liferay.portal.util.SamplePortalImpl`) to the bean tag's `class` attribute and the fully defined original class name (e.g., `com.liferay.portal.util.PortalImpl`) to the bean tag's `id` attribute.

When your Ext plugin is deployed, your new service (e.g., `SamplePortalImpl`) will extend the core `PortalImpl` class.

Awesome! You've created an Ext plugin that extends a core class in Liferay DXP! Follow the instructions in the Deploy the Plugin article to deploy it to your server.

177.2 Overriding Core Classes with Ext Plugins

A supported use case for using Ext plugins in Liferay DXP is overriding its core classes (e.g., `portal-impl`, `portal-kernel`, etc.). You can reference the Finding Core Artifacts section for help distinguishing core classes. Make sure you've reviewed the generalized Customization with Ext Plugins section before creating an Ext plugin.

As an example, you'll create a sample Ext plugin that overwrites the PortalImpl core class residing in the `portal-impl.jar`. You'll edit the `PortalImpl.getComputerName()` method, which returns your server's node name. The Ext plugin will override the entire `PortalImpl` class, adding the method modifying the server's returned node name.

1. Navigate to your Liferay Workspace's root folder and run the following command:

   ```
   blade create -t war-core-ext portal-impl-override
   ```

   Your Ext plugin is generated and now resides in the workspace's `/ext` folder with the name you assigned.

2. Displaying the server node name in your Liferay DXP installation is set to `false` by default. You'll need to enable this property. To do this, navigate into your Liferay bundle's root folder and create a `portal-ext.properties` file. In that file, insert the following property:

   ```
   web.server.display.node=true
   ```

   Now your server's node name will be displayed once your Liferay bundle is restarted.

3. In the `/extImpl/java` folder, create the folder structure matching the class's folder structure you'd like to override (e.g., `com/liferay/portal/util`). Then create the new Java class that will override the existing core class; your new class must have the same name as the original.

4. Copy all of the original class's (e.g., `PortalImpl`) logic into your new class. Then modify the method you want to customize. For this example, you want to edit the `getComputerName()` method. Therefore, replace it with the method below:

```
@Override
public String getComputerName() {
    return "sample_portalimpl_ext_installed_successfully_" + _computerName;
}
```

   The method defined in the new class overrides the `PortalImpl.getComputerName()` method. The `sample_portalimpl_ext_installed_successfully_` String is now prefixed to your server's node name.

When your Ext plugin is deployed, your new Java class will override the core `PortalImpl` class.

Awesome! You've created an Ext plugin that overrides a core class in Liferay DXP! Follow the instructions in the Deploy the Plugin article to deploy it to your server.

177.3 Adding to the web.xml with Ext Plugins

A supported use case for using Ext Plugins in Liferay DXP is adding additional functionality to its `web.xml` file. Before beginning, make sure you've reviewed the generalized Customization with Ext Plugins section.

As an example, you'll create a sample Ext plugin that adds to your Liferay DXP's existing `web.xml` file (e.g., in the `/tomcat-[version]/webapps/ROOT/WEB-INF` folder). You'll add a new printout in the console during startup.

1. Navigate to your Liferay Workspace's root folder and run the following command:

```
blade create -t war-core-ext add-printout
```

   Your Ext plugin is generated and now resides in the workspace's `/ext` folder with the name you assigned.

2. For your Liferay DXP installation to recognize new functionality in the `web.xml`, you must create a class that implements the ServletContextListener interface. This class will initialize a servlet context event for which you'll add your new functionality. In the `extImpl/java` folder, create the folder structure representing the package name you want your new class to reside in (e.g., `com/liferay/portal/servlet/context`). Then create your new Java class:

```
package com.liferay.portal.servlet.context;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

public class ExtAddEntryWebXmlPortalContextLoaderListener
        implements ServletContextListener {

    public void contextDestroyed(ServletContextEvent servletContextEvent) {
    }

    public void contextInitialized(ServletContextEvent servletContextEvent) {
        System.out.println("EXT_ADD_ENTRY_WEBXML_INSTALLED_SUCCESSFULLY");
```

```
    }

}
```

The above class includes two methods that initialize and destroy your servlet context event. Be sure to add the new `web.xml`'s functionality when the portal context is initializing. To add a printout verifying the Ext plugins installation, a simple print statement was defined in the `contextInitialized(...)` method:

```
System.out.println("EXT_ADD_ENTRY_WEBXML_INSTALLED_SUCCESSFULLY");
```

3. Now that you've defined a servlet context event, you should add a listener to your `web.xml` that listens for it. In the `ext-web/docroot/WEB-INF` folder, open the `web.xml` file, which was generated for you by default.
4. Add the following tag between the tags:

```
<listener>
    <listener-class>com.liferay.portal.servlet.context.ExtAddEntryWebXmlPortalContextLoaderListener</listener-class>
</listener>
```

Excellent! Now when your Ext plugin is deployed, your Liferay DXP installation will create a `ServletContextListener` instance, which will initialize a custom servlet context event. This event will be recognized by the `web.xml` file, which will add the new functionality to your Liferay DXP installation. Follow the instructions in the Deploy the Plugin article for help deploying the Ext plugin to your server.

177.4 Modifying the web.xml with Ext Plugins

A supported use case for using Ext Plugins in Liferay DXP is modifying its `web.xml` file. Before beginning, make sure you've reviewed the generalized Customization with Ext Plugins section.

As an example, you'll create a sample Ext plugin that modifies Liferay DXP's existing `web.xml file` (e.g., in the `/tomcat-[version]/webapps/ROOT/WEB-INF` folder). You'll modify the session timeout configuration, which is set to 30 (minutes) by default:

```
<session-config>
    <session-timeout>30</session-timeout>
    <cookie-config>
        <http-only>true</http-only>
    </cookie-config>
</session-config>
```

The Ext plugin will update the session timeout to one minute.

1. Navigate into your Liferay Workspace's `/ext` folder and run the following command:

```
blade create -t war-core-ext modify-session-timeout
```

Your Ext plugin is generated and now resides in the workspace's `/ext` folder with the name you assigned.

2. In the ext-web/docroot/WEB-INF folder, open the web.xml file, which was generated for you by default.

3. Insert the following logic between the <web-app> tags:

```
<session-config>
    <session-timeout>1</session-timeout>
    <cookie-config>
        <http-only>true</http-only>
    </cookie-config>
</session-config>
```

Notice that the <session-timeout> tag has been updated to 1.

That's it! Now when your Ext plugin is deployed, your Liferay DXP installation will timeout after one minute of inactivity. Follow the instructions in the Deploy the Plugin article for help deploying the Ext plugin to your server.

177.5 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).

177.6 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.

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, Java EE 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.

Breaking Changes List

Standardized Data Attribute Names Passed into Selectors

- **Date:** 2016-Oct-26
- **JIRA Ticket:** LPS-66646

What changed? The data attributes passed into the event when someone uses a selector (e.g., asset selector, document selector, file selector, role selector, site selector, user group selector, etc.) have been standardized from being selector specific (e.g., `groupid`, `groupdescriptivename`, `teamid`, `teamname`, etc.) to being more generic (e.g., `entityid` and `entityname`).

Who is affected? This affects anyone passing selector specific data attributes to a selector.

How should I update my code? Instead of using selector specific data attributes, you should change your data attributes to use `entityid` and `entityname`.

Example

Old way:

```
<portlet:namespace />selectFileEntryType(event.fileentrytypeid, event.fileentrytypename);
```

New way:

```
<portlet:namespace />selectFileEntryType(event.entityid, event.entityname);
```

Old way:

```
data.put("roleid", role.getRoleId());
data.put("roletitle", role.getTitle(locale));
```

New way:

```
data.put("entityid", role.getRoleId());
data.put("entityname", role.getTitle(locale));
```

Why was this change made? This change was made to standardize the data attribute names and allow utility methods to accept standardized event parameters.

Removed URL Parameters p_p_col_id, p_p_col_pos, and p_p_col_count from Every Portlet URL.

- **Date:** 2016-Dec-12
- **JIRA Ticket:** LPS-69482

What changed? The parameters `p_p_col_count`, `p_p_col_id`, and `p_p_col_pos` are no longer present in every portlet URL.

Who is affected? This affects developers who are reading these parameters in their custom code.

How should I update my code? You can no longer obtain these parameters from the portlet URL. If you need to read them, you should do it from `PortletDisplay`.

- The parameter `p_p_col_count` can be obtained via the `portletDisplay.getColumnCount()` method.
- The parameter `p_p_col_id` can be obtained via the `portletDisplay.getColumnId()` method.
- The parameter `p_p_col_pos` can be obtained via the `portletDisplay.getColumnPos()` method.

Why was this change made? This change simplifies portlet URLs so they only contain the required parameters. This was done as a preliminary step of a bigger story to create portlet URLs without passing the request as a necessary parameter.

Moved Users File Uploads Portlet Properties to OSGi Configuration

- **Date:** 2017-Feb-06
- **JIRA Ticket:** LPS-69211

What changed? The Users File Uploads portlet properties have been moved from Server Administration to an OSGi configuration named `UserFileUploadsConfiguration.java` in the `users-admin-api` module.

Who is affected? This affects anyone using the following portlet properties:

- `users.image.check.token`
- `users.image.max.size`
- `users.image.max.height`
- `users.image.max.width`

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 Portal's *Control Panel* → *Configuration* → *System Settings* → *Foundation* → *User 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 CAPTCHA Portal Properties to OSGi Configuration

- **Date:** 2017-Feb-13
- **JIRA Ticket:** LPS-67830

What changed? The CAPTCHA properties have been moved from `portal.properties` and Server Administration to an OSGi configuration named `CaptchaConfiguration.java` in the `captcha-api` module.

Who is affected? This affects anyone using the following portal properties:

- `captcha.max.challenges`
- `captcha.check.portal.create_account`
- `captcha.check.portal.send_password`
- `captcha.check.portlet.message_boards.edit_category`
- `captcha.check.portlet.message_boards.edit_message`
- `captcha.engine.impl`
- `captcha.engine.recaptcha.key.private`
- `captcha.engine.recaptcha.key.public`
- `captcha.engine.recaptcha.url.script`
- `captcha.engine.recaptcha.url.noscript`
- `captcha.engine.recaptcha.url.verify`
- `captcha.engine.simplecaptcha.height`
- `captcha.engine.simplecaptcha.width`
- `captcha.engine.simplecaptcha.background.producers`
- `captcha.engine.simplecaptcha.gimpy.renderers`
- `captcha.engine.simplecaptcha.noise.producers`

- `captcha.engine.simplecaptcha.text.producers`
- `captcha.engine.simplecaptcha.word.renderers`

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 Portal's *Control Panel → Configuration → System Settings → Captcha* 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 OpenOffice Properties to OSGi Configuration

- **Date:** 2017-Mar-24
- **JIRA Ticket:** LPS-71382

What changed? The OpenOffice properties have been moved from Server Administration to an OSGi configuration named `OpenOfficeConfiguration.java` in the `document-library-document-conversion` module.

Who is affected? This affects anyone using the following portal properties:

- `openoffice.cache.enabled`
- `openoffice.server.enabled`
- `openoffice.server.host`
- `openoffice.server.port`

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 Portal's *Control Panel → Configuration → System Settings → Other → OpenOffice Integration* 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.

No More Exceptions Are Thrown When a DDMStructure Is Fetched

- **Date:** 2017-Mar-31
- **JIRA Ticket:** LPS-52675

What changed? The following methods no longer throw `PortalException`:

```
public DDMStructure fetchStructure(
  long groupId, long classNameId, String structureKey,
  boolean includeAncestorStructures)

public DDMStructure fetchStructureByUuidAndGroupId(
  String uuid, long groupId, boolean includeAncestorStructures)
```

Who is affected? This affects anyone using these methods.

How should I update my code? Keep using these methods, but be aware that they don't throw exceptions.

Why was this change made? Since the current method implementations don't generate exceptions, there's no need for the methods to declare throwing a `PortalException`.

Removed Indexation of Fields ratings and viewCount

- **Date:** 2017-May-16
- **JIRA Ticket:** LPS-70724

What changed? The fields `ratings` and `viewCount` are no longer indexed in the `BaseIndexer` class for `AssetEntry` objects.

Who is affected? This affects any search-related custom code where the `ratings` and `viewCount` fields are used in queries.

How should I update my code? To adapt to these changes, consider several alternatives:

- Use the Highest Rated Assets and Most Viewed Assets Liferay portlets.
- Replace the index query with a database query.
- Implement an `IndexerPostProcessor` to index these fields in certain documents.

Why was this change made? Keeping the Ratings and View Count options in the search index in sync with the database has a negative impact on normal operations due to the significantly increased number of index Write requests causing throughput issues and, therefore, performance degradation.

In addition, the view count is not always up-to-date in the database. This behavior is controlled by the *Buffered Increment* mechanism. You can find more information about this in the `portal.properties` file.

Moved Upload Servlet Request Portal Properties to OSGi Configuration

- **Date:** 2017-May-30
- **JIRA Ticket:** LPS-69102

What changed? The Upload Servlet Request properties have been moved from the `portal.properties` file and Server Administration to an OSGi configuration named `UploadServletRequestConfiguratio` in the `portal-upload` module.

Who is affected? This affects anyone using the following portal properties:

- `com.liferay.portal.upload.UploadServletRequestImpl.max.size`
- `com.liferay.portal.upload.UploadServletRequestImpl.temp.dir`

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 Portal's *Control Panel → Configuration → System Settings → Upload Servlet Request* 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 Three DL File Properties to OSGi Configuration

- **Date:** 2017-Aug-01
- **JIRA Ticket:** LPS-69208

What changed? Two DL File properties have been moved from Server Administration to the OSGi configuration `DLConfiguration`, and one to `DLFileEntryConfiguration`. Both configurations are located in the `document-library-api` module.

Who is affected? This affects anyone who is using the following portal properties:

- `dl.file.entry.previewable.processor.max.size`
- `dl.file.extensions`
- `dl.file.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 Portal's *Control Panel → Configuration → System Settings → Collaboration → Documents & Media Service* or *Documents & Media File Entries* 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.

Why was this change made? This change was made as part of the modularization efforts to ease portal configuration changes.

Removed the soyutils Module

- **Date:** 2017-Aug-28
- **JIRA Ticket:** LPS-69102

What changed? The module `frontend-js-soyutils-web` is no longer available.

Who is affected? This affects anyone using the `soyutils` module.

How should I update my code? In the rare case that a component is affected, it is recommended that the code is migrated to use the `metal-soy` module instead. You can do this by extending the `Metal.js` provided `Component` classes.

Why was this change made? The removed module exposed a legacy version of `soyutils`. This caused interoperability issues between applications using different versions of the Closure Template library.

Converted liferay-ui Tags to Module-Specific Tags

- **Date:** 2017-Aug-28
- **JIRA Ticket:** LPS-74331

What changed? Several `liferay-ui` taglibs have been moved from the Portal's kernel into OSGi modules, resulting in taglib names being changed. The updated names are listed below:

- `liferay-ui:asset-add-button` → `liferay-asset:asset-add-button`
- `liferay-ui:asset-addon-entry-display` → `liferay-asset:asset-addon-entry-display`
- `liferay-ui:asset-addon-entry-selector` → `liferay-asset:asset-addon-entry-selector`
- `liferay-ui:asset-categories-available` → `liferay-asset:asset-categories-available`
- `liferay-ui:asset-categories-error` → `liferay-asset:asset-categories-error`
- `liferay-ui:asset-display` → `liferay-asset:asset-display`
- `liferay-ui:asset-links` → `liferay-asset:asset-links`
- `liferay-ui:asset-metadata` → `liferay-asset:asset-metadata`
- `liferay-ui:asset-tags-available` → `liferay-asset:asset-tags-available`
- `liferay-ui:asset-tags-error` → `liferay-asset:asset-tags-error`
- `liferay-ui:asset-tags-navigation` → `liferay-asset:asset-tags-navigation`
- `liferay-ui:input-asset-links` → `liferay-asset:input-asset-links`
- `liferay-ui:journal-content-search` → Removed (`journal-content-search-web` was deprecated)
- `liferay-ui:restore-entry` → Removed
- `liferay-ui:rss` →`liferay-rss:rss`
- `liferay-ui:rss-settings` → `liferay-rss:rss-settings`

Who is affected? This affects anyone who is using the taglibs listed above.

How should I update my code? You must update your `liferay-ui` tags to use the new names. If you prefer keeping the old names temporarily, you can rely on the compatibility layer offered by Liferay. To set this, add the `com.liferay.portal.web.compat` dependency to your project's build file.

Use the updated tag names as soon as you're able, as this compatibility layer is deprecated and will not be available for future releases.

Why was this change made? This change was made to categorize taglibs properly by moving them to their respective OSGi modules.

Changed Default Value for Browser Cache Properties

- **Date:** 2017-Sep-05
- **JIRA Ticket:** LPS-74452

What changed? The default values for the portal properties `browser.cache.disabled` and `browser.cache.signed.in.disabled` were changed to `true`.

Who is affected? This affects anyone relying on proxies and load balancers to cache HTML content.

How should I update my code? You should set both properties `browser.cache.disabled` and `browser.cache.signed.in.disabled` to `false`, as documented in `portal-legacy-7.0.properties`.

Why was this change made? The load balancer and web proxy's behavior when Cache-Control headers are missing is not defined. In the past, many preferred to not cache the content for correctness; however, it is now common to cache the content for performance.

When an aggressive caching load balancer or web proxy appears in the network architecture, the default value may result in security problems such as personalized content being mistakenly shared, including names or other personally identifiable information. As Liferay shifts towards use cases providing personalized experiences, this is becoming a serious problem.

While this is ultimately a load balancer or web proxy configuration issue, it is perceived as a Liferay issue because it is Liferay content being cached, and is viewed negatively because leaking sensitive information in a production environment is a very serious security issue.

A value of `true` will improve a portal administrator's experience, and a value of `false` can be considered during performance tuning, if needed.

Users Can Have Numeric Screen Names with No Limitations, and Sites Can No Longer Have Numeric Friendly URLs

- **Date:** 2017-Oct-10
- **JIRA Ticket:** LPS-66460

What changed?

- The portal property `users.screen.name.allow.numeric` now defaults to `true`.
- Numeric screen names are no longer limited by whether they correspond to an existing group ID.
- Sites can no longer set their group ID as their friendly URL.
- Sites can no longer be implicitly accessed by using their group ID in the URL (this used to be available automatically, even if it wasn't set that way).
- If the friendly URL of a site is already set to the group ID, it will continue to work as normal, but you will be forced to change it if you update the site in the Site Settings portlet.
- If a site is updated and no friendly URL is provided, it will default to `/group-<groupId>`. If that duplicates another friendly URL, the friendly URL will be incremented until a unique friendly URL is found (e.g., `/group-<groupId>-1`).
- The default friendly URL for new sites has **not** changed.

Who is affected? This affects anyone who

- has set the friendly URL of their site to the group ID.
- uses a group ID to navigate or direct to a site.

How should I update my code? No code updates should be required, but if you fall under one of the scenarios in the previous section, you should consider the following changes:

- If you have set the friendly URL of a site to its group ID, you should update the friendly URL of that site to something else. A site administrator can do this through the Site Settings portlet.
- If you have hard-coded the group ID in any links, you must change them to match the updated friendly URL.

Why was this change made? There were common complaints from customers who used LDAP to import users — if users were given a numeric screen name during import, some imports would fail because those screen names conflicted with an existing group ID.

This was because a site's group ID could be used as its friendly URL, while a user's screen name is used as the friendly URL to their personal site. This could introduce a routing conflict, so numeric screen names were disallowed if they conflicted with an existing group ID.

By removing sites' ability to use their group ID as their friendly URL, the possible conflict with numeric screen names is expunged, allowing users to have any number as their screen name. This makes it much less likely for LDAP imports to fail when using numeric screen names for imported users.

Since LDAP import is more commonly used than a site using the group ID as its friendly URL, the less useful feature was removed to stabilize the more common one.

Removed Support for Velocity in Themes

- **Date:** 2017-Oct-19
- **JIRA Ticket:** LPS-74379

What changed?

- Themes can no longer use Velocity for templates.
- Some helper methods have been removed from the public APIs `com.liferay.portal.kernel.util.ThemeHelper` and `com.liferay.taglib.util.ThemeUtil`.

Who is affected? This affects anyone who has themes using Velocity templates or is using the removed methods.

How should I update my code? If you have a theme using Velocity, consider migrating it to FreeMarker for better maintenance and improved security.

If you are using the removed methods, consider using the `com.liferay.portal.kernel.template.Template` functionality directly to process templates.

Why was this change made? Velocity was deprecated in Liferay Portal 7.0 and the recommendation was to migrate to FreeMarker. Also, Velocity has had no new releases for a long time.

The removal of Velocity support for Liferay Portal 7.1 themes allows for an increased focus on existing and new template engines.

Moved Organization Type Properties to OSGi Configuration

- **Date:** 2018-Jan-19
- **JIRA Ticket:** LPS-77183

What changed? The organization type properties have been moved from `portal.properties` to an OSGi configuration named `OrganizationsTypesConfiguration.java` in the `users-admin-api` module.

Who is affected? This affects anyone using the following portal properties:

- `organizations.types`
- `organizations.rootable`
- `organizations.children.types`
- `organizations.country.enabled`
- `organizations.country.required`

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 Portal's *Control Panel → Configuration → System Settings → Foundation → Organization Type* 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.

Why was this change made? This change was made as part of the modularization efforts to ease portal configuration changes.

Updated jQuery and Lodash Bundled Versions

- **Date:** 2018-Feb-07
- **JIRA Ticket:** LPS-77764, LPS-77765

What changed? The bundled jQuery version has been updated from 2.1.4 to 3.3.1. The bundled Lodash version has been updated from 3.10.1 to 4.17.4.

Who is affected? This affects anyone using the previous API versions in their code.

How should I update my code? Follow the changelogs on the jQuery and Lodash sites to update any affected code.

Why was this change made? This change provides the latest jQuery and Lodash versions available.

Removed the VALIDATE_DDM_FORM_VALUES Constant from DDMWebKeys

- **Date:** 2018-Feb-22
- **JIRA Ticket:** LPS-77168

What changed? The `VALIDATE_DDM_FORM_VALUES` constant has been removed from `DDMWebKeys`.

Who is affected? This affects anyone using this constant.

How should I update my code? Use the String `validateDDMFormValues`, which was the constant's value.

Why was this change made? A constant is unnecessary for a value that's not part of an API.

Removed JavaScript Minification Properties minifier.javascript.impl and yui.compressor. from portal.properties*

- **Date:** 2018-Feb-28
- **JIRA Ticket:** LPS-74375

What changed? The JavaScript minifiers have been extracted from `portal-kernel` and moved to their own OSGi module. Thus, they are not configured in `portal.properties` any more, but rather, through OSGi configuration.

Who is affected? This affects anyone who had the Yahoo JavaScript minifier active and configured to override its default settings.

How should I update my code? If you are implementing your own JavaScript minifier, you should extract it to its own OSGi module. See module frontend-js-minifier for an example of how to do this.

Why was this change made? The JavaScript minifiers were not easy to customize. For example, the Google minifier used an old version of the closure-compiler, which was difficult to upgrade because it required `portal-kernel` dependency changes. This could create conflicts.

Having JavaScript minifiers in their own OSGi modules requires less dependency management and makes it easier to provide new implementations of JavaScript minifiers. Also, configuration can now be done using OSGi standards.

Changed Behavior of liferay-ui:input-date Taglib's showDisableCheckbox Argument

- **Date:** 2018-Mar-06
- **JIRA Ticket:** LPS-78475

What changed? Previously, when the `liferay-ui:input-date` taglib's `showDisableCheckbox` argument was set to `true`, the disable checkbox was hidden. Now, the value `true` displays it, and `false` hides it.

Who is affected? This affects anyone trying to hide the `liferay-ui:input-date` taglib's disable checkbox.

How should I update my code? If you are setting the `showDisableCheckbox` argument to `true` to hide the `liferay-ui:input-date` taglib's disable checkbox, you should now set it to `false`, and vice versa.

Why was this change made? The behavior did not match with the name of the argument and was counter-intuitive.

DDLExporterFactory Became an Interface

- **Date:** 2018-Apr-20
- **JIRA Ticket:** LPS-79221

What changed? `DDLExporterFactory` was renamed to `DDLExporterFactoryImpl` and moved to the module `dynamic-data-lists-service`, once this class holds the logic associated to the DDL exporters management. On the other hand, a new interface named `DDLExporterFactory` was created on the module `dynamic-data-lists-api` and `DDLExporterFactoryImpl` is implementing it.

Who is affected? This affects anyone who is using `DDLExporterFactory` to manipulate (add, get and remove) the DDL exporters.

How should I update my code? At a first moment, isn't expected the developer use `DDLExporterFactory` to manipulate the DDL exporters. Actually, there is just one extension point dedicated to the DDL exporters, the possibility to export the data in formats not provided by default. In order to export the data to a new format, the developer will need to create a new Java class that extends the abstract class `BaseDDLExporter`.

Why was this change made? To encapsulate the default implementation of `DDLExporterFactory`, which doesn't need to be exposed, and also to keep on the module `dynamic-data-lists-api` only interfaces.

Updated Liferay Portal's Portlet API Implementation

- **Date:** 2018-May-10
- **JIRA Ticket:** LPS-73282

What changed? Liferay Portal 7.1 CE GA1 provides the Portlet 3.0 API dependency in the runtime classpath. Previous versions provided the Portlet 2.0 API.

Full support for Portlet 3.0 will not be available until Liferay Portal 7.1 CE GA4 is released. This will serve as a developer preview for the technology until it's officially promoted in Liferay Portal 7.2.

Who is affected? This affects developers planning to upgrade custom portlets from earlier versions of Liferay Portal.

How should I update my code? There are three development use-cases to plan for:

JSP Considerations Portlet 3.0 is a binary-backward-compatible upgrade. This means that Java source that was built against `portlet-api-2.0.0.jar` is compatible at runtime. Since JSP files are typically not compiled until the first request, however, they do not fall under the category of pre-compiled source.

Specifically, if a JSP contains a Java scriptlet that calls `MimeResponse.createActionURL()` and `MimeResponse.createRenderURL()`, then there is a possibility that the JSP will fail to compile or throw a `ClassCastException` at runtime. This is because the return type of these methods has changed.

For example, a Liferay Portal sample portlet's `view.jsp` had to be changed from

```
<aui:form action="<%= renderResponse.createActionURL() %>" method="post" name="fm">
```

to

```
<aui:form action="<%= (PortletURL)renderResponse.createActionURL() %>" method="post" name="fm">
```

Upgrade Considerations To take advantage of new features in Portlet 3.0, you must rebuild portlet projects against the `portlet-api-3.0.0.jar` dependency and *opt-in* by specifying version 3.0 in one of two ways:

1. Add the following tag in your portlet's `portlet.xml` file:

```
<portlet-app version="3.0">
```

2. Add the following property in your portlet's `@Component` tag:

```
@Component(
    property = {
        "javax.portlet.version=3.0"
    },
    service = Portlet.class
)
```

In addition, you must opt-in to new JSP features by specifying the Portlet 3.0 tag library in your JSP views. For example,

```
<%@ taglib uri="http://xmlns.jcp.org/portlet_3_0" prefix="portlet" %>
```

JSPs that opt-in with the new tag library may encounter JSP compilation problems related to the `<portlet:defineObjects>` tag. Specifically, if JSPs reference variables with the following names in Java scriptlets, then a JSP compilation will occur:

- `actionParams`
- `clientDataRequest`
- `cookies`
- `contextPath`
- `locale`
- `locales`
- `mutableRenderParams`
- `namespace`
- `portletContext`
- `portletMode`
- `portletRequest`
- `portletResponse`
- `resourceParams`
- `windowId`
- `windowState`
- `stateAwareResponse`

With the Portlet API 3.0 implementation, these variables are already added to this context by default, so attempting to initialize them in the JSP would duplicate them. Therefore, your JSP scriptlets adding them should be removed.

For example, JSP scriptlets like the following had to be removed from several of Liferay Portal's out-of-the-box portlets' `view.jsp`:

```
<%=
PortletRequest portletRequest = (PortletRequest)request.getAttribute(JavaConstants.JAVAX_PORTLET_REQUEST);

PortletResponse portletResponse = (PortletResponse)request.getAttribute(JavaConstants.JAVAX_PORTLET_RESPONSE);

String namespace = AUIUtil.getNamespace(portletRequest, portletResponse);

if (Validator.isNull(namespace)) {
    namespace = AUIUtil.getNamespace(request);
}
%>
```

JSF Considerations JSF Portlets must be upgraded to the latest version of Liferay Faces Bridge, which is planned for release in Q4, 2018. Download and upgrade instructions will be made available at https://www.liferayfaces.org at that time.

Why was this change made? This change provides the latest features offered by the Portlet 3.0 Specification, which was released in early 2017.

Changed the From Last Publish Date Option in Staging

- **Date:** 2018-Jun-06
- **JIRA Ticket:** LPS-81695

What changed? The *From Last Publish Date* option used in the publication process has programmatically changed.

Who is affected? This affects anyone who implemented Staging support for their custom entities.

How should I update my code? You must create a `*StagingModelListener` class for your custom entity, which extends the `com.liferay.portal.kernel.model.BaseModelListener`. You can examine the `BlogsEntryStagingModelListener` class as an example.

You must also update the `doPrepareManifestSummary` method in your custom `*PortletDataHandler` to use the `populateLastPublishDateCounts` method from the `com.liferay.exportimport.internal.staging.StagingImpl`, in case of a *From Last Publish Date* publication. See the `BlogsPortletDataHandler` as an example.

Why was this change made? It was hard to collect which entities should be published to the live site. Instead of running queries to find the contents that were modified since the last publication, now changesets are used to track this information.

Changed the Dependency for the liferay-util:html-top JSP tag

- **Date:** 2018-Jun-07
- **JIRA Ticket:** LPS-81983

What changed? The usage of `portal-kernel`'s `StringBundler` has been deprecated in favor of Liferay's Petra `StringBundler`.

Who is affected? This affects anyone using the `<liferay-util:html-top>` JSP tag.

How should I update my code? You must add the following dependency in your build file for your JSPs to compile successfully:

build.gradle:

```
dependencies {
    ...
    compileOnly group: "com.liferay", name: "com.liferay.petra.string", version: "1.2.0"
    ...
}
```

pom.xml:

```
<dependency>
    <groupId>com.liferay</groupId>
    <artifactId>com.liferay.petra.string</artifactId>
    <version>1.2.0</version>
    <scope>provided</scope>
</dependency>
```

Why was this change made? This change helps stabilize the foundation of Liferay Portal's utilities.

Decoupled Several Classes from PortletURLImpl

- **Date:** 2018-Jun-08
- **JIRA Ticket:** LPS-82119

What changed? All classes implementing `javax.portlet.BaseURL` have had their inheritance hierarchy change. These classes include

- `PortletURLImplWrapper`
- `LiferayStrutsPortletURLImpl`
- `StrutsActionPortletURL`

Who is affected? This affects code that attempts to subclass or create a new instance of the classes listed previously.

How should I update my code? You must refactor the constructors of your affected classes to receive `com.liferay.portal.kernel.portlet.LiferayPortletResponse` instead of `com.liferay.portlet.PortletResponseImp`

In addition, their class hierarchies must be changed. For example, the `com.liferay.portal.struts.StrutsActionP` class hierarchy was changed from

- `com.liferay.portlet.PortletURLImpl`
- `com.liferay.portlet.PortletURLImplWrapper`
- `com.liferay.portal.struts.StrutsActionPortletURL`

to

- `javax.portlet.filter.RenderStateWrapper`
- `javax.portlet.filter.BaseURLWrapper`
- `javax.portlet.filter.PortletURLWrapper`
- `com.liferay.portal.kernel.portlet.LiferayPortletURLWrapper`
- `com.liferay.portlet.PortletURLImplWrapper`
- `com.liferay.portal.struts.StrutsActionPortletURL`

Why was this change made? This change corrects a best practice violation regarding implementation-specific details being included within an API.

Changed the Request Object in Web Content Templates

- **Date:** 2018-Jun-12
- **JIRA Ticket:** LPS-77766

What changed? The request object is no longer accessible as a map, but rather, as an object of type `javax.servlet.http.HttpServletRequest`.

Who is affected? This affects users with Web Content templates that access request parameters as a map like this:

```
<#assign containerId = request["theme-display"]["portlet-display"]["instance-id"] >
```

How should I update my code? To keep retrieving the request parameter values as a map, `requestMap` must be used instead:

```
<#assign containerId = requestMap["theme-display"]["portlet-display"]["instance-id"] >
```

Why was this change made? This was done to allow template context contributors to work in Web Content templates.

Disabled Access to Gogo Shell Using Telnet

- **Date:** 2018-Jun-25
- **JIRA Ticket:** LPS-82849

What changed? The ability to access and interact with Liferay Portal's OSGi framework using the Gogo shell via your system's telnet client has been disabled.

Who is affected? This affects anyone who used their system's telnet client to access the Gogo shell, or leveraged the Gogo shell in external plugins/tooling using the telnet client.

How should I update my code? Liferay Portal now offers the Gogo Shell portlet, which you can access in the Control Panel → *Configuration* → *Gogo Shell*.

If you prefer using your telnet client to access the Gogo shell, you must enable Developer Mode. You can do this by creating a `portal-ext.properties` file in your Liferay home folder and adding the following property:

```
include-and-override=portal-developer.properties
```

Developer Mode is enabled upon starting your app server.

Why was this change made? This was done to strengthen Liferay Portal's security due to potential XXE/SSRF vulnerabilities.

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 relied on the portlet description's value already being escaped and used it to generate HTML. In that case, a small UI change might be observed as some characters could become 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 using `HtmlUtil.escape`.

Why was this change made? This change corrects a best practice violation regarding content escaping.

Changed modelName Attribute to be Mandatory in liferay-ui:input-permissions Taglib

- **Date:** 2018-Oct-04
- **JIRA Ticket:** LPS-85998

What changed? Previously, the taglib `liferay-ui:input-permissions` could be used without providing the attribute `modelName`. Now the attribute `modelName` is mandatory.

Who is affected? This affects any developer who used the taglib `liferay-ui:input-permissions` in their own portlets and was not setting the `modelName` attribute of the taglib.

How should I update my code? You should invoke the taglib providing the model name to which you are assigning the permissions.

Why was this change made? This change removes old logic that is no longer used in Liferay Portal.

Liferay `AssetEntries_AssetCategories` Is No Longer Used

- **Date:** 2019-Sep-11
- **JIRA Tickets:** LPS-99973, LPS-76488

What changed? Previously, Liferay used a mapping table and a corresponding interface for the relationship between `AssetEntry` and `AssetCategory` in `AssetEntryLocalService` and `AssetCategoryLocalService`. This mapping table and the corresponding interface have been replaced by the table `AssetEntryAssetCategoryRel` and the service `AssetEntryAssetCategoryRelLocalService`.

Who is affected? This affects any content or code that relies on calling the old interfaces for the `AssetEntries_AssetCategories` relationship, through the `AssetEntryLocalService` and `AssetCategoryLocalService`.

How should I update my code? Use the new methods in `AssetEntryAssetCategoryRelLocalService` to retrieve the same data as before. The method signatures haven't changed; they have just been relocated to a different service.

Example

Old way:

```
List<AssetEntry> entries =
AssetEntryLocalServiceUtil.getAssetCategoryAssetEntries(categoryId);

for (AssetEntry entry: entries) {
  ...
}
```

New way:

```
long[] assetEntryPKs =
_assetEntryAssetCategoryRelLocalService.getAssetEntryPrimaryKeys(assetCategoryId);

for (long assetEntryPK: assetEntryPKs) {
  AssetEntry = _assetEntryLocalService.getEntry(assetEntryPK);
  ...
}

...

@Reference
private AssetEntryAssetCategoryRelLocalService _assetEntryAssetCategoryRelLocalService;

@Reference
private AssetEntryLocalService _assetEntryLocalService;
```

Why was this change made? This change was made due to changes resulting from LPS-76488, which let developers control the order of a list of assets for a given category.

Removed Cache Bootstrap Feature

- **Date:** 2020-Jan-08
- **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.

Web Content Description Field Is Controlled by AlloyEditor

- **Date:** 2020-Apr-07
- **JIRA Ticket:** LPS-71850

What changed? Previously, the Web Content description field was a plain text field. This field is now managed by AlloyEditor, so any HTML characters entered into the field are escaped and rendered as plain text instead of HTML.

Who is affected? This affects explicit HTML tags in Web Content descriptions that a developer expects to be rendered as regular HTML tags by the browser.

How should I update my code? If you want these values rendered as HTML, you must unescape them using the proper unescape sequence: `HtmlUtil.unescape`.

For example, the FreeMarker expression `${.vars['reserved-article-description'].data}` should be unescaped like `${htmlUtil.unescape(.vars['reserved-article-description'].data)}`

Why was this change made? This change was made to take advantage of the AlloyEditor's styling and formatting tools in Web Content description fields.